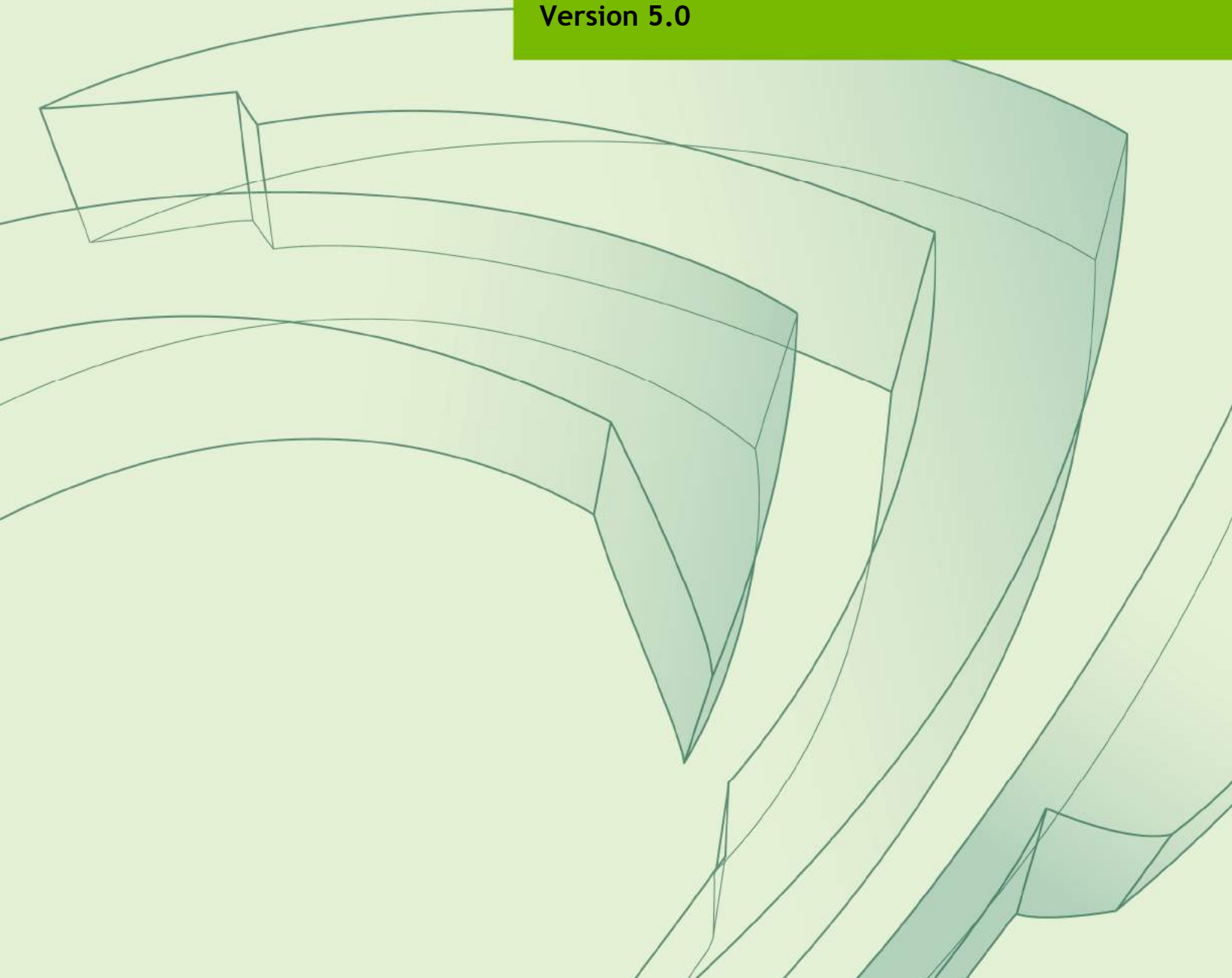
CUDA API REFERENCE MANUAL

October 2012

Version 5.0



Contents

Chapter 1

API synchronization behavior

1.1 Memcpy

The API provides memcpy/memset functions in both synchronous and asynchronous forms, the latter having an *"Async"* suffix. This is a misnomer as each function may exhibit synchronous or asynchronous behavior depending on the arguments passed to the function. In the reference documentation, each memcpy function is categorized as *synchronous* or *asynchronous*, corresponding to the definitions below.

1.1.1 Synchronous

1. For transfers from pageable host memory to device memory, a stream sync is performed before the copy is initiated. The function will return once the pageable buffer has been copied to the staging memory for DMA transfer to device memory, but the DMA to final destination may not have completed.
2. For transfers from pinned host memory to device memory, the function is synchronous with respect to the host.
3. For transfers from device to either pageable or pinned host memory, the function returns only once the copy has completed.
4. For transfers from device memory to device memory, no host-side synchronization is performed.
5. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.

1.1.2 Asynchronous

1. For transfers from pageable host memory to device memory, host memory is copied to a staging buffer immediately (no device synchronization is performed). The function will return once the pageable buffer has been copied to the staging memory. The DMA transfer to final destination may not have completed.
2. For transfers between pinned host memory and device memory, the function is fully asynchronous.
3. For transfers from device memory to pageable host memory, the function will return only once the copy has completed.
4. For all other transfers, the function is fully asynchronous. If pageable memory must first be staged to pinned memory, this will be handled asynchronously with a worker thread.
5. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.

1.2 Memset

The cudaMemset functions are asynchronous with respect to the host except when the target memory is pinned host memory. The *Async* versions are always asynchronous with respect to the host.

1.3 Kernel Launches

Kernel launches are asynchronous with respect to the host. Details of concurrent kernel execution and data transfers can be found in the CUDA Programmers Guide.

Chapter 2

Deprecated List

Global cudaThreadExit

Global cudaThreadGetCacheConfig

Global cudaThreadGetLimit

Global cudaThreadSetCacheConfig

Global cudaThreadSetLimit

Global cudaThreadSynchronize

Global cudaGLMapBufferObject This function is deprecated as of CUDA 3.0.

Global cudaGLMapBufferObjectAsync This function is deprecated as of CUDA 3.0.

Global cudaGLRegisterBufferObject This function is deprecated as of CUDA 3.0.

Global cudaGLSetBufferObjectMapFlags This function is deprecated as of CUDA 3.0.

Global cudaGLSetGLDevice This function is deprecated as of CUDA 5.0.

Global cudaGLUnmapBufferObject This function is deprecated as of CUDA 3.0.

Global cudaGLUnmapBufferObjectAsync This function is deprecated as of CUDA 3.0.

Global cudaGLUnregisterBufferObject This function is deprecated as of CUDA 3.0.

Global cudaD3D9MapResources This function is deprecated as of CUDA 3.0.

Global cudaD3D9RegisterResource This function is deprecated as of CUDA 3.0.

Global cudaD3D9ResourceGetMappedArray This function is deprecated as of CUDA 3.0.

Global cudaD3D9ResourceGetMappedPitch This function is deprecated as of CUDA 3.0.

Global cudaD3D9ResourceGetMappedPointer This function is deprecated as of CUDA 3.0.

Global cudaD3D9ResourceGetMappedSize This function is deprecated as of CUDA 3.0.

Global cudaD3D9ResourceGetSurfaceDimensions This function is deprecated as of CUDA 3.0.

Global cudaD3D9ResourceSetMapFlags This function is deprecated as of CUDA 3.0.

Global cudaD3D9UnmapResources This function is deprecated as of CUDA 3.0.

Global cudaD3D9UnregisterResource This function is deprecated as of CUDA 3.0.

Global cudaD3D10GetDirect3DDevice This function is deprecated as of CUDA 5.0.

Global cudaD3D10MapResources This function is deprecated as of CUDA 3.0.

Global cudaD3D10RegisterResource This function is deprecated as of CUDA 3.0.

Global cudaD3D10ResourceGetMappedArray This function is deprecated as of CUDA 3.0.

Global cudaD3D10ResourceGetMappedPitch This function is deprecated as of CUDA 3.0.

Global cudaD3D10ResourceGetMappedPointer This function is deprecated as of CUDA 3.0.

Global cudaD3D10ResourceGetMappedSize This function is deprecated as of CUDA 3.0.

Global cudaD3D10ResourceGetSurfaceDimensions This function is deprecated as of CUDA 3.0.

Global cudaD3D10ResourceSetMapFlags This function is deprecated as of CUDA 3.0.

Global cudaD3D10SetDirect3DDevice This function is deprecated as of CUDA 5.0.

Global cudaD3D10UnmapResources This function is deprecated as of CUDA 3.0.

Global cudaD3D10UnregisterResource This function is deprecated as of CUDA 3.0.

Global cudaD3D11GetDirect3DDevice This function is deprecated as of CUDA 5.0.

Global cudaD3D11SetDirect3DDevice This function is deprecated as of CUDA 5.0.

Global cudaErrorPriorLaunchFailure This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global cudaErrorAddressOfConstant This error return is deprecated as of CUDA 3.1. Variables in constant memory may now have their address taken by the runtime via cudaGetSymbolAddress().

Global cudaErrorTextureFetchFailed This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global cudaErrorTextureNotBound This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global cudaErrorSynchronizationError This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global cudaErrorMixedDeviceExecution This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global cudaErrorNotYetImplemented This error return is deprecated as of CUDA 4.1.

Global cudaErrorMemoryValueTooLarge This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Global cudaErrorProfilerNotInitialized This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via cudaProfilerStart or cudaProfilerStop without initialization.

Global cudaErrorProfilerAlreadyStarted This error return is deprecated as of CUDA 5.0. It is no longer an error to call cudaProfilerStart() when profiling is already enabled.

Global cudaErrorProfilerAlreadyStopped This error return is deprecated as of CUDA 5.0. It is no longer an error to call cudaProfilerStop() when profiling is already disabled.

Global cudaErrorApiFailureBase This error return is deprecated as of CUDA 4.1.

Global cudaDeviceBlockingSync This flag was deprecated as of CUDA 4.0 and replaced with cudaDeviceScheduleBlockingSync.

Global CU_CTX_BLOCKING_SYNC This flag was deprecated as of CUDA 4.0 and was replaced with CU_-CTX_SCHED_BLOCKING_SYNC.

Global CUDA_ERROR_PROFILER_NOT_INITIALIZED This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via cuProfilerStart or cuProfilerStop without initialization.

Global CUDA_ERROR_PROFILER_ALREADY_STARTED This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStart() when profiling is already enabled.

Global CUDA_ERROR_PROFILER_ALREADY_STOPPED This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStop() when profiling is already disabled.

Global CUDA_ERROR_CONTEXT_ALREADY_CURRENT This error return is deprecated as of CUDA 3.2. It is no longer an error to attempt to push the active context via cuCtxPushCurrent().

Global cuDeviceComputeCapability

Global cuDeviceGetProperties

Global cuCtxAttach

Global cuCtxDetach

Global cuFuncSetBlockShape

Global cuFuncSetSharedSize

Global cuLaunch

Global cuLaunchGrid

Global cuLaunchGridAsync

Global cuParamSetf

Global cuParamSeti

Global cuParamSetSize

Global cuParamSetTexRef

Global cuParamSetv

Global cuTexRefCreate

Global cuTexRefDestroy

Global cuGLCtxCreate This function is deprecated as of Cuda 5.0.

Global cuGLInit This function is deprecated as of Cuda 3.0.

Global cuGLMapBufferObject This function is deprecated as of Cuda 3.0.

Global cuGLMapBufferObjectAsync This function is deprecated as of Cuda 3.0.

Global cuGLRegisterBufferObject This function is deprecated as of Cuda 3.0.

Global cuGLSetBufferObjectMapFlags This function is deprecated as of Cuda 3.0.

Global cuGLUnmapBufferObject This function is deprecated as of Cuda 3.0.

Global cuGLUnmapBufferObjectAsync This function is deprecated as of Cuda 3.0.

Global cuGLUnregisterBufferObject This function is deprecated as of Cuda 3.0.

Global cuD3D9MapResources This function is deprecated as of Cuda 3.0.

Global cuD3D9RegisterResource This function is deprecated as of Cuda 3.0.

Global cuD3D9ResourceGetMappedArray This function is deprecated as of Cuda 3.0.

Global cuD3D9ResourceGetMappedPitch This function is deprecated as of Cuda 3.0.

Global cuD3D9ResourceGetMappedPointer This function is deprecated as of Cuda 3.0.

Global cuD3D9ResourceGetMappedSize This function is deprecated as of Cuda 3.0.

Global cuD3D9ResourceGetSurfaceDimensions This function is deprecated as of Cuda 3.0.

Global cuD3D9ResourceSetMapFlags This function is deprecated as of Cuda 3.0.

Global cuD3D9UnmapResources This function is deprecated as of Cuda 3.0.

Global cuD3D9UnregisterResource This function is deprecated as of Cuda 3.0.

Global cuD3D10CtxCreate This function is deprecated as of Cuda 5.0.

Global cuD3D10CtxCreateOnDevice This function is deprecated as of Cuda 5.0.

Global cuD3D10GetDirect3DDevice This function is deprecated as of Cuda 5.0.

Global cuD3D10MapResources This function is deprecated as of Cuda 3.0.

Global cuD3D10RegisterResource This function is deprecated as of Cuda 3.0.

Global cuD3D10ResourceGetMappedArray This function is deprecated as of Cuda 3.0.

Global cuD3D10ResourceGetMappedPitch This function is deprecated as of Cuda 3.0.

Global cuD3D10ResourceGetMappedPointer This function is deprecated as of Cuda 3.0.

Global cuD3D10ResourceGetMappedSize This function is deprecated as of Cuda 3.0.

Global cuD3D10ResourceGetSurfaceDimensions This function is deprecated as of Cuda 3.0.

Global cuD3D10ResourceSetMapFlags This function is deprecated as of Cuda 3.0.

Global cuD3D10UnmapResources This function is deprecated as of Cuda 3.0.

Global cuD3D10UnregisterResource This function is deprecated as of Cuda 3.0.

Global cuD3D11CtxCreate This function is deprecated as of Cuda 5.0.

Global cuD3D11CtxCreateOnDevice This function is deprecated as of Cuda 5.0.

Global cuD3D11GetDirect3DDevice This function is deprecated as of Cuda 5.0.

Chapter 3

Module Index

3.1 Modules

Here is a list of all modules:

Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 5

Module Documentation

5.1 Device Management

Functions

- cudaError_t cudaChooseDevice (int *device, const struct cudaDeviceProp *prop)
  Select compute-device which best matches criteria.
- __cudart_builtin__ cudaError_t cudaDeviceGetAttribute (int *value, enum cudaDeviceAttr attr, int device)
  Returns information about the device.
- cudaError_t cudaDeviceGetByPCIBusId (int *device, char *pciBusId)
  Returns a handle to a compute device.
- __cudart_builtin__ cudaError_t cudaDeviceGetCacheConfig (enum cudaFuncCache *pCacheConfig)
  Returns the preferred cache configuration for the current device.
- __cudart_builtin__ cudaError_t cudaDeviceGetLimit (size_t *pValue, enum cudaLimit limit)
  Returns resource limits.
- cudaError_t cudaDeviceGetPCIBusId (char *pciBusId, int len, int device)
  Returns a PCI Bus Id string for the device.
- __cudart_builtin__ cudaError_t cudaDeviceGetSharedMemConfig (enum cudaSharedMemConfig *pConfig)
  Returns the shared memory configuration for the current device.
- cudaError_t cudaDeviceReset (void)
  Destroy all allocations and reset all state on the current device in the current process.
- cudaError_t cudaDeviceSetCacheConfig (enum cudaFuncCache cacheConfig)
  Sets the preferred cache configuration for the current device.
- cudaError_t cudaDeviceSetLimit (enum cudaLimit limit, size_t value)
  Set resource limits.
- cudaError_t cudaDeviceSetSharedMemConfig (enum cudaSharedMemConfig config)

Sets the shared memory configuration for the current device.

- __cudart_builtin__ cudaError_t cudaDeviceSynchronize (void)

  Wait for compute device to finish.

- __cudart_builtin__ cudaError_t cudaGetDevice (int *device)

  Returns which device is currently being used.

- __cudart_builtin__ cudaError_t cudaGetDeviceCount (int *count)

  Returns the number of compute-capable devices.

- __cudart_builtin__ cudaError_t cudaGetDeviceProperties (struct cudaDeviceProp *prop, int device)

  Returns information about the compute-device.

- cudaError_t cudaIpcCloseMemHandle (void *devPtr)

  Close memory mapped with cudaIpcOpenMemHandle.

- cudaError_t cudaIpcGetEventHandle (cudaIpcEventHandle_t *handle, cudaEvent_t event)

  Gets an interprocess handle for a previously allocated event.

- cudaError_t cudaIpcGetMemHandle (cudaIpcMemHandle_t *handle, void *devPtr)
- cudaError_t cudaIpcOpenEventHandle (cudaEvent_t *event, cudaIpcEventHandle_t handle)

  Opens an interprocess event handle for use in the current process.

- cudaError_t cudaIpcOpenMemHandle (void **devPtr, cudaIpcMemHandle_t handle, unsigned int flags)
- cudaError_t cudaSetDevice (int device)

  Set device to be used for GPU executions.

- cudaError_t cudaSetDeviceFlags (unsigned int flags)

  Sets flags to be used for device executions.

- cudaError_t cudaSetValidDevices (int *device_arr, int len)

  Set a list of devices that can be used for CUDA.

5.1.1 Detailed Description

This section describes the device management functions of the CUDA runtime application programming interface.

5.1.2 Function Documentation

5.1.2.1 cudaError_t cudaChooseDevice (int * *device*, const struct cudaDeviceProp * *prop*)

Returns in `*device` the device which has properties that best match `*prop`.

Parameters:

device - Device with best match

prop - Desired device properties

Returns:

cudaSuccess, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGetDeviceCount, cudaGetDevice, cudaSetDevice, cudaGetDeviceProperties

5.1.2.2 __cudart_builtin__ cudaError_t cudaDeviceGetAttribute (int * *value*, enum cudaDeviceAttr *attr*, int *device*)

Returns in `*value` the integer value of the attribute `attr` on device `device`. The supported attributes are:

- cudaDevAttrMaxThreadsPerBlock: Maximum number of threads per block;
- cudaDevAttrMaxBlockDimX: Maximum x-dimension of a block;
- cudaDevAttrMaxBlockDimY: Maximum y-dimension of a block;
- cudaDevAttrMaxBlockDimZ: Maximum z-dimension of a block;
- cudaDevAttrMaxGridDimX: Maximum x-dimension of a grid;
- cudaDevAttrMaxGridDimY: Maximum y-dimension of a grid;
- cudaDevAttrMaxGridDimZ: Maximum z-dimension of a grid;
- cudaDevAttrMaxSharedMemoryPerBlock: Maximum amount of shared memory available to a thread block in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- cudaDevAttrTotalConstantMemory: Memory available on device for __constant__ variables in a CUDA C kernel in bytes;
- cudaDevAttrWarpSize: Warp size in threads;
- cudaDevAttrMaxPitch: Maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through cudaMallocPitch();
- cudaDevAttrMaxTexture1DWidth: Maximum 1D texture width;
- cudaDevAttrMaxTexture1DLinearWidth: Maximum width for a 1D texture bound to linear memory;
- cudaDevAttrMaxTexture1DMipmappedWidth: Maximum mipmapped 1D texture width;
- cudaDevAttrMaxTexture2DWidth: Maximum 2D texture width;
- cudaDevAttrMaxTexture2DHeight: Maximum 2D texture height;
- cudaDevAttrMaxTexture2DLinearWidth: Maximum width for a 2D texture bound to linear memory;
- cudaDevAttrMaxTexture2DLinearHeight: Maximum height for a 2D texture bound to linear memory;
- cudaDevAttrMaxTexture2DLinearPitch: Maximum pitch in bytes for a 2D texture bound to linear memory;
- cudaDevAttrMaxTexture2DMipmappedWidth: Maximum mipmapped 2D texture width;
- cudaDevAttrMaxTexture2DMipmappedHeight: Maximum mipmapped 2D texture height;

- cudaDevAttrMaxTexture3DWidth: Maximum 3D texture width;
- cudaDevAttrMaxTexture3DHeight: Maximum 3D texture height;
- cudaDevAttrMaxTexture3DDepth: Maximum 3D texture depth;
- cudaDevAttrMaxTexture3DWidthAlt: Alternate maximum 3D texture width, 0 if no alternate maximum 3D texture size is supported;
- cudaDevAttrMaxTexture3DHeightAlt: Alternate maximum 3D texture height, 0 if no alternate maximum 3D texture size is supported;
- cudaDevAttrMaxTexture3DDepthAlt: Alternate maximum 3D texture depth, 0 if no alternate maximum 3D texture size is supported;
- cudaDevAttrMaxTextureCubemapWidth: Maximum cubemap texture width or height;
- cudaDevAttrMaxTexture1DLayeredWidth: Maximum 1D layered texture width;
- cudaDevAttrMaxTexture1DLayeredLayers: Maximum layers in a 1D layered texture;
- cudaDevAttrMaxTexture2DLayeredWidth: Maximum 2D layered texture width;
- cudaDevAttrMaxTexture2DLayeredHeight: Maximum 2D layered texture height;
- cudaDevAttrMaxTexture2DLayeredLayers: Maximum layers in a 2D layered texture;
- cudaDevAttrMaxTextureCubemapLayeredWidth: Maximum cubemap layered texture width or height;
- cudaDevAttrMaxTextureCubemapLayeredLayers: Maximum layers in a cubemap layered texture;
- cudaDevAttrMaxSurface1DWidth: Maximum 1D surface width;
- cudaDevAttrMaxSurface2DWidth: Maximum 2D surface width;
- cudaDevAttrMaxSurface2DHeight: Maximum 2D surface height;
- cudaDevAttrMaxSurface3DWidth: Maximum 3D surface width;
- cudaDevAttrMaxSurface3DHeight: Maximum 3D surface height;
- cudaDevAttrMaxSurface3DDepth: Maximum 3D surface depth;
- cudaDevAttrMaxSurface1DLayeredWidth: Maximum 1D layered surface width;
- cudaDevAttrMaxSurface1DLayeredLayers: Maximum layers in a 1D layered surface;
- cudaDevAttrMaxSurface2DLayeredWidth: Maximum 2D layered surface width;
- cudaDevAttrMaxSurface2DLayeredHeight: Maximum 2D layered surface height;
- cudaDevAttrMaxSurface2DLayeredLayers: Maximum layers in a 2D layered surface;
- cudaDevAttrMaxSurfaceCubemapWidth: Maximum cubemap surface width;
- cudaDevAttrMaxSurfaceCubemapLayeredWidth: Maximum cubemap layered surface width;
- cudaDevAttrMaxSurfaceCubemapLayeredLayers: Maximum layers in a cubemap layered surface;
- cudaDevAttrMaxRegistersPerBlock: Maximum number of 32-bit registers available to a thread block; this number is shared by all thread blocks simultaneously resident on a multiprocessor;
- cudaDevAttrClockRate: Peak clock frequency in kilohertz;

- cudaDevAttrTextureAlignment: Alignment requirement; texture base addresses aligned to textureAlign bytes do not need an offset applied to texture fetches;
- cudaDevAttrTexturePitchAlignment: Pitch alignment requirement for 2D texture references bound to pitched memory;
- cudaDevAttrGpuOverlap: 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not;
- cudaDevAttrMultiProcessorCount: Number of multiprocessors on the device;
- cudaDevAttrKernelExecTimeout: 1 if there is a run time limit for kernels executed on the device, or 0 if not;
- cudaDevAttrIntegrated: 1 if the device is integrated with the memory subsystem, or 0 if not;
- cudaDevAttrCanMapHostMemory: 1 if the device can map host memory into the CUDA address space, or 0 if not;
- cudaDevAttrComputeMode: Compute mode is the compute mode that the device is currently in. Available modes are as follows:
  - cudaComputeModeDefault: Default mode - Device is not restricted and multiple threads can use cudaSetDevice() with this device.
  - cudaComputeModeExclusive: Compute-exclusive mode - Only one thread will be able to use cudaSetDevice() with this device.
  - cudaComputeModeProhibited: Compute-prohibited mode - No threads can use cudaSetDevice() with this device.
  - cudaComputeModeExclusiveProcess: Compute-exclusive-process mode - Many threads in one process will be able to use cudaSetDevice() with this device.
- cudaDevAttrConcurrentKernels: 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;
- cudaDevAttrEccEnabled: 1 if error correction is enabled on the device, 0 if error correction is disabled or not supported by the device;
- cudaDevAttrPciBusId: PCI bus identifier of the device;
- cudaDevAttrPciDeviceId: PCI device (also known as slot) identifier of the device;
- cudaDevAttrTccDriver: 1 if the device is using a TCC driver. TCC is only available on Tesla hardware running Windows Vista or later;
- cudaDevAttrMemoryClockRate: Peak memory clock frequency in kilohertz;
- cudaDevAttrGlobalMemoryBusWidth: Global memory bus width in bits;
- cudaDevAttrL2CacheSize: Size of L2 cache in bytes. 0 if the device doesn't have L2 cache;
- cudaDevAttrMaxThreadsPerMultiProcessor: Maximum resident threads per multiprocessor;
- cudaDevAttrUnifiedAddressing: 1 if the device shares a unified address space with the host, or 0 if not;
- cudaDevAttrComputeCapabilityMajor: Major compute capability version number;
- cudaDevAttrComputeCapabilityMinor: Minor compute capability version number;

Parameters:

value - Returned device attribute value

attr - Device attribute to query

device - Device number to query

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGetDeviceCount, cudaGetDevice, cudaSetDevice, cudaChooseDevice, cudaGetDeviceProperties

5.1.2.3 cudaError_t cudaDeviceGetByPCIBusId (int * *device*, char * *pciBusId*)

Returns in `*device` a device ordinal given a PCI bus ID string.

Parameters:

device - Returned device ordinal

pciBusId - String in one of the following forms: [domain]:[bus]:[device].[function] [domain]:[bus]:[device] [bus]:[device].[function] where `domain`, `bus`, `device`, and `function` are all hexadecimal values

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceGetPCIBusId

5.1.2.4 __cudart_builtin__ cudaError_t cudaDeviceGetCacheConfig (enum cudaFuncCache * *pCacheConfig*)

On devices where the L1 cache and shared memory use the same hardware resources, this returns through `pCacheConfig` the preferred cache configuration for the current device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute functions.

This will return a `pCacheConfig` of cudaFuncCachePreferNone on devices where the size of the L1 cache and shared memory are fixed.

The supported cache configurations are:

- cudaFuncCachePreferNone: no preference for shared memory or L1 (default)
- cudaFuncCachePreferShared: prefer larger shared memory and smaller L1 cache
- cudaFuncCachePreferL1: prefer larger L1 cache and smaller shared memory

Parameters:

pCacheConfig - Returned cache configuration

Returns:

cudaSuccess, cudaErrorInitializationError

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceSetCacheConfig, cudaFuncSetCacheConfig (C API), cudaFuncSetCacheConfig (C++ API)

5.1.2.5 __cudart_builtin__ cudaError_t cudaDeviceGetLimit (size_t * *pValue*, enum cudaLimit *limit*)

Returns in `*pValue` the current size of `limit`. The supported cudaLimit values are:

- cudaLimitStackSize: stack size in bytes of each GPU thread;
- cudaLimitPrintfFifoSize: size in bytes of the shared FIFO used by the printf() and fprintf() device system calls.
- cudaLimitMallocHeapSize: size in bytes of the heap used by the malloc() and free() device system calls;
- cudaLimitDevRuntimeSyncDepth: maximum grid depth at which a thread can isssue the device runtime call cudaDeviceSynchronize() to wait on child grid launches to complete.
- cudaLimitDevRuntimePendingLaunchCount: maximum number of outstanding device runtime launches.

Parameters:

limit - Limit to query

pValue - Returned size of the limit

Returns:

cudaSuccess, cudaErrorUnsupportedLimit, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceSetLimit

5.1.2.6 cudaError_t cudaDeviceGetPCIBusId (char * *pciBusId*, int *len*, int *device*)

Returns an ASCII string identifying the device `dev` in the NULL-terminated string pointed to by `pciBusId`. `len` specifies the maximum length of the string that may be returned.

Parameters:

pciBusId - Returned identifier string for the device in the following format [domain]:[bus]:[device].[function] where `domain`, `bus`, `device`, and `function` are all hexadecimal values. pciBusId should be large enough to store 13 characters including the NULL-terminator.

len - Maximum length of string to store in `name`

device - Device to get identifier string for

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceGetByPCIBusId

5.1.2.7 __cudart_builtin__ cudaError_t cudaDeviceGetSharedMemConfig (enum cudaSharedMemConfig * *pConfig*)

This function will return in `pConfig` the current size of shared memory banks on the current device. On devices with configurable shared memory banks, cudaDeviceSetSharedMemConfig can be used to change this setting, so that all subsequent kernel launches will by default use the new bank size. When cudaDeviceGetSharedMemConfig is called on devices without configurable shared memory, it will return the fixed bank size of the hardware.

The returned bank configurations can be either:

- cudaSharedMemBankSizeFourByte - shared memory bank width is four bytes.
- cudaSharedMemBankSizeEightByte - shared memory bank width is eight bytes.

Parameters:

pConfig - Returned cache configuration

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInitializationError

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceSetCacheConfig, cudaDeviceGetCacheConfig, cudaDeviceSetSharedMemConfig, cudaFuncSetCacheConfig

5.1.2.8 cudaError_t cudaDeviceReset (void)

Explicitly destroys and cleans up all resources associated with the current device in the current process. Any subsequent API call to this device will reinitialize the device.

Note that this function will reset the device immediately. It is the caller's responsibility to ensure that the device is not being accessed by any other host threads from the process when this function is called.

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceSynchronize

5.1.2.9 cudaError_t cudaDeviceSetCacheConfig (enum cudaFuncCache *cacheConfig*)

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `cacheConfig` the preferred cache configuration for the current device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute the function. Any function preference set via cudaFuncSetCacheConfig (C API) or cudaFuncSetCacheConfig (C++ API) will be preferred over this device-wide setting. Setting the device-wide cache configuration to cudaFuncCachePreferNone will cause subsequent kernel launches to prefer to not change the cache configuration unless required to launch the kernel.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- cudaFuncCachePreferNone: no preference for shared memory or L1 (default)
- cudaFuncCachePreferShared: prefer larger shared memory and smaller L1 cache
- cudaFuncCachePreferL1: prefer larger L1 cache and smaller shared memory

Parameters:

cacheConfig - Requested cache configuration

Returns:

cudaSuccess, cudaErrorInitializationError

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceGetCacheConfig, cudaFuncSetCacheConfig (C API), cudaFuncSetCacheConfig (C++ API)

5.1.2.10 cudaError_t cudaDeviceSetLimit (enum cudaLimit *limit*, size_t *value*)

Setting `limit` to `value` is a request by the application to update the current limit maintained by the device. The driver is free to modify the requested value to meet h/w requirements (this could be clamping to minimum or maximum values, rounding up to nearest element size, etc). The application can use cudaDeviceGetLimit() to find out exactly what the limit has been set to.

Setting each cudaLimit has its own specific restrictions, so each is discussed here.

- cudaLimitStackSize controls the stack size in bytes of each GPU thread. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error cudaErrorUnsupportedLimit being returned.

- cudaLimitPrintfFifoSize controls the size in bytes of the shared FIFO used by the printf() and fprintf() device system calls. Setting cudaLimitPrintfFifoSize must be performed before launching any kernel that uses the printf() or fprintf() device system calls, otherwise cudaErrorInvalidValue will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error cudaErrorUnsupportedLimit being returned.

- cudaLimitMallocHeapSize controls the size in bytes of the heap used by the malloc() and free() device system calls. Setting cudaLimitMallocHeapSize must be performed before launching any kernel that uses the malloc() or free() device system calls, otherwise cudaErrorInvalidValue will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error cudaErrorUnsupportedLimit being returned.

- cudaLimitDevRuntimeSyncDepth controls the maximum nesting depth of a grid at which a thread can safely call cudaDeviceSynchronize(). Setting this limit must be performed before any launch of a kernel that uses the device runtime and calls cudaDeviceSynchronize() above the default sync depth, two levels of grids. Calls to cudaDeviceSynchronize() will fail with error code cudaErrorSyncDepthExceeded if the limitation is violated. This limit can be set smaller than the default or up the maximum launch depth of 24. When setting this limit, keep in mind that additional levels of sync depth require the runtime to reserve large amounts of device memory which can no longer be used for user allocations. If these reservations of device memory fail, cudaDeviceSetLimit will return cudaErrorMemoryAllocation, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error cudaErrorUnsupportedLimit being returned.

- cudaLimitDevRuntimePendingLaunchCount controls the maximum number of outstanding device runtime launches that can be made from the current device. A grid is outstanding from the point of launch up until the grid is known to have been completed. Device runtime launches which violate this limitation fail and return cudaErrorLaunchPendingCountExceeded when cudaGetLastError() is called after launch. If more pending launches than the default (2048 launches) are needed for a module using the device runtime, this limit can be increased. Keep in mind that being able to sustain additional pending launches will require the runtime to reserve larger amounts of device memory upfront which can no longer be used for allocations. If these reservations fail, cudaDeviceSetLimit will return cudaErrorMemoryAllocation, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error cudaErrorUnsupportedLimit being returned.

Parameters:

limit - Limit to set
value - Size of limit

Returns:

cudaSuccess, cudaErrorUnsupportedLimit, cudaErrorInvalidValue, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceGetLimit

5.1.2.11 cudaError_t cudaDeviceSetSharedMemConfig (enum cudaSharedMemConfig *config*)

On devices with configurable shared memory banks, this function will set the shared memory bank size which is used for all subsequent kernel launches. Any per-function setting of shared memory set via cudaFuncSetSharedMemConfig will override the device wide setting.

Changing the shared memory configuration between launches may introduce a device side synchronization point.

Changing the shared memory bank size will not increase shared memory usage or affect occupancy of kernels, but may have major effects on performance. Larger bank sizes will allow for greater potential bandwidth to shared memory, but will change what kinds of accesses to shared memory will result in bank conflicts.

This function will do nothing on devices with fixed shared memory bank size.

The supported bank configurations are:

- cudaSharedMemBankSizeDefault: set bank width the device default (currently, four bytes)
- cudaSharedMemBankSizeFourByte: set shared memory bank width to be four bytes natively.
- cudaSharedMemBankSizeEightByte: set shared memory bank width to be eight bytes natively.

Parameters:

config - Requested cache configuration

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInitializationError

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceSetCacheConfig, cudaDeviceGetCacheConfig, cudaDeviceGetSharedMemConfig, cudaFuncSetCacheConfig

5.1.2.12 __cudart_builtin__ cudaError_t cudaDeviceSynchronize (void)

Blocks until the device has completed all preceding requested tasks. cudaDeviceSynchronize() returns an error if one of the preceding tasks has failed. If the cudaDeviceScheduleBlockingSync flag was set for this device, the host thread will block until the device has finished its work.

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceReset

5.1.2.13 __cudart_builtin__ cudaError_t cudaGetDevice (int * *device*)

Returns in `*device` the current device for the calling host thread.

Parameters:

device - Returns the device on which the active host thread executes the device code.

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGetDeviceCount, cudaSetDevice, cudaGetDeviceProperties, cudaChooseDevice

5.1.2.14 __cudart_builtin__ cudaError_t cudaGetDeviceCount (int * *count*)

Returns in `*count` the number of devices with compute capability greater or equal to 1.0 that are available for execution. If there is no such device then cudaGetDeviceCount() will return cudaErrorNoDevice. If no driver can be loaded to determine if any such devices exist then cudaGetDeviceCount() will return cudaErrorInsufficientDriver.

Parameters:

count - Returns the number of devices with compute capability greater or equal to 1.0

Returns:

cudaSuccess, cudaErrorNoDevice, cudaErrorInsufficientDriver

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGetDevice, cudaSetDevice, cudaGetDeviceProperties, cudaChooseDevice

5.1.2.15 __cudart_builtin__ cudaError_t cudaGetDeviceProperties (struct cudaDeviceProp * *prop*, int *device*)

Returns in `*prop` the properties of device `dev`. The cudaDeviceProp structure is defined as:

```
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    int clockRate;
    size_t totalConstMem;
    int major;
    int minor;
    size_t textureAlignment;
    size_t texturePitchAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
```

```
    int integrated;
    int canMapHostMemory;
    int computeMode;
    int maxTexture1D;
    int maxTexture1DMipmap;
    int maxTexture1DLinear;
    int maxTexture2D[2];
    int maxTexture2DMipmap[2];
    int maxTexture2DLinear[3];
    int maxTexture2DGather[2];
    int maxTexture3D[3];
    int maxTextureCubemap;
    int maxTexture1DLayered[2];
    int maxTexture2DLayered[3];
    int maxTextureCubemapLayered[2];
    int maxSurface1D;
    int maxSurface2D[2];
    int maxSurface3D[3];
    int maxSurface1DLayered[2];
    int maxSurface2DLayered[3];
    int maxSurfaceCubemap;
    int maxSurfaceCubemapLayered[2];
    size_t surfaceAlignment;
    int concurrentKernels;
    int ECCEnabled;
    int pciBusID;
    int pciDeviceID;
    int pciDomainID;
    int tccDriver;
    int asyncEngineCount;
    int unifiedAddressing;
    int memoryClockRate;
    int memoryBusWidth;
    int l2CacheSize;
    int maxThreadsPerMultiProcessor;
}
```

where:

- name[256] is an ASCII string identifying the device;
- totalGlobalMem is the total amount of global memory available on the device in bytes;
- sharedMemPerBlock is the maximum amount of shared memory available to a thread block in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- regsPerBlock is the maximum number of 32-bit registers available to a thread block; this number is shared by all thread blocks simultaneously resident on a multiprocessor;
- warpSize is the warp size in threads;
- memPitch is the maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through cudaMallocPitch();
- maxThreadsPerBlock is the maximum number of threads per block;
- maxThreadsDim[3] contains the maximum size of each dimension of a block;
- maxGridSize[3] contains the maximum size of each dimension of a grid;
- clockRate is the clock frequency in kilohertz;
- totalConstMem is the total amount of constant memory available on the device in bytes;
- major, minor are the major and minor revision numbers defining the device's compute capability;

- textureAlignment is the alignment requirement; texture base addresses that are aligned to textureAlignment bytes do not need an offset applied to texture fetches;
- texturePitchAlignment is the pitch alignment requirement for 2D texture references that are bound to pitched memory;
- deviceOverlap is 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not. Deprecated, use instead asyncEngineCount.
- multiProcessorCount is the number of multiprocessors on the device;
- kernelExecTimeoutEnabled is 1 if there is a run time limit for kernels executed on the device, or 0 if not.
- integrated is 1 if the device is an integrated (motherboard) GPU and 0 if it is a discrete (card) component.
- canMapHostMemory is 1 if the device can map host memory into the CUDA address space for use with cudaHostAlloc()/cudaHostGetDevicePointer(), or 0 if not;
- computeMode is the compute mode that the device is currently in. Available modes are as follows:
  - cudaComputeModeDefault: Default mode - Device is not restricted and multiple threads can use cudaSetDevice() with this device.
  - cudaComputeModeExclusive: Compute-exclusive mode - Only one thread will be able to use cudaSetDevice() with this device.
  - cudaComputeModeProhibited: Compute-prohibited mode - No threads can use cudaSetDevice() with this device.
  - cudaComputeModeExclusiveProcess: Compute-exclusive-process mode - Many threads in one process will be able to use cudaSetDevice() with this device.

    If cudaSetDevice() is called on an already occupied `device` with computeMode cudaComputeModeExclusive, cudaErrorDeviceAlreadyInUse will be immediately returned indicating the device cannot be used. When an occupied exclusive mode device is chosen with cudaSetDevice, all subsequent non-device management runtime functions will return cudaErrorDevicesUnavailable.
- maxTexture1D is the maximum 1D texture size.
- maxTexture1DMipmap is the maximum 1D mipmapped texture texture size.
- maxTexture1DLinear is the maximum 1D texture size for textures bound to linear memory.
- maxTexture2D[2] contains the maximum 2D texture dimensions.
- maxTexture2DMipmap[2] contains the maximum 2D mipmapped texture dimensions.
- maxTexture2DLinear[3] contains the maximum 2D texture dimensions for 2D textures bound to pitch linear memory.
- maxTexture2DGather[2] contains the maximum 2D texture dimensions if texture gather operations have to be performed.
- maxTexture3D[3] contains the maximum 3D texture dimensions.
- maxTextureCubemap is the maximum cubemap texture width or height.
- maxTexture1DLayered[2] contains the maximum 1D layered texture dimensions.
- maxTexture2DLayered[3] contains the maximum 2D layered texture dimensions.
- maxTextureCubemapLayered[2] contains the maximum cubemap layered texture dimensions.
- maxSurface1D is the maximum 1D surface size.

- maxSurface2D[2] contains the maximum 2D surface dimensions.
- maxSurface3D[3] contains the maximum 3D surface dimensions.
- maxSurface1DLayered[2] contains the maximum 1D layered surface dimensions.
- maxSurface2DLayered[3] contains the maximum 2D layered surface dimensions.
- maxSurfaceCubemap is the maximum cubemap surface width or height.
- maxSurfaceCubemapLayered[2] contains the maximum cubemap layered surface dimensions.
- surfaceAlignment specifies the alignment requirements for surfaces.
- concurrentKernels is 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;
- ECCEnabled is 1 if the device has ECC support turned on, or 0 if not.
- pciBusID is the PCI bus identifier of the device.
- pciDeviceID is the PCI device (sometimes called slot) identifier of the device.
- pciDomainID is the PCI domain identifier of the device.
- tccDriver is 1 if the device is using a TCC driver or 0 if not.
- asyncEngineCount is 1 when the device can concurrently copy memory between host and device while executing a kernel. It is 2 when the device can concurrently copy memory between host and device in both directions and execute a kernel at the same time. It is 0 if neither of these is supported.
- unifiedAddressing is 1 if the device shares a unified address space with the host and 0 otherwise.
- memoryClockRate is the peak memory clock frequency in kilohertz.
- memoryBusWidth is the memory bus width in bits.
- l2CacheSize is L2 cache size in bytes.
- maxThreadsPerMultiProcessor is the number of maximum resident threads per multiprocessor.

Parameters:

prop - Properties for the specified device

device - Device number to get properties for

Returns:

cudaSuccess, cudaErrorInvalidDevice

See also:

cudaGetDeviceCount, cudaGetDevice, cudaSetDevice, cudaChooseDevice, cudaDeviceGetAttribute

5.1.2.16 cudaError_t cudaIpcCloseMemHandle (void * *devPtr*)

Unmaps memory returnd by cudaIpcOpenMemHandle. The original allocation in the exporting process as well as imported mappings in other processes will be unaffected.

Any resources used to enable peer access will be freed if this is the last mapping using them.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

Parameters:

devPtr - Device pointer returned by cudaIpcOpenMemHandle

Returns:

cudaSuccess, cudaErrorMapBufferObjectFailed, cudaErrorInvalidResourceHandle,

See also:

cudaMalloc, cudaFree, cudaIpcGetEventHandle, cudaIpcOpenEventHandle, cudaIpcGetMemHandle, cudaIpcOpenMemHandle,

5.1.2.17 cudaError_t cudaIpcGetEventHandle (cudaIpcEventHandle_t * *handle*, cudaEvent_t *event*)

Takes as input a previously allocated event. This event must have been created with the cudaEventInterprocess and cudaEventDisableTiming flags set. This opaque handle may be copied into other processes and opened with cudaIpcOpenEventHandle to allow efficient hardware synchronization between GPU work in different processes.

After the event has been been opened in the importing process, cudaEventRecord, cudaEventSynchronize, cudaStreamWaitEvent and cudaEventQuery may be used in either process. Performing operations on the imported event after the exported event has been freed with cudaEventDestroy will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

Parameters:

handle - Pointer to a user allocated cudaIpcEventHandle in which to return the opaque event handle

event - Event allocated with cudaEventInterprocess and cudaEventDisableTiming flags.

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorMemoryAllocation, cudaErrorMapBufferObjectFailed

See also:

cudaEventCreate, cudaEventDestroy, cudaEventSynchronize, cudaEventQuery, cudaStreamWaitEvent, cudaIpcOpenEventHandle, cudaIpcGetMemHandle, cudaIpcOpenMemHandle, cudaIpcCloseMemHandle

5.1.2.18 cudaError_t cudaIpcGetMemHandle (cudaIpcMemHandle_t * *handle*, void * *devPtr*)

/brief Gets an interprocess memory handle for an existing device memory allocation

Takes a pointer to the base of an existing device memory allocation created with cudaMalloc and exports it for use in another process. This is a lightweight operation and may be called multiple times on an allocation without adverse effects.

If a region of memory is freed with cudaFree and a subsequent call to cudaMalloc returns memory with the same device address, cudaIpcGetMemHandle will return a unique handle for the new memory.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

Parameters:

handle - Pointer to user allocated cudaIpcMemHandle to return the handle in.

devPtr - Base pointer to previously allocated device memory

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorMemoryAllocation, cudaErrorMapBufferObjectFailed,

See also:

cudaMalloc, cudaFree, cudaIpcGetEventHandle, cudaIpcOpenEventHandle, cudaIpcOpenMemHandle, cudaIpcCloseMemHandle

5.1.2.19 cudaError_t cudaIpcOpenEventHandle (cudaEvent_t * *event*, cudaIpcEventHandle_t *handle*)

Opens an interprocess event handle exported from another process with cudaIpcGetEventHandle. This function returns a cudaEvent_t that behaves like a locally created event with the cudaEventDisableTiming flag specified. This event must be freed with cudaEventDestroy.

Performing operations on the imported event after the exported event has been freed with cudaEventDestroy will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

Parameters:

event - Returns the imported event

handle - Interprocess handle to open

Returns:

cudaSuccess, cudaErrorMapBufferObjectFailed, cudaErrorInvalidResourceHandle

See also:

cudaEventCreate, cudaEventDestroy, cudaEventSynchronize, cudaEventQuery, cudaStreamWaitEvent, cudaIpcGetEventHandle, cudaIpcGetMemHandle, cudaIpcOpenMemHandle, cudaIpcCloseMemHandle

5.1.2.20 cudaError_t cudaIpcOpenMemHandle (void ** *devPtr*, cudaIpcMemHandle_t *handle*, unsigned int *flags*)

/brief Opens an interprocess memory handle exported from another process and returns a device pointer usable in the local process.

Maps memory exported from another process with cudaIpcGetMemHandle into the current device address space. For contexts on different devices cudaIpcOpenMemHandle can attempt to enable peer access between the devices as if the user called cudaDeviceEnablePeerAccess. This behavior is controlled by the cudaIpcMemLazyEnablePeerAccess flag. cudaDeviceCanAccessPeer can determine if a mapping is possible.

Contexts that may open cudaIpcMemHandles are restricted in the following way. cudaIpcMemHandles from each device in a given process may only be opened by one context per device per other process.

Memory returned from cudaIpcOpenMemHandle must be freed with cudaIpcCloseMemHandle.

Calling cudaFree on an exported memory region before calling cudaIpcCloseMemHandle in the importing context will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

Parameters:

devPtr - Returned device pointer

handle - cudaIpcMemHandle to open

flags - Flags for this operation. Must be specified as cudaIpcMemLazyEnablePeerAccess

Returns:

cudaSuccess, cudaErrorMapBufferObjectFailed, cudaErrorInvalidResourceHandle, cudaErrorTooManyPeers

See also:

cudaMalloc, cudaFree, cudaIpcGetEventHandle, cudaIpcOpenEventHandle, cudaIpcGetMemHandle, cudaIpcCloseMemHandle, cudaDeviceEnablePeerAccess, cudaDeviceCanAccessPeer,

5.1.2.21 cudaError_t cudaSetDevice (int *device*)

Sets `device` as the current device for the calling host thread.

Any device memory subsequently allocated from this host thread using cudaMalloc(), cudaMallocPitch() or cudaMallocArray() will be physically resident on `device`. Any host memory allocated from this host thread using cudaMallocHost() or cudaHostAlloc() or cudaHostRegister() will have its lifetime associated with `device`. Any streams or events created from this host thread will be associated with `device`. Any kernels launched from this host thread using the <<<>>> operator or cudaLaunch() will be executed on `device`.

This call may be made from any host thread, to any device, and at any time. This function will do no synchronization with the previous or new device, and should be considered a very low overhead call.

Parameters:

device - Device on which the active host thread should execute the device code.

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorDeviceAlreadyInUse

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGetDeviceCount, cudaGetDevice, cudaGetDeviceProperties, cudaChooseDevice

5.1.2.22 cudaError_t cudaSetDeviceFlags (unsigned int *flags*)

Records `flags` as the flags to use when initializing the current device. If no device has been made current to the calling thread then `flags` will be applied to the initialization of any device initialized by the calling host thread, unless that device has had its initialization flags set explicitly by this or any host thread.

If the current device has been set and that device has already been initialized then this call will fail with the error cudaErrorSetOnActiveProcess. In this case it is necessary to reset `device` using cudaDeviceReset() before the device's initialization flags may be set.

The two LSBs of the `flags` parameter can be used to control how the CPU thread interacts with the OS scheduler when waiting for results from the device.

- cudaDeviceScheduleAuto: The default value if the `flags` parameter is zero, uses a heuristic based on the number of active CUDA contexts in the process `C` and the number of logical processors in the system `P`. If `C` > `P`, then CUDA will yield to other OS threads when waiting for the device, otherwise CUDA will not yield while waiting for results and actively spin on the processor.

- cudaDeviceScheduleSpin: Instruct CUDA to actively spin when waiting for results from the device. This can decrease latency when waiting for the device, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.

- cudaDeviceScheduleYield: Instruct CUDA to yield its thread when waiting for results from the device. This can increase latency when waiting for the device, but can increase the performance of CPU threads performing work in parallel with the device.

- cudaDeviceScheduleBlockingSync: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the device to finish work.

- cudaDeviceBlockingSync: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the device to finish work.

  Deprecated: This flag was deprecated as of CUDA 4.0 and replaced with cudaDeviceScheduleBlockingSync.

- cudaDeviceMapHost: This flag must be set in order to allocate pinned host memory that is accessible to the device. If this flag is not set, cudaHostGetDevicePointer() will always return a failure code.

- cudaDeviceLmemResizeToMax: Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage.

Parameters:

flags - Parameters for device operation

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorSetOnActiveProcess

See also:

cudaGetDeviceCount, cudaGetDevice, cudaGetDeviceProperties, cudaSetDevice, cudaSetValidDevices, cudaChooseDevice

5.1.2.23 cudaError_t cudaSetValidDevices (int * *device_arr*, int *len*)

Sets a list of devices for CUDA execution in priority order using `device_arr`. The parameter `len` specifies the number of elements in the list. CUDA will try devices from the list sequentially until it finds one that works. If this function is not called, or if it is called with a `len` of 0, then CUDA will go back to its default behavior of trying devices sequentially from a default list containing all of the available CUDA devices in the system. If a specified device ID in the list does not exist, this function will return cudaErrorInvalidDevice. If `len` is not 0 and `device_arr` is NULL or if `len` exceeds the number of devices in the system, then cudaErrorInvalidValue is returned.

Parameters:

device_arr - List of devices to try

len - Number of devices in specified list

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGetDeviceCount, cudaSetDevice, cudaGetDeviceProperties, cudaSetDeviceFlags, cudaChooseDevice

5.2 Thread Management [DEPRECATED]

Functions

- cudaError_t cudaThreadExit (void)

  Exit and clean up from CUDA launches.

- cudaError_t cudaThreadGetCacheConfig (enum cudaFuncCache *pCacheConfig)

  Returns the preferred cache configuration for the current device.

- cudaError_t cudaThreadGetLimit (size_t *pValue, enum cudaLimit limit)

  Returns resource limits.

- cudaError_t cudaThreadSetCacheConfig (enum cudaFuncCache cacheConfig)

  Sets the preferred cache configuration for the current device.

- cudaError_t cudaThreadSetLimit (enum cudaLimit limit, size_t value)

  Set resource limits.

- cudaError_t cudaThreadSynchronize (void)

  Wait for compute device to finish.

5.2.1 Detailed Description

This section describes deprecated thread management functions of the CUDA runtime application programming interface.

5.2.2 Function Documentation

5.2.2.1 cudaError_t cudaThreadExit (void)

Deprecated

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is identical to the non-deprecated function cudaDeviceReset(), which should be used instead.

Explicitly destroys all cleans up all resources associated with the current device in the current process. Any subsequent API call to this device will reinitialize the device.

Note that this function will reset the device immediately. It is the caller's responsibility to ensure that the device is not being accessed by any other host threads from the process when this function is called.

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceReset

5.2.2.2 cudaError_t cudaThreadGetCacheConfig (enum cudaFuncCache * *pCacheConfig*)

Deprecated

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is identical to the non-deprecated function cudaDeviceGetCacheConfig(), which should be used instead.

On devices where the L1 cache and shared memory use the same hardware resources, this returns through `pCacheConfig` the preferred cache configuration for the current device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute functions.

This will return a `pCacheConfig` of cudaFuncCachePreferNone on devices where the size of the L1 cache and shared memory are fixed.

The supported cache configurations are:

- cudaFuncCachePreferNone: no preference for shared memory or L1 (default)
- cudaFuncCachePreferShared: prefer larger shared memory and smaller L1 cache
- cudaFuncCachePreferL1: prefer larger L1 cache and smaller shared memory

Parameters:

pCacheConfig - Returned cache configuration

Returns:

cudaSuccess, cudaErrorInitializationError

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceGetCacheConfig

5.2.2.3 cudaError_t cudaThreadGetLimit (size_t * *pValue*, enum cudaLimit *limit*)

Deprecated

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is identical to the non-deprecated function cudaDeviceGetLimit(), which should be used instead.

Returns in `*pValue` the current size of `limit`. The supported cudaLimit values are:

- cudaLimitStackSize: stack size of each GPU thread;

- cudaLimitPrintfFifoSize: size of the shared FIFO used by the printf() and fprintf() device system calls.
- cudaLimitMallocHeapSize: size of the heap used by the malloc() and free() device system calls;

Parameters:

limit - Limit to query

pValue - Returned size in bytes of limit

Returns:

cudaSuccess, cudaErrorUnsupportedLimit, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceGetLimit

5.2.2.4 cudaError_t cudaThreadSetCacheConfig (enum cudaFuncCache *cacheConfig*)

Deprecated

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is identical to the non-deprecated function cudaDeviceSetCacheConfig(), which should be used instead.

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `cacheConfig` the preferred cache configuration for the current device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute the function. Any function preference set via cudaFuncSetCacheConfig (C API) or cudaFuncSetCacheConfig (C++ API) will be preferred over this device-wide setting. Setting the device-wide cache configuration to cudaFuncCachePreferNone will cause subsequent kernel launches to prefer to not change the cache configuration unless required to launch the kernel.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- cudaFuncCachePreferNone: no preference for shared memory or L1 (default)
- cudaFuncCachePreferShared: prefer larger shared memory and smaller L1 cache
- cudaFuncCachePreferL1: prefer larger L1 cache and smaller shared memory

Parameters:

cacheConfig - Requested cache configuration

Returns:

cudaSuccess, cudaErrorInitializationError

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceSetCacheConfig

5.2.2.5 cudaError_t cudaThreadSetLimit (enum cudaLimit *limit*, size_t *value*)

Deprecated

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is identical to the non-deprecated function cudaDeviceSetLimit(), which should be used instead.

Setting `limit` to `value` is a request by the application to update the current limit maintained by the device. The driver is free to modify the requested value to meet h/w requirements (this could be clamping to minimum or maximum values, rounding up to nearest element size, etc). The application can use cudaThreadGetLimit() to find out exactly what the limit has been set to.

Setting each cudaLimit has its own specific restrictions, so each is discussed here.

- cudaLimitStackSize controls the stack size of each GPU thread. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error cudaErrorUnsupportedLimit being returned.

- cudaLimitPrintfFifoSize controls the size of the shared FIFO used by the printf() and fprintf() device system calls. Setting cudaLimitPrintfFifoSize must be performed before launching any kernel that uses the printf() or fprintf() device system calls, otherwise cudaErrorInvalidValue will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error cudaErrorUnsupportedLimit being returned.

- cudaLimitMallocHeapSize controls the size of the heap used by the malloc() and free() device system calls. Setting cudaLimitMallocHeapSize must be performed before launching any kernel that uses the malloc() or free() device system calls, otherwise cudaErrorInvalidValue will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error cudaErrorUnsupportedLimit being returned.

Parameters:

limit - Limit to set

value - Size in bytes of limit

Returns:

cudaSuccess, cudaErrorUnsupportedLimit, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceSetLimit

5.2.2.6 cudaError_t cudaThreadSynchronize (void)

Deprecated

Note that this function is deprecated because its name does not reflect its behavior. Its functionality is similar to the non-deprecated function cudaDeviceSynchronize(), which should be used instead.

Blocks until the device has completed all preceding requested tasks. cudaThreadSynchronize() returns an error if one of the preceding tasks has failed. If the cudaDeviceScheduleBlockingSync flag was set for this device, the host thread will block until the device has finished its work.

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceSynchronize

5.3 Error Handling

Functions

- __cudart_builtin__ const char * cudaGetErrorString (cudaError_t error)

  Returns the message string from an error code.

- __cudart_builtin__ cudaError_t cudaGetLastError (void)

  Returns the last error from a runtime call.

- __cudart_builtin__ cudaError_t cudaPeekAtLastError (void)

  Returns the last error from a runtime call.

5.3.1 Detailed Description

This section describes the error handling functions of the CUDA runtime application programming interface.

5.3.2 Function Documentation

5.3.2.1 __cudart_builtin__ const char* cudaGetErrorString (cudaError_t *error*)

Returns the message string from an error code.

Parameters:

error - Error code to convert to string

Returns:

`char*` pointer to a NULL-terminated string

See also:

cudaGetLastError, cudaPeekAtLastError, cudaError

5.3.2.2 __cudart_builtin__ cudaError_t cudaGetLastError (void)

Returns the last error that has been produced by any of the runtime calls in the same host thread and resets it to cudaSuccess.

Returns:

cudaSuccess, cudaErrorMissingConfiguration, cudaErrorMemoryAllocation, cudaErrorInitializationError, cudaErrorLaunchFailure, cudaErrorLaunchTimeout, cudaErrorLaunchOutOfResources, cudaErrorInvalidDeviceFunction, cudaErrorInvalidConfiguration, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidSymbol, cudaErrorUnmapBufferObjectFailed, cudaErrorInvalidHostPointer, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture, cudaErrorInvalidTextureBinding, cudaErrorInvalidChannelDescriptor, cudaErrorInvalidMemcpyDirection, cudaErrorInvalidFilterSetting, cudaErrorInvalidNormSetting, cudaErrorUnknown, cudaErrorInvalidResourceHandle, cudaErrorInsufficientDriver, cudaErrorSetOnActiveProcess, cudaErrorStartupFailure,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaPeekAtLastError, cudaGetErrorString, cudaError

5.3.2.3 __cudart_builtin__ cudaError_t cudaPeekAtLastError (void)

Returns the last error that has been produced by any of the runtime calls in the same host thread. Note that this call does not reset the error to cudaSuccess like cudaGetLastError().

Returns:

cudaSuccess, cudaErrorMissingConfiguration, cudaErrorMemoryAllocation, cudaErrorInitializationError, cudaErrorLaunchFailure, cudaErrorLaunchTimeout, cudaErrorLaunchOutOfResources, cudaErrorInvalidDeviceFunction, cudaErrorInvalidConfiguration, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidSymbol, cudaErrorUnmapBufferObjectFailed, cudaErrorInvalidHostPointer, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture, cudaErrorInvalidTextureBinding, cudaErrorInvalidChannelDescriptor, cudaErrorInvalidMemcpyDirection, cudaErrorInvalidFilterSetting, cudaErrorInvalidNormSetting, cudaErrorUnknown, cudaErrorInvalidResourceHandle, cudaErrorInsufficientDriver, cudaErrorSetOnActiveProcess, cudaErrorStartupFailure,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGetLastError, cudaGetErrorString, cudaError

5.4 Stream Management

Typedefs

- typedef void(CUDART_CB ∗ cudaStreamCallback_t)(cudaStream_t stream, cudaError_t status, void ∗userData)

Functions

- cudaError_t cudaStreamAddCallback (cudaStream_t stream, cudaStreamCallback_t callback, void ∗userData, unsigned int flags)

  Add a callback to a compute stream.

- cudaError_t cudaStreamCreate (cudaStream_t ∗pStream)

  Create an asynchronous stream.

- __cudart_builtin__ cudaError_t cudaStreamCreateWithFlags (cudaStream_t ∗pStream, unsigned int flags)

  Create an asynchronous stream.

- __cudart_builtin__ cudaError_t cudaStreamDestroy (cudaStream_t stream)

  Destroys and cleans up an asynchronous stream.

- cudaError_t cudaStreamQuery (cudaStream_t stream)

  Queries an asynchronous stream for completion status.

- cudaError_t cudaStreamSynchronize (cudaStream_t stream)

  Waits for stream tasks to complete.

- __cudart_builtin__ cudaError_t cudaStreamWaitEvent (cudaStream_t stream, cudaEvent_t event, unsigned int flags)

  Make a compute stream wait on an event.

5.4.1 Detailed Description

This section describes the stream management functions of the CUDA runtime application programming interface.

5.4.2 Typedef Documentation

5.4.2.1 typedef void(CUDART_CB ∗ cudaStreamCallback_t)(cudaStream_t stream, cudaError_t status, void ∗userData)

Type of stream callback functions.

Parameters:

stream The stream as passed to cudaStreamAddCallback, may be NULL.

status cudaSuccess or any persistent error on the stream.

userData User parameter provided at registration.

5.4.3 Function Documentation

5.4.3.1 cudaError_t cudaStreamAddCallback (cudaStream_t *stream*, cudaStreamCallback_t *callback*, void * *userData*, unsigned int *flags*)

Adds a callback to be called on the host after all currently enqueued items in the stream have completed. For each cudaStreamAddCallback call, a callback will be executed exactly once. The callback will block later work in the stream until it is finished.

The callback may be passed cudaSuccess or an error code. In the event of a device error, all subsequently executed callbacks will receive an appropriate cudaError_t.

Callbacks must not make any CUDA API calls. Attempting to use CUDA APIs will result in cudaErrorNotPermitted. Callbacks must not perform any synchronization that may depend on outstanding device work or other callbacks that are not mandated to run earlier. Callbacks without a mandated order (in independent streams) execute in undefined order and may be serialized.

This API requires compute capability 1.1 or greater. See cudaDeviceGetAttribute or cudaGetDeviceProperties to query compute capability. Calling this API with an earlier compute version will return cudaErrorNotSupported.

Parameters:

stream - Stream to add callback to

callback - The function to call once preceding stream operations are complete

userData - User specified data to be passed to the callback function

flags - Reserved for future use, must be 0

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorNotSupported

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaStreamCreate, cudaStreamCreateWithFlags, cudaStreamQuery, cudaStreamSynchronize, cudaStreamWaitEvent, cudaStreamDestroy

5.4.3.2 cudaError_t cudaStreamCreate (cudaStream_t * *pStream*)

Creates a new asynchronous stream.

Parameters:

pStream - Pointer to new stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaStreamCreate, cudaStreamCreateWithFlags, cudaStreamQuery, cudaStreamSynchronize, cudaStreamWaitEvent, cudaStreamAddCallback, cudaStreamDestroy

5.4.3.3 __cudart_builtin__ cudaError_t cudaStreamCreateWithFlags (cudaStream_t * *pStream*, unsigned int *flags*)

Creates a new asynchronous stream. The `flags` argument determines the behaviors of the stream. Valid values for `flags` are

- cudaStreamDefault: Default stream creation flag.
- cudaStreamNonBlocking: Specifies that work running in the created stream may run concurrently with work in stream 0 (the NULL stream), and that the created stream should perform no implicit synchronization with stream 0.

Parameters:

pStream - Pointer to new stream identifier

flags - Parameters for stream creation

Returns:

cudaSuccess, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaStreamCreate, cudaStreamQuery, cudaStreamSynchronize, cudaStreamWaitEvent, cudaStreamAddCallback, cudaStreamDestroy

5.4.3.4 __cudart_builtin__ cudaError_t cudaStreamDestroy (cudaStream_t *stream*)

Destroys and cleans up the asynchronous stream specified by `stream`.

In case the device is still doing work in the stream `stream` when cudaStreamDestroy() is called, the function will return immediately and the resources associated with `stream` will be released automatically once the device has completed all work in `stream`.

Parameters:

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaStreamCreate, cudaStreamCreateWithFlags, cudaStreamQuery, cudaStreamWaitEvent, cudaStreamSynchronize, cudaStreamAddCallback

5.4.3.5 cudaError_t cudaStreamQuery (cudaStream_t *stream*)

Returns cudaSuccess if all operations in `stream` have completed, or cudaErrorNotReady if not.

Parameters:

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorNotReady, cudaErrorInvalidResourceHandle

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaStreamCreate, cudaStreamCreateWithFlags, cudaStreamWaitEvent, cudaStreamSynchronize, cudaStreamAddCallback, cudaStreamDestroy

5.4.3.6 cudaError_t cudaStreamSynchronize (cudaStream_t *stream*)

Blocks until `stream` has completed all operations. If the cudaDeviceScheduleBlockingSync flag was set for this device, the host thread will block until the stream is finished with all of its tasks.

Parameters:

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaStreamCreate, cudaStreamCreateWithFlags, cudaStreamQuery, cudaStreamWaitEvent, cudaStreamAddCallback, cudaStreamDestroy

5.4.3.7 __cudart_builtin__ cudaError_t cudaStreamWaitEvent (cudaStream_t *stream*, cudaEvent_t *event*, unsigned int *flags*)

Makes all future work submitted to `stream` wait until `event` reports completion before beginning execution. This synchronization will be performed efficiently on the device. The event `event` may be from a different context than `stream`, in which case this function will perform cross-device synchronization.

The stream `stream` will wait only for the completion of the most recent host call to cudaEventRecord() on `event`. Once this call has returned, any functions (including cudaEventRecord() and cudaEventDestroy()) may be called on `event` again, and the subsequent calls will not have any effect on `stream`.

If `stream` is NULL, any future work submitted in any stream will wait for `event` to complete before beginning execution. This effectively creates a barrier for all future work submitted to the device on this thread.

If cudaEventRecord() has not been called on `event`, this call acts as if the record has already completed, and so is a functional no-op.

Parameters:

stream - Stream to wait

event - Event to wait on

flags - Parameters for the operation (must be 0)

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaStreamCreate, cudaStreamCreateWithFlags, cudaStreamQuery, cudaStreamSynchronize, cudaStreamAddCallback, cudaStreamDestroy

5.5 Event Management

Functions

- cudaError_t cudaEventCreate (cudaEvent_t ∗event)
  Creates an event object.
- __cudart_builtin__ cudaError_t cudaEventCreateWithFlags (cudaEvent_t ∗event, unsigned int flags)
  Creates an event object with the specified flags.
- __cudart_builtin__ cudaError_t cudaEventDestroy (cudaEvent_t event)
  Destroys an event object.
- cudaError_t cudaEventElapsedTime (float ∗ms, cudaEvent_t start, cudaEvent_t end)
  Computes the elapsed time between events.
- cudaError_t cudaEventQuery (cudaEvent_t event)
  Queries an event's status.
- __cudart_builtin__ cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream=0)
  Records an event.
- cudaError_t cudaEventSynchronize (cudaEvent_t event)
  Waits for an event to complete.

5.5.1 Detailed Description

This section describes the event management functions of the CUDA runtime application programming interface.

5.5.2 Function Documentation

5.5.2.1 cudaError_t cudaEventCreate (cudaEvent_t ∗ *event*)

Creates an event object using cudaEventDefault.

Parameters:

event - Newly created event

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorLaunchFailure, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaEventCreate (C++ API), cudaEventCreateWithFlags, cudaEventRecord, cudaEventQuery, cudaEventSynchronize, cudaEventDestroy, cudaEventElapsedTime, cudaStreamWaitEvent

5.5.2.2 __cudart_builtin__ cudaError_t cudaEventCreateWithFlags (cudaEvent_t * *event*, unsigned int *flags*)

Creates an event object with the specified flags. Valid flags include:

- cudaEventDefault: Default event creation flag.
- cudaEventBlockingSync: Specifies that event should use blocking synchronization. A host thread that uses cudaEventSynchronize() to wait on an event created with this flag will block until the event actually completes.
- cudaEventDisableTiming: Specifies that the created event does not need to record timing data. Events created with this flag specified and the cudaEventBlockingSync flag not specified will provide the best performance when used with cudaStreamWaitEvent() and cudaEventQuery().
- cudaEventInterprocess: Specifies that the created event may be used as an interprocess event by cudaIpcGetEventHandle(). cudaEventInterprocess must be specified along with cudaEventDisableTiming.

Parameters:

event - Newly created event

flags - Flags for new event

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorLaunchFailure, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaEventCreate (C API), cudaEventSynchronize, cudaEventDestroy, cudaEventElapsedTime, cudaStreamWaitEvent

5.5.2.3 __cudart_builtin__ cudaError_t cudaEventDestroy (cudaEvent_t *event*)

Destroys the event specified by `event`.

In case `event` has been recorded but has not yet been completed when cudaEventDestroy() is called, the function will return immediately and the resources associated with `event` will be released automatically once the device has completed `event`.

Parameters:

event - Event to destroy

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorLaunchFailure

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaEventCreate (C API), cudaEventCreateWithFlags, cudaEventQuery, cudaEventSynchronize, cudaEventRecord, cudaEventElapsedTime

5.5.2.4 cudaError_t cudaEventElapsedTime (float * *ms*, cudaEvent_t *start*, cudaEvent_t *end*)

Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds).

If either event was last recorded in a non-NULL stream, the resulting time may be greater than expected (even if both used the same stream handle). This happens because the cudaEventRecord() operation takes place asynchronously and there is no guarantee that the measured latency is actually just between the two events. Any number of other different stream operations could execute in between the two measured events, thus altering the timing in a significant way.

If cudaEventRecord() has not been called on either event, then cudaErrorInvalidResourceHandle is returned. If cudaEventRecord() has been called on both events but one or both of them has not yet been completed (that is, cudaEventQuery() would return cudaErrorNotReady on at least one of the events), cudaErrorNotReady is returned. If either event was created with the cudaEventDisableTiming flag, then this function will return cudaErrorInvalidResourceHandle.

Parameters:

ms - Time between `start` and `end` in ms

start - Starting event

end - Ending event

Returns:

cudaSuccess, cudaErrorNotReady, cudaErrorInvalidValue, cudaErrorInitializationError, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaEventCreate (C API), cudaEventCreateWithFlags, cudaEventQuery, cudaEventSynchronize, cudaEventDestroy, cudaEventRecord

5.5.2.5 cudaError_t cudaEventQuery (cudaEvent_t *event*)

Query the status of all device work preceding the most recent call to cudaEventRecord() (in the appropriate compute streams, as specified by the arguments to cudaEventRecord()).

If this work has successfully been completed by the device, or if cudaEventRecord() has not been called on `event`, then cudaSuccess is returned. If this work has not yet been completed by the device then cudaErrorNotReady is returned.

Parameters:

event - Event to query

Returns:

cudaSuccess, cudaErrorNotReady, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaEventCreate (C API), cudaEventCreateWithFlags, cudaEventRecord, cudaEventSynchronize, cudaEventDestroy, cudaEventElapsedTime

5.5.2.6 __cudart_builtin__ cudaError_t cudaEventRecord (cudaEvent_t *event*, cudaStream_t *stream* = 0)

Records an event. If `stream` is non-zero, the event is recorded after all preceding operations in `stream` have been completed; otherwise, it is recorded after all preceding operations in the CUDA context have been completed. Since operation is asynchronous, cudaEventQuery() and/or cudaEventSynchronize() must be used to determine when the event has actually been recorded.

If cudaEventRecord() has previously been called on `event`, then this call will overwrite any existing state in `event`. Any subsequent calls which examine the status of `event` will only examine the completion of this most recent call to cudaEventRecord().

Parameters:

event - Event to record

stream - Stream in which to record event

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInitializationError, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaEventCreate (C API), cudaEventCreateWithFlags, cudaEventQuery, cudaEventSynchronize, cudaEventDestroy, cudaEventElapsedTime, cudaStreamWaitEvent

5.5.2.7 cudaError_t cudaEventSynchronize (cudaEvent_t *event*)

Wait until the completion of all device work preceding the most recent call to cudaEventRecord() (in the appropriate compute streams, as specified by the arguments to cudaEventRecord()).

If cudaEventRecord() has not been called on `event`, cudaSuccess is returned immediately.

Waiting for an event that was created with the cudaEventBlockingSync flag will cause the calling CPU thread to block until the event has been completed by the device. If the cudaEventBlockingSync flag has not been set, then the CPU thread will busy-wait until the event has been completed by the device.

Parameters:

event - Event to wait for

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaEventCreate (C API), cudaEventCreateWithFlags, cudaEventRecord, cudaEventQuery, cudaEventDestroy, cudaEventElapsedTime

5.6 Execution Control

Functions

- cudaError_t cudaConfigureCall (dim3 gridDim, dim3 blockDim, size_t sharedMem=0, cudaStream_t stream=0)

  Configure a device-launch.

- __cudart_builtin__ cudaError_t cudaFuncGetAttributes (struct cudaFuncAttributes *attr, const void *func)

  Find out attributes for a given function.

- cudaError_t cudaFuncSetCacheConfig (const void *func, enum cudaFuncCache cacheConfig)

  Sets the preferred cache configuration for a device function.

- cudaError_t cudaFuncSetSharedMemConfig (const void *func, enum cudaSharedMemConfig config)

  Sets the shared memory configuration for a device function.

- cudaError_t cudaLaunch (const void *func)

  Launches a device function.

- cudaError_t cudaSetDoubleForDevice (double *d)

  Converts a double argument to be executed on a device.

- cudaError_t cudaSetDoubleForHost (double *d)

  Converts a double argument after execution on a device.

- cudaError_t cudaSetupArgument (const void *arg, size_t size, size_t offset)

  Configure a device launch.

5.6.1 Detailed Description

This section describes the execution control functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module.

5.6.2 Function Documentation

5.6.2.1 cudaError_t cudaConfigureCall (dim3 *gridDim*, dim3 *blockDim*, size_t *sharedMem* = 0, cudaStream_t *stream* = 0)

Specifies the grid and block dimensions for the device call to be executed similar to the execution configuration syntax. cudaConfigureCall() is stack based. Each call pushes data on top of an execution stack. This data contains the dimension for the grid and thread blocks, together with any arguments for the call.

Parameters:

gridDim - Grid dimensions

blockDim - Block dimensions

sharedMem - Shared memory

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidConfiguration

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaFuncSetCacheConfig (C API), cudaFuncGetAttributes (C API), cudaLaunch (C API), cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaSetupArgument (C API),

5.6.2.2 __cudart_builtin__ cudaError_t cudaFuncGetAttributes (struct cudaFuncAttributes * *attr*, const void * *func*)

This function obtains the attributes of a function specified via `func`. `func` is a device function symbol and must be declared as a `__global__` function. The fetched attributes are placed in `attr`. If the specified function does not exist, then cudaErrorInvalidDeviceFunction is returned.

Note that some function attributes such as maxThreadsPerBlock may vary based on the device that is currently being used.

Parameters:

attr - Return pointer to function's attributes

func - Device function symbol

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidDeviceFunction

Note:

Note that this function may also return error codes from previous, asynchronous launches.
Use of a string naming a function as the `func` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaConfigureCall, cudaFuncSetCacheConfig (C API), cudaFuncGetAttributes (C++ API), cudaLaunch (C API), cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaSetupArgument (C API)

5.6.2.3 cudaError_t cudaFuncSetCacheConfig (const void * *func*, enum cudaFuncCache *cacheConfig*)

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `cacheConfig` the preferred cache configuration for the function specified via `func`. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute `func`.

`func` is a device function symbol and must be declared as a `__global__` function. If the specified function does not exist, then cudaErrorInvalidDeviceFunction is returned.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- cudaFuncCachePreferNone: no preference for shared memory or L1 (default)
- cudaFuncCachePreferShared: prefer larger shared memory and smaller L1 cache
- cudaFuncCachePreferL1: prefer larger L1 cache and smaller shared memory

Parameters:

func - Device function symbol

cacheConfig - Requested cache configuration

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidDeviceFunction

Note:

Note that this function may also return error codes from previous, asynchronous launches.
Use of a string naming a function as the `func` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaConfigureCall, cudaFuncSetCacheConfig (C++ API), cudaFuncGetAttributes (C API), cudaLaunch (C API), cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaSetupArgument (C API), cudaThreadGetCacheConfig, cudaThreadSetCacheConfig

5.6.2.4 cudaError_t cudaFuncSetSharedMemConfig (const void * *func*, enum cudaSharedMemConfig *config*)

On devices with configurable shared memory banks, this function will force all subsequent launches of the specified device function to have the given shared memory bank size configuration. On any given launch of the function, the shared memory configuration of the device will be temporarily changed if needed to suit the function's preferred configuration. Changes in shared memory configuration between subsequent launches of functions, may introduce a device side synchronization point.

Any per-function setting of shared memory bank size set via cudaFuncSetSharedMemConfig will override the device wide setting set by cudaDeviceSetSharedMemConfig.

Changing the shared memory bank size will not increase shared memory usage or affect occupancy of kernels, but may have major effects on performance. Larger bank sizes will allow for greater potential bandwidth to shared memory, but will change what kinds of accesses to shared memory will result in bank conflicts.

This function will do nothing on devices with fixed shared memory bank size.

The supported bank configurations are:

- cudaSharedMemBankSizeDefault: use the device's shared memory configuration when launching this function.
- cudaSharedMemBankSizeFourByte: set shared memory bank width to be four bytes natively when launching this function.
- cudaSharedMemBankSizeEightByte: set shared memory bank width to be eight bytes natively when launching this function.

Parameters:

func - Device function symbol

config - Requested shared memory configuration

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidDeviceFunction, cudaErrorInvalidValue,

Note:

Note that this function may also return error codes from previous, asynchronous launches.
Use of a string naming a function as the `func` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaConfigureCall, cudaDeviceSetSharedMemConfig, cudaDeviceGetSharedMemConfig, cudaDeviceSetCacheConfig, cudaDeviceGetCacheConfig, cudaFuncSetCacheConfig

5.6.2.5 cudaError_t cudaLaunch (const void * *func*)

Launches the function `func` on the device. The parameter `func` must be a device function symbol. The parameter specified by `func` must be declared as a `__global__` function. cudaLaunch() must be preceded by a call to cudaConfigureCall() since it pops the data that was pushed by cudaConfigureCall() from the execution stack.

Parameters:

func - Device function symbol

Returns:

cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidConfiguration, cudaErrorLaunchFailure, cudaErrorLaunchTimeout, cudaErrorLaunchOutOfResources, cudaErrorSharedObjectInitFailed

Note:

Note that this function may also return error codes from previous, asynchronous launches.
Use of a string naming a variable as the `symbol` paramater was removed in CUDA 5.0.

See also:

cudaConfigureCall, cudaFuncSetCacheConfig (C API), cudaFuncGetAttributes (C API), cudaLaunch (C++ API), cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaSetupArgument (C API), cudaThreadGetCacheConfig, cudaThreadSetCacheConfig

5.6.2.6 cudaError_t cudaSetDoubleForDevice (double * *d*)

Parameters:

d - Double to convert

Converts the double value of `d` to an internal float representation if the device does not support double arithmetic. If the device does natively support doubles, then this function does nothing.

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaConfigureCall, cudaFuncSetCacheConfig (C API), cudaFuncGetAttributes (C API), cudaLaunch (C API), cudaSetDoubleForHost, cudaSetupArgument (C API)

5.6.2.7 cudaError_t cudaSetDoubleForHost (double * *d*)

Converts the double value of `d` from a potentially internal float representation if the device does not support double arithmetic. If the device does natively support doubles, then this function does nothing.

Parameters:

d - Double to convert

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaConfigureCall, cudaFuncSetCacheConfig (C API), cudaFuncGetAttributes (C API), cudaLaunch (C API), cudaSetDoubleForDevice, cudaSetupArgument (C API)

5.6.2.8 cudaError_t cudaSetupArgument (const void * *arg*, size_t *size*, size_t *offset*)

Pushes `size` bytes of the argument pointed to by `arg` at `offset` bytes from the start of the parameter passing area, which starts at offset 0. The arguments are stored in the top of the execution stack. cudaSetupArgument() must be preceded by a call to cudaConfigureCall().

Parameters:

arg - Argument to push for a kernel launch

size - Size of argument

offset - Offset in argument stack to push new arg

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaConfigureCall, cudaFuncSetCacheConfig (C API), cudaFuncGetAttributes (C API), cudaLaunch (C API), cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaSetupArgument (C++ API),

5.7 Memory Management

Functions

- cudaError_t cudaArrayGetInfo (struct cudaChannelFormatDesc *desc, struct cudaExtent *extent, unsigned int *flags, cudaArray_t array)

  Gets info about the specified cudaArray.

- __cudart_builtin__ cudaError_t cudaFree (void *devPtr)

  Frees memory on the device.

- cudaError_t cudaFreeArray (cudaArray_t array)

  Frees an array on the device.

- cudaError_t cudaFreeHost (void *ptr)

  Frees page-locked memory.

- cudaError_t cudaFreeMipmappedArray (cudaMipmappedArray_t mipmappedArray)

  Frees a mipmapped array on the device.

- cudaError_t cudaGetMipmappedArrayLevel (cudaArray_t *levelArray, cudaMipmappedArray_const_t mipmappedArray, unsigned int level)

  Gets a mipmap level of a CUDA mipmapped array.

- cudaError_t cudaGetSymbolAddress (void **devPtr, const void *symbol)

  Finds the address associated with a CUDA symbol.

- cudaError_t cudaGetSymbolSize (size_t *size, const void *symbol)

  Finds the size of the object associated with a CUDA symbol.

- cudaError_t cudaHostAlloc (void **pHost, size_t size, unsigned int flags)

  Allocates page-locked memory on the host.

- cudaError_t cudaHostGetDevicePointer (void **pDevice, void *pHost, unsigned int flags)

  Passes back device pointer of mapped host memory allocated by cudaHostAlloc or registered by cudaHostRegister.

- cudaError_t cudaHostGetFlags (unsigned int *pFlags, void *pHost)

  Passes back flags used to allocate pinned host memory allocated by cudaHostAlloc.

- cudaError_t cudaHostRegister (void *ptr, size_t size, unsigned int flags)

  Registers an existing host memory range for use by CUDA.

- cudaError_t cudaHostUnregister (void *ptr)

  Unregisters a memory range that was registered with cudaHostRegister.

- __cudart_builtin__ cudaError_t cudaMalloc (void **devPtr, size_t size)

  Allocate memory on the device.

- cudaError_t cudaMalloc3D (struct cudaPitchedPtr *pitchedDevPtr, struct cudaExtent extent)

  Allocates logical 1D, 2D, or 3D memory objects on the device.

- cudaError_t cudaMalloc3DArray (cudaArray_t *array, const struct cudaChannelFormatDesc *desc, struct cudaExtent extent, unsigned int flags=0)

  Allocate an array on the device.

- cudaError_t cudaMallocArray (cudaArray_t *array, const struct cudaChannelFormatDesc *desc, size_t width, size_t height=0, unsigned int flags=0)

  Allocate an array on the device.

- cudaError_t cudaMallocHost (void **ptr, size_t size)

  Allocates page-locked memory on the host.

- cudaError_t cudaMallocMipmappedArray (cudaMipmappedArray_t *mipmappedArray, const struct cudaChannelFormatDesc *desc, struct cudaExtent extent, unsigned int numLevels, unsigned int flags=0)

  Allocate a mipmapped array on the device.

- cudaError_t cudaMallocPitch (void **devPtr, size_t *pitch, size_t width, size_t height)

  Allocates pitched memory on the device.

- cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)

  Copies data between host and device.

- cudaError_t cudaMemcpy2D (void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)

  Copies data between host and device.

- cudaError_t cudaMemcpy2DArrayToArray (cudaArray_t dst, size_t wOffsetDst, size_t hOffsetDst, cudaArray_const_t src, size_t wOffsetSrc, size_t hOffsetSrc, size_t width, size_t height, enum cudaMemcpyKind kind=cudaMemcpyDeviceToDevice)

  Copies data between host and device.

- __cudart_builtin__ cudaError_t cudaMemcpy2DAsync (void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream=0)

  Copies data between host and device.

- cudaError_t cudaMemcpy2DFromArray (void *dst, size_t dpitch, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t width, size_t height, enum cudaMemcpyKind kind)

  Copies data between host and device.

- cudaError_t cudaMemcpy2DFromArrayAsync (void *dst, size_t dpitch, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream=0)

  Copies data between host and device.

- cudaError_t cudaMemcpy2DToArray (cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)

  Copies data between host and device.

- cudaError_t cudaMemcpy2DToArrayAsync (cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream=0)

  Copies data between host and device.

- cudaError_t cudaMemcpy3D (const struct cudaMemcpy3DParms *p)

  Copies data between 3D objects.

- __cudart_builtin__ cudaError_t cudaMemcpy3DAsync (const struct cudaMemcpy3DParms *p, cudaStream_t stream=0)

  Copies data between 3D objects.

- cudaError_t cudaMemcpy3DPeer (const struct cudaMemcpy3DPeerParms *p)

  Copies memory between devices.

- cudaError_t cudaMemcpy3DPeerAsync (const struct cudaMemcpy3DPeerParms *p, cudaStream_t stream=0)

  Copies memory between devices asynchronously.

- cudaError_t cudaMemcpyArrayToArray (cudaArray_t dst, size_t wOffsetDst, size_t hOffsetDst, cudaArray_const_t src, size_t wOffsetSrc, size_t hOffsetSrc, size_t count, enum cudaMemcpyKind kind=cudaMemcpyDeviceToDevice)

  Copies data between host and device.

- __cudart_builtin__ cudaError_t cudaMemcpyAsync (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream=0)

  Copies data between host and device.

- cudaError_t cudaMemcpyFromArray (void *dst, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t count, enum cudaMemcpyKind kind)

  Copies data between host and device.

- cudaError_t cudaMemcpyFromArrayAsync (void *dst, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream=0)

  Copies data between host and device.

- cudaError_t cudaMemcpyFromSymbol (void *dst, const void *symbol, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyDeviceToHost)

  Copies data from the given symbol on the device.

- cudaError_t cudaMemcpyFromSymbolAsync (void *dst, const void *symbol, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream_t stream=0)

  Copies data from the given symbol on the device.

- cudaError_t cudaMemcpyPeer (void *dst, int dstDevice, const void *src, int srcDevice, size_t count)

  Copies memory between two devices.

- cudaError_t cudaMemcpyPeerAsync (void *dst, int dstDevice, const void *src, int srcDevice, size_t count, cudaStream_t stream=0)

  Copies memory between two devices asynchronously.

- cudaError_t cudaMemcpyToArray (cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t count, enum cudaMemcpyKind kind)

  Copies data between host and device.

- cudaError_t cudaMemcpyToArrayAsync (cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream=0)

Copies data between host and device.

- cudaError_t cudaMemcpyToSymbol (const void ∗symbol, const void ∗src, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyHostToDevice)

  Copies data to the given symbol on the device.

- cudaError_t cudaMemcpyToSymbolAsync (const void ∗symbol, const void ∗src, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream_t stream=0)

  Copies data to the given symbol on the device.

- cudaError_t cudaMemGetInfo (size_t ∗free, size_t ∗total)

  Gets free and total device memory.

- cudaError_t cudaMemset (void ∗devPtr, int value, size_t count)

  Initializes or sets device memory to a value.

- cudaError_t cudaMemset2D (void ∗devPtr, size_t pitch, int value, size_t width, size_t height)

  Initializes or sets device memory to a value.

- __cudart_builtin__ cudaError_t cudaMemset2DAsync (void ∗devPtr, size_t pitch, int value, size_t width, size_t height, cudaStream_t stream=0)

  Initializes or sets device memory to a value.

- cudaError_t cudaMemset3D (struct cudaPitchedPtr pitchedDevPtr, int value, struct cudaExtent extent)

  Initializes or sets device memory to a value.

- __cudart_builtin__ cudaError_t cudaMemset3DAsync (struct cudaPitchedPtr pitchedDevPtr, int value, struct cudaExtent extent, cudaStream_t stream=0)

  Initializes or sets device memory to a value.

- __cudart_builtin__ cudaError_t cudaMemsetAsync (void ∗devPtr, int value, size_t count, cudaStream_t stream=0)

  Initializes or sets device memory to a value.

- struct cudaExtent make_cudaExtent (size_t w, size_t h, size_t d)

  Returns a cudaExtent based on input parameters.

- struct cudaPitchedPtr make_cudaPitchedPtr (void ∗d, size_t p, size_t xsz, size_t ysz)

  Returns a cudaPitchedPtr based on input parameters.

- struct cudaPos make_cudaPos (size_t x, size_t y, size_t z)

  Returns a cudaPos based on input parameters.

5.7.1 Detailed Description

This section describes the memory management functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module.

5.7.2 Function Documentation

5.7.2.1 cudaError_t cudaArrayGetInfo (struct cudaChannelFormatDesc * *desc*, struct cudaExtent * *extent*, unsigned int * *flags*, cudaArray_t *array*)

Returns in `*desc`, `*extent` and `*flags` respectively, the type, shape and flags of `array`.

Any of `*desc`, `*extent` and `*flags` may be specified as NULL.

Parameters:

desc - Returned array type

extent - Returned array shape. 2D arrays will have depth of zero

flags - Returned array flags

array - The cudaArray to get info for

Returns:

cudaSuccess, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

5.7.2.2 __cudart_builtin__ cudaError_t cudaFree (void * *devPtr*)

Frees the memory space pointed to by `devPtr`, which must have been returned by a previous call to cudaMalloc() or cudaMallocPitch(). Otherwise, or if cudaFree(`devPtr`) has already been called before, an error is returned. If `devPtr` is 0, no operation is performed. cudaFree() returns cudaErrorInvalidDevicePointer in case of failure.

Parameters:

devPtr - Device pointer to memory to free

Returns:

cudaSuccess, cudaErrorInvalidDevicePointer, cudaErrorInitializationError

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMalloc, cudaMallocPitch, cudaMallocArray, cudaFreeArray, cudaMallocHost (C API), cudaFreeHost, cudaMalloc3D, cudaMalloc3DArray, cudaHostAlloc

5.7.2.3 cudaError_t cudaFreeArray (cudaArray_t *array*)

Frees the CUDA array `array`, which must have been * returned by a previous call to cudaMallocArray(). If cudaFreeArray(`array`) has already been called before, cudaErrorInvalidValue is returned. If `devPtr` is 0, no operation is performed.

Parameters:

array - Pointer to array to free

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInitializationError

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMalloc, cudaMallocPitch, cudaFree, cudaMallocArray, cudaMallocHost (C API), cudaFreeHost, cudaHostAlloc

5.7.2.4 cudaError_t cudaFreeHost (void * *ptr*)

Frees the memory space pointed to by `hostPtr`, which must have been returned by a previous call to cudaMallocHost() or cudaHostAlloc().

Parameters:

ptr - Pointer to memory to free

Returns:

cudaSuccess, cudaErrorInitializationError

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMalloc, cudaMallocPitch, cudaFree, cudaMallocArray, cudaFreeArray, cudaMallocHost (C API), cudaMalloc3D, cudaMalloc3DArray, cudaHostAlloc

5.7.2.5 cudaError_t cudaFreeMipmappedArray (cudaMipmappedArray_t *mipmappedArray*)

Frees the CUDA mipmapped array `mipmappedArray`, which must have been returned by a previous call to cudaMallocMipmappedArray(). If cudaFreeMipmappedArray(`mipmappedArray`) has already been called before, cudaErrorInvalidValue is returned.

Parameters:

mipmappedArray - Pointer to mipmapped array to free

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInitializationError

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMalloc, cudaMallocPitch, cudaFree, cudaMallocArray, cudaMallocHost (C API), cudaFreeHost, cudaHostAlloc

5.7.2.6 cudaError_t cudaGetMipmappedArrayLevel (cudaArray_t * *levelArray*, cudaMipmappedArray_const_t *mipmappedArray*, unsigned int *level*)

Returns in `*levelArray` a CUDA array that represents a single mipmap level of the CUDA mipmapped array `mipmappedArray`.

If `level` is greater than the maximum number of levels in this mipmapped array, cudaErrorInvalidValue is returned.

Parameters:

levelArray - Returned mipmap level CUDA array

mipmappedArray - CUDA mipmapped array

level - Mipmap level

Returns:

cudaSuccess, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMalloc3D, cudaMalloc, cudaMallocPitch, cudaFree, cudaFreeArray, cudaMallocHost (C API), cudaFreeHost, cudaHostAlloc, make_cudaExtent

5.7.2.7 cudaError_t cudaGetSymbolAddress (void ** *devPtr*, const void * *symbol*)

Returns in `*devPtr` the address of symbol `symbol` on the device. `symbol` is a variable that resides in global or constant memory space. If `symbol` cannot be found, or if `symbol` is not declared in the global or constant memory space, `*devPtr` is unchanged and the error cudaErrorInvalidSymbol is returned.

Parameters:

devPtr - Return device pointer associated with symbol

symbol - Device symbol address

Returns:

cudaSuccess, cudaErrorInvalidSymbol

Note:

Note that this function may also return error codes from previous, asynchronous launches.
Use of a string naming a variable as the `symbol` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaGetSymbolAddress (C++ API), cudaGetSymbolSize (C API)

5.7.2.8 cudaError_t cudaGetSymbolSize (size_t * *size*, const void * *symbol*)

Returns in `*size` the size of symbol `symbol`. `symbol` is a variable that resides in global or constant memory space. If `symbol` cannot be found, or if `symbol` is not declared in global or constant memory space, `*size` is unchanged and the error cudaErrorInvalidSymbol is returned.

Parameters:

size - Size of object associated with symbol

symbol - Device symbol address

Returns:

cudaSuccess, cudaErrorInvalidSymbol

Note:

Note that this function may also return error codes from previous, asynchronous launches.
Use of a string naming a variable as the `symbol` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaGetSymbolAddress (C API), cudaGetSymbolSize (C++ API)

5.7.2.9 cudaError_t cudaHostAlloc (void ** *pHost*, size_t *size*, unsigned int *flags*)

Allocates `size` bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as cudaMemcpy(). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc(). Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- cudaHostAllocDefault: This flag's value is defined to be 0 and causes cudaHostAlloc() to emulate cudaMallocHost().
- cudaHostAllocPortable: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- cudaHostAllocMapped: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling cudaHostGetDevicePointer().
- cudaHostAllocWriteCombined: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host->device transfers.

All of these flags are orthogonal to one another: a developer may allocate memory that is portable, mapped and/or write-combined with no restrictions.

cudaSetDeviceFlags() must have been called with the cudaDeviceMapHost flag in order for the cudaHostAllocMapped flag to have any effect.

The cudaHostAllocMapped flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to cudaHostGetDevicePointer() because the memory may be mapped into other CUDA contexts via the cudaHostAllocPortable flag.

Memory allocated by this function must be freed with cudaFreeHost().

Parameters:

pHost - Device pointer to allocated memory

size - Requested allocation size in bytes

flags - Requested properties of allocated memory

Returns:

cudaSuccess, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaSetDeviceFlags, cudaMallocHost (C API), cudaFreeHost

5.7.2.10 cudaError_t cudaHostGetDevicePointer (void ** *pDevice*, void * *pHost*, unsigned int *flags*)

Passes back the device pointer corresponding to the mapped, pinned host buffer allocated by cudaHostAlloc() or registered by cudaHostRegister().

cudaHostGetDevicePointer() will fail if the cudaDeviceMapHost flag was not specified before deferred context creation occurred, or if called on a device that does not support mapped, pinned memory.

`flags` provides for future releases. For now, it must be set to 0.

Parameters:

pDevice - Returned device pointer for mapped memory

pHost - Requested host pointer mapping

flags - Flags for extensions (must be 0 for now)

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaSetDeviceFlags, cudaHostAlloc

5.7.2.11 cudaError_t cudaHostGetFlags (unsigned int * *pFlags*, void * *pHost*)

cudaHostGetFlags() will fail if the input pointer does not reside in an address range allocated by cudaHostAlloc().

Parameters:

pFlags - Returned flags word

pHost - Host pointer

Returns:

cudaSuccess, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaHostAlloc

5.7.2.12 cudaError_t cudaHostRegister (void * *ptr*, size_t *size*, unsigned int *flags*)

Page-locks the memory range specified by `ptr` and `size` and maps it for the device(s) as specified by `flags`. This memory range also is added to the same tracking mechanism as cudaHostAlloc() to automatically accelerate calls to functions such as cudaMemcpy(). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory that has not been registered. Page-locking excessive amounts of memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to register staging areas for data exchange between host and device.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- cudaHostRegisterPortable: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.

- cudaHostRegisterMapped: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling cudaHostGetDevicePointer(). This feature is available only on GPUs with compute capability greater than or equal to 1.1.

All of these flags are orthogonal to one another: a developer may page-lock memory that is portable or mapped with no restrictions.

The CUDA context must have been created with the cudaMapHost flag in order for the cudaHostRegisterMapped flag to have any effect.

The cudaHostRegisterMapped flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to cudaHostGetDevicePointer() because the memory may be mapped into other CUDA contexts via the cudaHostRegisterPortable flag.

The memory page-locked by this function must be unregistered with cudaHostUnregister().

Parameters:

ptr - Host pointer to memory to page-lock

size - Size in bytes of the address range to page-lock in bytes

flags - Flags for allocation request

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaHostUnregister, cudaHostGetFlags, cudaHostGetDevicePointer

5.7.2.13 cudaError_t cudaHostUnregister (void * *ptr*)

Unmaps the memory range whose base address is specified by `ptr`, and makes it pageable again.

The base address must be the same one specified to cudaHostRegister().

Parameters:

ptr - Host pointer to memory to unregister

Returns:

cudaSuccess, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaHostUnregister

5.7.2.14 __cudart_builtin__ cudaError_t cudaMalloc (void ** *devPtr*, size_t *size*)

Allocates `size` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. cudaMalloc() returns cudaErrorMemoryAllocation in case of failure.

Parameters:

devPtr - Pointer to allocated device memory

size - Requested allocation size in bytes

Returns:

cudaSuccess, cudaErrorMemoryAllocation

See also:

cudaMallocPitch, cudaFree, cudaMallocArray, cudaFreeArray, cudaMalloc3D, cudaMalloc3DArray, cudaMallocHost (C API), cudaFreeHost, cudaHostAlloc

5.7.2.15 cudaError_t cudaMalloc3D (struct cudaPitchedPtr * *pitchedDevPtr*, struct cudaExtent *extent*)

Allocates at least `width * height * depth` bytes of linear memory on the device and returns a cudaPitchedPtr in which `ptr` is a pointer to the allocated memory. The function may pad the allocation to ensure hardware alignment requirements are met. The pitch returned in the `pitch` field of `pitchedDevPtr` is the width in bytes of the allocation.

The returned cudaPitchedPtr contains additional fields `xsize` and `ysize`, the logical width and height of the allocation, which are equivalent to the `width` and `height extent` parameters provided by the programmer during allocation.

For allocations of 2D and 3D objects, it is highly recommended that programmers perform allocations using cudaMalloc3D() or cudaMallocPitch(). Due to alignment restrictions in the hardware, this is especially true if the application will be performing memory copies involving 2D or 3D objects (whether linear memory or CUDA arrays).

Parameters:

pitchedDevPtr - Pointer to allocated pitched device memory

extent - Requested allocation size (`width` field in bytes)

Returns:

cudaSuccess, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMallocPitch, cudaFree, cudaMemcpy3D, cudaMemset3D, cudaMalloc3DArray, cudaMallocArray, cudaFreeArray, cudaMallocHost (C API), cudaFreeHost, cudaHostAlloc, make_cudaPitchedPtr, make_cudaExtent

5.7.2.16 cudaError_t cudaMalloc3DArray (cudaArray_t * *array*, const struct cudaChannelFormatDesc * *desc*, struct cudaExtent *extent*, unsigned int *flags* = `0`)

Allocates a CUDA array according to the cudaChannelFormatDesc structure `desc` and returns a handle to the new CUDA array in `*array`.

The cudaChannelFormatDesc is defined as:

```
struct cudaChannelFormatDesc {
    int x, y, z, w;
    enum cudaChannelFormatKind f;
};
```

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned, cudaChannelFormatKindUnsigned, or cudaChannelFormatKindFloat.

cudaMalloc3DArray() can allocate the following:

- A 1D array is allocated if the height and depth extents are both zero.
- A 2D array is allocated if only the depth extent is zero.
- A 3D array is allocated if all three extents are non-zero.

- A 1D layered CUDA array is allocated if only the height extent is zero and the cudaArrayLayered flag is set. Each layer is a 1D array. The number of layers is determined by the depth extent.
- A 2D layered CUDA array is allocated if all three extents are non-zero and the cudaArrayLayered flag is set. Each layer is a 2D array. The number of layers is determined by the depth extent.
- A cubemap CUDA array is allocated if all three extents are non-zero and the cudaArrayCubemap flag is set. Width must be equal to height, and depth must be six. A cubemap is a special type of 2D layered CUDA array, where the six layers represent the six faces of a cube. The order of the six layers in memory is the same as that listed in cudaGraphicsCubeFace.
- A cubemap layered CUDA array is allocated if all three extents are non-zero, and both, cudaArrayCubemap and cudaArrayLayered flags are set. Width must be equal to height, and depth must be a multiple of six. A cubemap layered CUDA array is a special type of 2D layered CUDA array that consists of a collection of cubemaps. The first six layers represent the first cubemap, the next six layers form the second cubemap, and so on.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- cudaArrayDefault: This flag's value is defined to be 0 and provides default array allocation
- cudaArrayLayered: Allocates a layered CUDA array, with the depth extent indicating the number of layers
- cudaArrayCubemap: Allocates a cubemap CUDA array. Width must be equal to height, and depth must be six. If the cudaArrayLayered flag is also set, depth must be a multiple of six.
- cudaArraySurfaceLoadStore: Allocates a CUDA array that could be read from or written to using a surface reference.
- cudaArrayTextureGather: This flag indicates that texture gather operations will be performed on the CUDA array. Texture gather can only be performed on 2D CUDA arrays.

The width, height and depth extents must meet certain size requirements as listed in the following table. All values are specified in elements.

Note that 2D CUDA arrays have different size requirements if the cudaArrayTextureGather flag is set. In that case, the valid range for (width, height, depth) is ((1,maxTexture2DGather[0]), (1,maxTexture2DGather[1]), 0).

CUDA array type	**Valid extents that must always be met {(width range in elements), (height range), (depth range)}**	**Valid extents with cudaArraySurfaceLoadStore set {(width range in elements), (height range), (depth range)}**
1D	{ (1,maxTexture1D), 0, 0 }	{ (1,maxSurface1D), 0, 0 }
2D	{ (1,maxTexture2D[0]), (1,maxTexture2D[1]), 0 }	{ (1,maxSurface2D[0]), (1,maxSurface2D[1]), 0 }
3D	{ (1,maxTexture3D[0]), (1,maxTexture3D[1]), (1,maxTexture3D[2]) }	{ (1,maxSurface3D[0]), (1,maxSurface3D[1]), (1,maxSurface3D[2]) }
1D Layered	{ (1,maxTexture1DLayered[0]), 0, (1,maxTexture1DLayered[1]) }	{ (1,maxSurface1DLayered[0]), 0, (1,maxSurface1DLayered[1]) }
2D Layered	{ (1,maxTexture2DLayered[0]), (1,maxTexture2DLayered[1]), (1,maxTexture2DLayered[2]) }	{ (1,maxSurface2DLayered[0]), (1,maxSurface2DLayered[1]), (1,maxSurface2DLayered[2]) }
Cubemap	{ (1,maxTextureCubemap), (1,maxTextureCubemap), 6 }	{ (1,maxSurfaceCubemap), (1,maxSurfaceCubemap), 6 }
Cubemap Layered	{ (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[1]) }	{ (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[1]) }

Parameters:

array - Pointer to allocated array in device memory

desc - Requested channel format

extent - Requested allocation size (`width` field in elements)

flags - Flags for extensions

Returns:

cudaSuccess, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMalloc3D, cudaMalloc, cudaMallocPitch, cudaFree, cudaFreeArray, cudaMallocHost (C API), cudaFreeHost, cudaHostAlloc, make_cudaExtent

5.7.2.17 cudaError_t cudaMallocArray (cudaArray_t * *array*, const struct cudaChannelFormatDesc * *desc*, size_t *width*, size_t *height* = 0, unsigned int *flags* = 0)

Allocates a CUDA array according to the cudaChannelFormatDesc structure `desc` and returns a handle to the new CUDA array in `*array`.

The cudaChannelFormatDesc is defined as:

```
struct cudaChannelFormatDesc {
    int x, y, z, w;
enum cudaChannelFormatKind f;
};
```

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned, cudaChannelFormatKindUnsigned, or cudaChannelFormatKindFloat.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- cudaArrayDefault: This flag's value is defined to be 0 and provides default array allocation
- cudaArraySurfaceLoadStore: Allocates an array that can be read from or written to using a surface reference
- cudaArrayTextureGather: This flag indicates that texture gather operations will be performed on the array.

`width` and `height` must meet certain size requirements. See cudaMalloc3DArray() for more details.

Parameters:

array - Pointer to allocated array in device memory

desc - Requested channel format

width - Requested array allocation width

height - Requested array allocation height

flags - Requested properties of allocated array

Returns:

cudaSuccess, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMalloc, cudaMallocPitch, cudaFree, cudaFreeArray, cudaMallocHost (C API), cudaFreeHost, cudaMalloc3D, cudaMalloc3DArray, cudaHostAlloc

5.7.2.18 cudaError_t cudaMallocHost (void ** *ptr*, size_t *size*)

Allocates `size` bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as cudaMemcpy*(). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc(). Allocating excessive amounts of memory with cudaMallocHost() may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

Parameters:

ptr - Pointer to allocated host memory

size - Requested allocation size in bytes

Returns:

cudaSuccess, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMalloc, cudaMallocPitch, cudaMallocArray, cudaMalloc3D, cudaMalloc3DArray, cudaHostAlloc, cudaFree, cudaFreeArray, cudaMallocHost (C++ API), cudaFreeHost, cudaHostAlloc

5.7.2.19 cudaError_t cudaMallocMipmappedArray (cudaMipmappedArray_t * *mipmappedArray*, const struct cudaChannelFormatDesc * *desc*, struct cudaExtent *extent*, unsigned int *numLevels*, unsigned int *flags* = 0)

Allocates a CUDA mipmapped array according to the cudaChannelFormatDesc structure `desc` and returns a handle to the new CUDA mipmapped array in `*mipmappedArray`. `numLevels` specifies the number of mipmap levels to be allocated. This value is clamped to the range [1, 1 + floor(log2(max(width, height, depth)))].

The cudaChannelFormatDesc is defined as:

```
struct cudaChannelFormatDesc {
    int x, y, z, w;
    enum cudaChannelFormatKind f;
};
```

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned, cudaChannelFormatKindUnsigned, or cudaChannelFormatKindFloat.

cudaMallocMipmappedArray() can allocate the following:

- A 1D mipmapped array is allocated if the height and depth extents are both zero.
- A 2D mipmapped array is allocated if only the depth extent is zero.
- A 3D mipmapped array is allocated if all three extents are non-zero.
- A 1D layered CUDA mipmapped array is allocated if only the height extent is zero and the cudaArrayLayered flag is set. Each layer is a 1D mipmapped array. The number of layers is determined by the depth extent.
- A 2D layered CUDA mipmapped array is allocated if all three extents are non-zero and the cudaArrayLayered flag is set. Each layer is a 2D mipmapped array. The number of layers is determined by the depth extent.
- A cubemap CUDA mipmapped array is allocated if all three extents are non-zero and the cudaArrayCubemap flag is set. Width must be equal to height, and depth must be six. The order of the six layers in memory is the same as that listed in cudaGraphicsCubeFace.
- A cubemap layered CUDA mipmapped array is allocated if all three extents are non-zero, and both, cudaArrayCubemap and cudaArrayLayered flags are set. Width must be equal to height, and depth must be a multiple of six. A cubemap layered CUDA mipmapped array is a special type of 2D layered CUDA mipmapped array that consists of a collection of cubemap mipmapped arrays. The first six layers represent the first cubemap mipmapped array, the next six layers form the second cubemap mipmapped array, and so on.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- cudaArrayDefault: This flag's value is defined to be 0 and provides default mipmapped array allocation
- cudaArrayLayered: Allocates a layered CUDA mipmapped array, with the depth extent indicating the number of layers
- cudaArrayCubemap: Allocates a cubemap CUDA mipmapped array. Width must be equal to height, and depth must be six. If the cudaArrayLayered flag is also set, depth must be a multiple of six.
- cudaArraySurfaceLoadStore: This flag indicates that individual mipmap levels of the CUDA mipmapped array will be read from or written to using a surface reference.
- cudaArrayTextureGather: This flag indicates that texture gather operations will be performed on the CUDA array. Texture gather can only be performed on 2D CUDA mipmapped arrays, and the gather operations are performed only on the most detailed mipmap level.

The width, height and depth extents must meet certain size requirements as listed in the following table. All values are specified in elements.

CUDA array type	**Valid extents {(width range in elements), (height range), (depth range)}**
1D	{ (1,maxTexture1DMipmap), 0, 0 }
2D	{ (1,maxTexture2DMipmap[0]), (1,maxTexture2DMipmap[1]), 0 }
3D	{ (1,maxTexture3D[0]), (1,maxTexture3D[1]), (1,maxTexture3D[2]) }
1D Layered	{ (1,maxTexture1DLayered[0]), 0, (1,maxTexture1DLayered[1]) }
2D Layered	{ (1,maxTexture2DLayered[0]), (1,maxTexture2DLayered[1]), (1,maxTexture2DLayered[2]) }
Cubemap	{ (1,maxTextureCubemap), (1,maxTextureCubemap), 6 }
Cubemap Layered	{ (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[1]) }

Parameters:

mipmappedArray - Pointer to allocated mipmapped array in device memory

desc - Requested channel format

extent - Requested allocation size (`width` field in elements)

numLevels - Number of mipmap levels to allocate

flags - Flags for extensions

Returns:

cudaSuccess, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMalloc3D, cudaMalloc, cudaMallocPitch, cudaFree, cudaFreeArray, cudaMallocHost (C API), cudaFreeHost, cudaHostAlloc, make_cudaExtent

5.7.2.20 cudaError_t cudaMallocPitch (void ** *devPtr*, size_t * *pitch*, size_t *width*, size_t *height*)

Allocates at least `width` (in bytes) * `height` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. The function may pad the allocation to ensure that corresponding pointers in any given row will continue to meet the alignment requirements for coalescing as the address is updated from row to row. The pitch returned in `*pitch` by cudaMallocPitch() is the width in bytes of the allocation. The intended usage of `pitch` is as a separate parameter of the allocation, used to compute addresses within the 2D array. Given the row and column of an array element of type `T`, the address is computed as:

```
T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;
```

For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using cudaMallocPitch(). Due to pitch alignment restrictions in the hardware, this is especially true if the application will be performing 2D memory copies between different regions of device memory (whether linear memory or CUDA arrays).

Parameters:

devPtr - Pointer to allocated pitched device memory

pitch - Pitch for allocation

width - Requested pitched allocation width (in bytes)

height - Requested pitched allocation height

Returns:

cudaSuccess, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMalloc, cudaFree, cudaMallocArray, cudaFreeArray, cudaMallocHost (C API), cudaFreeHost, cudaMalloc3D, cudaMalloc3DArray, cudaHostAlloc

5.7.2.21 cudaError_t cudaMemcpy (void * *dst*, const void * *src*, size_t *count*, enum cudaMemcpyKind *kind*)

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. The memory areas may not overlap. Calling cudaMemcpy() with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior.

Parameters:

dst - Destination memory address

src - Source memory address

count - Size in bytes to copy

kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.

See also:

cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.22 cudaError_t cudaMemcpy2D (void * *dst*, size_t *dpitch*, const void * *src*, size_t *spitch*, size_t *width*, size_t *height*, enum cudaMemcpyKind *kind*)

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. `dpitch` and `spitch` are the widths in memory in bytes of the 2D arrays pointed to by `dst` and `src`, including any padding added to the end of each row. The memory areas may not overlap. `width` must not exceed either `dpitch` or `spitch`. Calling cudaMemcpy2D() with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior. cudaMemcpy2D() returns an error if `dpitch` or `spitch` exceeds the maximum allowed.

Parameters:

dst - Destination memory address

dpitch - Pitch of destination memory

src - Source memory address

spitch - Pitch of source memory

width - Width of matrix transfer (columns in bytes)

height - Height of matrix transfer (rows)

kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMemcpy, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.23 cudaError_t cudaMemcpy2DArrayToArray (cudaArray_t *dst*, size_t *wOffsetDst*, size_t *hOffsetDst*, cudaArray_const_t *src*, size_t *wOffsetSrc*, size_t *hOffsetSrc*, size_t *width*, size_t *height*, enum cudaMemcpyKind *kind* = `cudaMemcpyDeviceToDevice`)

Copies a matrix (`height` rows of `width` bytes each) from the CUDA array `srcArray` starting at the upper left corner (`wOffsetSrc`, `hOffsetSrc`) to the CUDA array `dst` starting at the upper left corner (`wOffsetDst`, `hOffsetDst`), where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. `wOffsetDst` + `width` must not exceed the width of the CUDA array `dst`. `wOffsetSrc` + `width` must not exceed the width of the CUDA array `src`.

Parameters:

dst - Destination memory address
wOffsetDst - Destination starting X offset
hOffsetDst - Destination starting Y offset
src - Source memory address
wOffsetSrc - Source starting X offset
hOffsetSrc - Source starting Y offset
width - Width of matrix transfer (columns in bytes)
height - Height of matrix transfer (rows)
kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.24 __cudart_builtin__ cudaError_t cudaMemcpy2DAsync (void * *dst*, size_t *dpitch*, const void * *src*, size_t *spitch*, size_t *width*, size_t *height*, enum cudaMemcpyKind *kind*, cudaStream_t *stream* = 0)

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. `dpitch` and `spitch` are the widths in memory in bytes of the 2D arrays pointed to by `dst` and `src`, including any padding added to the end of each row. The memory areas may not overlap. `width` must not exceed either `dpitch` or `spitch`. Calling cudaMemcpy2DAsync() with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior. cudaMemcpy2DAsync() returns an error if `dpitch` or `spitch` is greater than the maximum allowed.

cudaMemcpy2DAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

dst - Destination memory address

dpitch - Pitch of destination memory

src - Source memory address

spitch - Pitch of source memory

width - Width of matrix transfer (columns in bytes)

height - Height of matrix transfer (rows)

kind - Type of transfer

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.25 cudaError_t cudaMemcpy2DFromArray (void * *dst*, size_t *dpitch*, cudaArray_const_t *src*, size_t *wOffset*, size_t *hOffset*, size_t *width*, size_t *height*, enum cudaMemcpyKind *kind*)

Copies a matrix (`height` rows of `width` bytes each) from the CUDA array `srcArray` starting at the upper left corner (`wOffset`, `hOffset`) to the memory area pointed to by `dst`, where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. `dpitch` is the width in memory in bytes of the 2D array pointed to by `dst`, including any padding added to the end of each row. `wOffset` + `width` must not exceed the width of the CUDA array `src`. `width` must not exceed `dpitch`. cudaMemcpy2DFromArray() returns an error if `dpitch` exceeds the maximum allowed.

Parameters:

dst - Destination memory address

dpitch - Pitch of destination memory

src - Source memory address

wOffset - Source starting X offset

hOffset - Source starting Y offset

width - Width of matrix transfer (columns in bytes)

height - Height of matrix transfer (rows)

kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.26 cudaError_t cudaMemcpy2DFromArrayAsync (void * *dst*, size_t *dpitch*, cudaArray_const_t *src*, size_t *wOffset*, size_t *hOffset*, size_t *width*, size_t *height*, enum cudaMemcpyKind *kind*, cudaStream_t *stream* = 0)

Copies a matrix (`height` rows of `width` bytes each) from the CUDA array `srcArray` starting at the upper left corner (`wOffset`, `hOffset`) to the memory area pointed to by `dst`, where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. `dpitch` is the width in memory in bytes of the 2D array pointed to by `dst`, including any padding added to the end of each row. `wOffset` + `width` must not exceed the width of the CUDA array `src`. `width` must not exceed `dpitch`. cudaMemcpy2DFromArrayAsync() returns an error if `dpitch` exceeds the maximum allowed.

cudaMemcpy2DFromArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

dst - Destination memory address

dpitch - Pitch of destination memory

src - Source memory address

wOffset - Source starting X offset

hOffset - Source starting Y offset

width - Width of matrix transfer (columns in bytes)

height - Height of matrix transfer (rows)

kind - Type of transfer

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.27 cudaError_t cudaMemcpy2DToArray (cudaArray_t *dst*, size_t *wOffset*, size_t *hOffset*, const void * *src*, size_t *spitch*, size_t *width*, size_t *height*, enum cudaMemcpyKind *kind*)

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the CUDA array `dst` starting at the upper left corner (`wOffset`, `hOffset`) where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. `spitch` is the width in memory in bytes of the 2D array pointed to by `src`, including any padding added to the end of each row. `wOffset` + `width` must not exceed the width of the CUDA array `dst`. `width` must not exceed `spitch`. cudaMemcpy2DToArray() returns an error if `spitch` exceeds the maximum allowed.

Parameters:

dst - Destination memory address

wOffset - Destination starting X offset

hOffset - Destination starting Y offset

src - Source memory address

spitch - Pitch of source memory

width - Width of matrix transfer (columns in bytes)

height - Height of matrix transfer (rows)

kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.28 cudaError_t cudaMemcpy2DToArrayAsync (cudaArray_t *dst*, size_t *wOffset*, size_t *hOffset*, const void * *src*, size_t *spitch*, size_t *width*, size_t *height*, enum cudaMemcpyKind *kind*, cudaStream_t *stream* = 0)

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the CUDA array `dst` starting at the upper left corner (`wOffset`, `hOffset`) where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. `spitch` is the width in memory in bytes of the 2D array pointed to by `src`, including any padding added to the end of each row. `wOffset` + `width` must not exceed the width of the CUDA array `dst`. `width` must not exceed `spitch`. cudaMemcpy2DToArrayAsync() returns an error if `spitch` exceeds the maximum allowed.

cudaMemcpy2DToArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

dst - Destination memory address

wOffset - Destination starting X offset

hOffset - Destination starting Y offset

src - Source memory address

spitch - Pitch of source memory

width - Width of matrix transfer (columns in bytes)

height - Height of matrix transfer (rows)

kind - Type of transfer

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.29 cudaError_t cudaMemcpy3D (const struct cudaMemcpy3DParms * *p*)

```
struct cudaExtent {
  size_t width;
  size_t height;
  size_t depth;
};
struct cudaExtent make_cudaExtent(size_t w, size_t h, size_t d);

struct cudaPos {
  size_t x;
  size_t y;
  size_t z;
};
struct cudaPos make_cudaPos(size_t x, size_t y, size_t z);

struct cudaMemcpy3DParms {
  cudaArray_t           srcArray;
  struct cudaPos        srcPos;
  struct cudaPitchedPtr srcPtr;
  cudaArray_t           dstArray;
  struct cudaPos        dstPos;
  struct cudaPitchedPtr dstPtr;
  struct cudaExtent     extent;
  enum cudaMemcpyKind   kind;
};
```

cudaMemcpy3D() copies data betwen two 3D objects. The source and destination objects may be in either host memory, device memory, or a CUDA array. The source, destination, extent, and kind of copy performed is specified by the cudaMemcpy3DParms struct which should be initialized to zero before use:

```
cudaMemcpy3DParms myParms = {0};
```

The struct passed to cudaMemcpy3D() must specify one of `srcArray` or `srcPtr` and one of `dstArray` or `dstPtr`. Passing more than one non-zero source or destination will cause cudaMemcpy3D() to return an error.

The `srcPos` and `dstPos` fields are optional offsets into the source and destination objects and are defined in units of each object's elements. The element for a host or device pointer is assumed to be **unsigned char**. For CUDA arrays, positions must be in the range [0, 2048) for any dimension.

The `extent` field defines the dimensions of the transferred area in elements. If a CUDA array is participating in the copy, the extent is defined in terms of that array's elements. If no CUDA array is participating in the copy then the extents are defined in elements of **unsigned char**.

The `kind` field defines the direction of the copy. It must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice.

If the source and destination are both arrays, cudaMemcpy3D() will return an error if they do not have the same element size.

The source and destination object may not overlap. If overlapping source and destination objects are specified, undefined behavior will result.

The source object must lie entirely within the region defined by `srcPos` and `extent`. The destination object must lie entirely within the region defined by `dstPos` and `extent`.

cudaMemcpy3D() returns an error if the pitch of `srcPtr` or `dstPtr` exceeds the maximum allowed. The pitch of a cudaPitchedPtr allocated with cudaMalloc3D() will always be valid.

Parameters:

p - 3D memory copy parameters

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.

See also:

cudaMalloc3D, cudaMalloc3DArray, cudaMemset3D, cudaMemcpy3DAsync, cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, make_cudaExtent, make_cudaPos

5.7.2.30 __cudart_builtin__ cudaError_t cudaMemcpy3DAsync (const struct cudaMemcpy3DParms * *p*, cudaStream_t *stream* = 0)

```
struct cudaExtent {
  size_t width;
  size_t height;
  size_t depth;
};
struct cudaExtent make_cudaExtent(size_t w, size_t h, size_t d);

struct cudaPos {
  size_t x;
  size_t y;
  size_t z;
};
struct cudaPos make_cudaPos(size_t x, size_t y, size_t z);

struct cudaMemcpy3DParms {
  cudaArray_t           srcArray;
  struct cudaPos        srcPos;
  struct cudaPitchedPtr srcPtr;
  cudaArray_t           dstArray;
  struct cudaPos        dstPos;
  struct cudaPitchedPtr dstPtr;
  struct cudaExtent     extent;
  enum cudaMemcpyKind   kind;
};
```

cudaMemcpy3DAsync() copies data betwen two 3D objects. The source and destination objects may be in either host memory, device memory, or a CUDA array. The source, destination, extent, and kind of copy performed is specified by the cudaMemcpy3DParms struct which should be initialized to zero before use:

```
cudaMemcpy3DParms myParms = {0};
```

The struct passed to cudaMemcpy3DAsync() must specify one of `srcArray` or `srcPtr` and one of `dstArray` or `dstPtr`. Passing more than one non-zero source or destination will cause cudaMemcpy3DAsync() to return an error.

The `srcPos` and `dstPos` fields are optional offsets into the source and destination objects and are defined in units of each object's elements. The element for a host or device pointer is assumed to be **unsigned char**. For CUDA arrays, positions must be in the range [0, 2048) for any dimension.

The `extent` field defines the dimensions of the transferred area in elements. If a CUDA array is participating in the copy, the extent is defined in terms of that array's elements. If no CUDA array is participating in the copy then the extents are defined in elements of **unsigned char**.

The `kind` field defines the direction of the copy. It must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice.

If the source and destination are both arrays, cudaMemcpy3DAsync() will return an error if they do not have the same element size.

The source and destination object may not overlap. If overlapping source and destination objects are specified, undefined behavior will result.

The source object must lie entirely within the region defined by `srcPos` and `extent`. The destination object must lie entirely within the region defined by `dstPos` and `extent`.

cudaMemcpy3DAsync() returns an error if the pitch of `srcPtr` or `dstPtr` exceeds the maximum allowed. The pitch of a cudaPitchedPtr allocated with cudaMalloc3D() will always be valid.

cudaMemcpy3DAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

p - 3D memory copy parameters

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidPitchValue, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.

See also:

cudaMalloc3D, cudaMalloc3DArray, cudaMemset3D, cudaMemcpy3D, cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync, make_cudaExtent, make_cudaPos

5.7.2.31 cudaError_t cudaMemcpy3DPeer (const struct cudaMemcpy3DPeerParms * *p*)

Perform a 3D memory copy according to the parameters specified in `p`. See the definition of the cudaMemcpy3DPeerParms structure for documentation of its parameters.

Note that this function is synchronous with respect to the host only if the source or destination of the transfer is host memory. Note also that this copy is serialized with respect to all pending and future asynchronous work in to the current device, the copy's source device, and the copy's destination device (use cudaMemcpy3DPeerAsync to avoid this synchronization).

Parameters:

p - Parameters for the memory copy

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpyPeerAsync, cudaMemcpy3DPeerAsync

5.7.2.32 cudaError_t cudaMemcpy3DPeerAsync (const struct cudaMemcpy3DPeerParms * *p*, cudaStream_t *stream* = 0)

Perform a 3D memory copy according to the parameters specified in `p`. See the definition of the cudaMemcpy3DPeerParms structure for documentation of its parameters.

Parameters:

p - Parameters for the memory copy

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpyPeerAsync, cudaMemcpy3DPeerAsync

5.7.2.33 cudaError_t cudaMemcpyArrayToArray (cudaArray_t *dst*, size_t *wOffsetDst*, size_t *hOffsetDst*, cudaArray_const_t *src*, size_t *wOffsetSrc*, size_t *hOffsetSrc*, size_t *count*, enum cudaMemcpyKind *kind* = cudaMemcpyDeviceToDevice)

Copies `count` bytes from the CUDA array `src` starting at the upper left corner (`wOffsetSrc`, `hOffsetSrc`) to the CUDA array `dst` starting at the upper left corner (`wOffsetDst`, `hOffsetDst`) where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy.

Parameters:

dst - Destination memory address

wOffsetDst - Destination starting X offset

hOffsetDst - Destination starting Y offset

src - Source memory address

wOffsetSrc - Source starting X offset

hOffsetSrc - Source starting Y offset

count - Size in bytes to copy

kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.34 __cudart_builtin__ cudaError_t cudaMemcpyAsync (void * *dst*, const void * *src*, size_t *count*, enum cudaMemcpyKind *kind*, cudaStream_t *stream* = 0)

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy. The memory areas may not overlap. Calling cudaMemcpyAsync() with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior.

cudaMemcpyAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and the `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

dst - Destination memory address

src - Source memory address

count - Size in bytes to copy

kind - Type of transfer

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.35 cudaError_t cudaMemcpyFromArray (void * *dst*, cudaArray_const_t *src*, size_t *wOffset*, size_t *hOffset*, size_t *count*, enum cudaMemcpyKind *kind*)

Copies `count` bytes from the CUDA array `src` starting at the upper left corner (`wOffset`, hOffset) to the memory area pointed to by `dst`, where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy.

Parameters:

dst - Destination memory address

src - Source memory address

wOffset - Source starting X offset

hOffset - Source starting Y offset

count - Size in bytes to copy

kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.36 cudaError_t cudaMemcpyFromArrayAsync (void * *dst*, cudaArray_const_t *src*, size_t *wOffset*, size_t *hOffset*, size_t *count*, enum cudaMemcpyKind *kind*, cudaStream_t *stream* = 0)

Copies `count` bytes from the CUDA array `src` starting at the upper left corner (`wOffset`, hOffset) to the memory area pointed to by `dst`, where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy.

cudaMemcpyFromArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

dst - Destination memory address

src - Source memory address

wOffset - Source starting X offset

hOffset - Source starting Y offset

count - Size in bytes to copy

kind - Type of transfer

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.37 cudaError_t cudaMemcpyFromSymbol (void * *dst*, const void * *symbol*, size_t *count*, size_t *offset* = `0`, enum cudaMemcpyKind *kind* = `cudaMemcpyDeviceToHost`)

Copies `count` bytes from the memory area pointed to by `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either cudaMemcpyDeviceToHost or cudaMemcpyDeviceToDevice.

Parameters:

dst - Destination memory address

symbol - Device symbol address

count - Size in bytes to copy

offset - Offset from start of symbol in bytes

kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.
Use of a string naming a variable as the `symbol` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.38 cudaError_t cudaMemcpyFromSymbolAsync (void * *dst*, const void * *symbol*, size_t *count*, size_t *offset*, enum cudaMemcpyKind *kind*, cudaStream_t *stream* = 0)

Copies `count` bytes from the memory area pointed to by `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either cudaMemcpyDeviceToHost or cudaMemcpyDeviceToDevice.

cudaMemcpyFromSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is cudaMemcpyDeviceToHost and `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

dst - Destination memory address

symbol - Device symbol address

count - Size in bytes to copy

offset - Offset from start of symbol in bytes

kind - Type of transfer

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.
Use of a string naming a variable as the `symbol` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync

5.7.2.39 cudaError_t cudaMemcpyPeer (void * *dst*, int *dstDevice*, const void * *src*, int *srcDevice*, size_t *count*)

Copies memory from one device to memory on another device. `dst` is the base device pointer of the destination memory and `dstDevice` is the destination device. `src` is the base device pointer of the source memory and `srcDevice` is the source device. `count` specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host, but serialized with respect all pending and future asynchronous work in to the current device, `srcDevice`, and `dstDevice` (use cudaMemcpyPeerAsync to avoid this synchronization).

Parameters:

dst - Destination device pointer

dstDevice - Destination device

src - Source device pointer

srcDevice - Source device

count - Size of memory copy in bytes

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpyAsync, cudaMemcpyPeerAsync, cudaMemcpy3DPeerAsync

5.7.2.40 cudaError_t cudaMemcpyPeerAsync (void * *dst*, int *dstDevice*, const void * *src*, int *srcDevice*, size_t *count*, cudaStream_t *stream* = 0)

Copies memory from one device to memory on another device. `dst` is the base device pointer of the destination memory and `dstDevice` is the destination device. `src` is the base device pointer of the source memory and `srcDevice` is the source device. `count` specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host and all work in other streams and other devices.

Parameters:

dst - Destination device pointer

dstDevice - Destination device

src - Source device pointer

srcDevice - Source device

count - Size of memory copy in bytes

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevice

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpyPeer, cudaMemcpyAsync, cudaMemcpy3DPeerAsync

5.7.2.41 cudaError_t cudaMemcpyToArray (cudaArray_t *dst*, size_t *wOffset*, size_t *hOffset*, const void * *src*, size_t *count*, enum cudaMemcpyKind *kind*)

Copies `count` bytes from the memory area pointed to by `src` to the CUDA array `dst` starting at the upper left corner (`wOffset`, `hOffset`), where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy.

Parameters:

dst - Destination memory address

wOffset - Destination starting X offset

hOffset - Destination starting Y offset

src - Source memory address

count - Size in bytes to copy

kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.42 cudaError_t cudaMemcpyToArrayAsync (cudaArray_t *dst*, size_t *wOffset*, size_t *hOffset*, const void * *src*, size_t *count*, enum cudaMemcpyKind *kind*, cudaStream_t *stream* = 0)

Copies `count` bytes from the memory area pointed to by `src` to the CUDA array `dst` starting at the upper left corner (`wOffset`, `hOffset`), where `kind` is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice, and specifies the direction of the copy.

cudaMemcpyToArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

dst - Destination memory address

wOffset - Destination starting X offset

hOffset - Destination starting Y offset

src - Source memory address

count - Size in bytes to copy

kind - Type of transfer

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.43 cudaError_t cudaMemcpyToSymbol (const void * *symbol*, const void * *src*, size_t *count*, size_t *offset* = 0, enum cudaMemcpyKind *kind* = cudaMemcpyHostToDevice)

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either cudaMemcpyHostToDevice or cudaMemcpyDeviceToDevice.

Parameters:

symbol - Device symbol address

src - Source memory address

count - Size in bytes to copy

offset - Offset from start of symbol in bytes

kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.
Use of a string naming a variable as the `symbol` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.7.2.44 cudaError_t cudaMemcpyToSymbolAsync (const void * *symbol*, const void * *src*, size_t *count*, size_t *offset*, enum cudaMemcpyKind *kind*, cudaStream_t *stream* = 0)

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either cudaMemcpyHostToDevice or cudaMemcpyDeviceToDevice.

cudaMemcpyToSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is cudaMemcpyHostToDevice and `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

symbol - Device symbol address

src - Source memory address

count - Size in bytes to copy

offset - Offset from start of symbol in bytes

kind - Type of transfer

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.
Use of a string naming a variable as the `symbol` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyFromSymbolAsync

5.7.2.45 cudaError_t cudaMemGetInfo (size_t * *free*, size_t * *total*)

Returns in `*free` and `*total` respectively, the free and total amount of memory available for allocation by the device in bytes.

Parameters:

free - Returned free memory in bytes

total - Returned total memory in bytes

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorLaunchFailure

Note:

Note that this function may also return error codes from previous, asynchronous launches.

5.7.2.46 cudaError_t cudaMemset (void * *devPtr*, int *value*, size_t *count*)

Fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte value `value`.

Note that this function is asynchronous with respect to the host unless `devPtr` refers to pinned host memory.

Parameters:

devPtr - Pointer to device memory

value - Value to set for each byte of specified memory

count - Size in bytes to set

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer

Note:

Note that this function may also return error codes from previous, asynchronous launches.
See also memset synchronization details.

See also:

cudaMemset2D, cudaMemset3D, cudaMemsetAsync, cudaMemset2DAsync, cudaMemset3DAsync

5.7.2.47 cudaError_t cudaMemset2D (void * *devPtr*, size_t *pitch*, int *value*, size_t *width*, size_t *height*)

Sets to the specified value `value` a matrix (`height` rows of `width` bytes each) pointed to by `dstPtr`. `pitch` is the width in bytes of the 2D array pointed to by `dstPtr`, including any padding added to the end of each row. This function performs fastest when the pitch is one that has been passed back by cudaMallocPitch().

Note that this function is asynchronous with respect to the host unless `devPtr` refers to pinned host memory.

Parameters:

devPtr - Pointer to 2D device memory

pitch - Pitch in bytes of 2D device memory

value - Value to set for each byte of specified memory

width - Width of matrix set (columns in bytes)

height - Height of matrix set (rows)

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer

Note:

Note that this function may also return error codes from previous, asynchronous launches.
See also memset synchronization details.

See also:

cudaMemset, cudaMemset3D, cudaMemsetAsync, cudaMemset2DAsync, cudaMemset3DAsync

5.7.2.48 __cudart_builtin__ cudaError_t cudaMemset2DAsync (void * *devPtr*, size_t *pitch*, int *value*, size_t *width*, size_t *height*, cudaStream_t *stream* = 0)

Sets to the specified value `value` a matrix (`height` rows of `width` bytes each) pointed to by `dstPtr`. `pitch` is the width in bytes of the 2D array pointed to by `dstPtr`, including any padding added to the end of each row. This function performs fastest when the pitch is one that has been passed back by cudaMallocPitch().

cudaMemset2DAsync() is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

Parameters:

devPtr - Pointer to 2D device memory

pitch - Pitch in bytes of 2D device memory

value - Value to set for each byte of specified memory

width - Width of matrix set (columns in bytes)

height - Height of matrix set (rows)

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer

Note:

Note that this function may also return error codes from previous, asynchronous launches.
See also memset synchronization details.

See also:

cudaMemset, cudaMemset2D, cudaMemset3D, cudaMemsetAsync, cudaMemset3DAsync

5.7.2.49 cudaError_t cudaMemset3D (struct cudaPitchedPtr *pitchedDevPtr*, int *value*, struct cudaExtent *extent*)

Initializes each element of a 3D array to the specified value `value`. The object to initialize is defined by `pitchedDevPtr`. The `pitch` field of `pitchedDevPtr` is the width in memory in bytes of the 3D array pointed to by `pitchedDevPtr`, including any padding added to the end of each row. The `xsize` field specifies the logical width of each row in bytes, while the `ysize` field specifies the height of each 2D slice in rows.

The extents of the initialized region are specified as a `width` in bytes, a `height` in rows, and a `depth` in slices.

Extents with `width` greater than or equal to the `xsize` of `pitchedDevPtr` may perform significantly faster than extents narrower than the `xsize`. Secondarily, extents with `height` equal to the `ysize` of `pitchedDevPtr` will perform faster than when the `height` is shorter than the `ysize`.

This function performs fastest when the `pitchedDevPtr` has been allocated by cudaMalloc3D().

Note that this function is asynchronous with respect to the host unless `pitchedDevPtr` refers to pinned host memory.

Parameters:

pitchedDevPtr - Pointer to pitched device memory

value - Value to set for each byte of specified memory

extent - Size parameters for where to set device memory (`width` field in bytes)

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer

Note:

Note that this function may also return error codes from previous, asynchronous launches.
See also memset synchronization details.

See also:

cudaMemset, cudaMemset2D, cudaMemsetAsync, cudaMemset2DAsync, cudaMemset3DAsync, cudaMalloc3D, make_cudaPitchedPtr, make_cudaExtent

5.7.2.50 __cudart_builtin__ cudaError_t cudaMemset3DAsync (struct cudaPitchedPtr *pitchedDevPtr*, int *value*, struct cudaExtent *extent*, cudaStream_t *stream* = 0)

Initializes each element of a 3D array to the specified value `value`. The object to initialize is defined by `pitchedDevPtr`. The `pitch` field of `pitchedDevPtr` is the width in memory in bytes of the 3D array pointed to by `pitchedDevPtr`, including any padding added to the end of each row. The `xsize` field specifies the logical width of each row in bytes, while the `ysize` field specifies the height of each 2D slice in rows.

The extents of the initialized region are specified as a `width` in bytes, a `height` in rows, and a `depth` in slices.

Extents with `width` greater than or equal to the `xsize` of `pitchedDevPtr` may perform significantly faster than extents narrower than the `xsize`. Secondarily, extents with `height` equal to the `ysize` of `pitchedDevPtr` will perform faster than when the `height` is shorter than the `ysize`.

This function performs fastest when the `pitchedDevPtr` has been allocated by cudaMalloc3D().

cudaMemset3DAsync() is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

Parameters:

pitchedDevPtr - Pointer to pitched device memory

value - Value to set for each byte of specified memory

extent - Size parameters for where to set device memory (`width` field in bytes)

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer

Note:

Note that this function may also return error codes from previous, asynchronous launches.
See also memset synchronization details.

See also:

cudaMemset, cudaMemset2D, cudaMemset3D, cudaMemsetAsync, cudaMemset2DAsync, cudaMalloc3D, make_cudaPitchedPtr, make_cudaExtent

5.7.2.51 __cudart_builtin__ cudaError_t cudaMemsetAsync (void * *devPtr*, int *value*, size_t *count*, cudaStream_t *stream* = 0)

Fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte value `value`.

cudaMemsetAsync() is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

Parameters:

devPtr - Pointer to device memory

value - Value to set for each byte of specified memory

count - Size in bytes to set

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer

Note:

Note that this function may also return error codes from previous, asynchronous launches.
See also memset synchronization details.

See also:

cudaMemset, cudaMemset2D, cudaMemset3D, cudaMemset2DAsync, cudaMemset3DAsync

5.7.2.52 struct cudaExtent make_cudaExtent (size_t *w*, size_t *h*, size_t *d*) [read]

Returns a cudaExtent based on the specified input parameters `w`, `h`, and `d`.

Parameters:

w - Width in bytes

h - Height in elements

d - Depth in elements

Returns:

cudaExtent specified by `w`, `h`, and `d`

See also:

make_cudaPitchedPtr, make_cudaPos

5.7.2.53 struct cudaPitchedPtr make_cudaPitchedPtr (void * *d*, size_t *p*, size_t *xsz*, size_t *ysz*) [read]

Returns a cudaPitchedPtr based on the specified input parameters `d`, `p`, `xsz`, and `ysz`.

Parameters:

d - Pointer to allocated memory

p - Pitch of allocated memory in bytes

xsz - Logical width of allocation in elements

ysz - Logical height of allocation in elements

Returns:

cudaPitchedPtr specified by `d`, `p`, `xsz`, and `ysz`

See also:

make_cudaExtent, make_cudaPos

5.7.2.54 struct cudaPos make_cudaPos (size_t *x*, size_t *y*, size_t *z*) [read]

Returns a cudaPos based on the specified input parameters `x`, `y`, and `z`.

Parameters:

x - X position

y - Y position

z - Z position

Returns:

cudaPos specified by `x`, `y`, and `z`

See also:

make_cudaExtent, make_cudaPitchedPtr

5.8 Unified Addressing

Functions

- cudaError_t cudaPointerGetAttributes (struct cudaPointerAttributes *attributes, const void *ptr)

  Returns attributes about a specified pointer.

5.8.1 Detailed Description

This section describes the unified addressing functions of the CUDA runtime application programming interface.

5.8.2 Overview

CUDA devices can share a unified address space with the host. For these devices there is no distinction between a device pointer and a host pointer – the same pointer value may be used to access memory from the host program and from a kernel running on the device (with exceptions enumerated below).

5.8.3 Supported Platforms

Whether or not a device supports unified addressing may be queried by calling cudaGetDeviceProperties() with the device property cudaDeviceProp::unifiedAddressing.

Unified addressing is automatically enabled in 64-bit processes on devices with compute capability greater than or equal to 2.0.

Unified addressing is not yet supported on Windows Vista or Windows 7 for devices that do not use the TCC driver model.

5.8.4 Looking Up Information from Pointer Values

It is possible to look up information about the memory which backs a pointer value. For instance, one may want to know if a pointer points to host or device memory. As another example, in the case of device memory, one may want to know on which CUDA device the memory resides. These properties may be queried using the function cudaPointerGetAttributes()

Since pointers are unique, it is not necessary to specify information about the pointers specified to cudaMemcpy() and other copy functions. The copy direction cudaMemcpyDefault may be used to specify that the CUDA runtime should infer the location of the pointer from its value.

5.8.5 Automatic Mapping of Host Allocated Host Memory

All host memory allocated through all devices using cudaMallocHost() and cudaHostAlloc() is always directly accessible from all devices that support unified addressing. This is the case regardless of whether or not the flags cudaHostAllocPortable and cudaHostAllocMapped are specified.

The pointer value through which allocated host memory may be accessed in kernels on all devices that support unified addressing is the same as the pointer value through which that memory is accessed on the host. It is not necessary to call cudaHostGetDevicePointer() to get the device pointer for these allocations.

Note that this is not the case for memory allocated using the flag cudaHostAllocWriteCombined, as discussed below.

5.8.6 Direct Access of Peer Memory

Upon enabling direct access from a device that supports unified addressing to another peer device that supports unified addressing using cudaDeviceEnablePeerAccess() all memory allocated in the peer device using cudaMalloc() and cudaMallocPitch() will immediately be accessible by the current device. The device pointer value through which any peer's memory may be accessed in the current device is the same pointer value through which that memory may be accessed from the peer device.

5.8.7 Exceptions, Disjoint Addressing

Not all memory may be accessed on devices through the same pointer value through which they are accessed on the host. These exceptions are host memory registered using cudaHostRegister() and host memory allocated using the flag cudaHostAllocWriteCombined. For these exceptions, there exists a distinct host and device address for the memory. The device address is guaranteed to not overlap any valid host pointer range and is guaranteed to have the same value across all devices that support unified addressing.

This device address may be queried using cudaHostGetDevicePointer() when a device using unified addressing is current. Either the host or the unified device pointer value may be used to refer to this memory in cudaMemcpy() and similar functions using the cudaMemcpyDefault memory direction.

5.8.8 Function Documentation

5.8.8.1 cudaError_t cudaPointerGetAttributes (struct cudaPointerAttributes * *attributes*, const void * *ptr*)

Returns in `*attributes` the attributes of the pointer `ptr`.

The cudaPointerAttributes structure is defined as:

```
struct cudaPointerAttributes {
    enum cudaMemoryType memoryType;
    int device;
    void *devicePointer;
    void *hostPointer;
}
```

In this structure, the individual fields mean

- memoryType identifies the physical location of the memory associated with pointer `ptr`. It can be cudaMemoryTypeHost for host memory or cudaMemoryTypeDevice for device memory.
- device is the device against which `ptr` was allocated. If `ptr` has memory type cudaMemoryTypeDevice then this identifies the device on which the memory referred to by `ptr` physically resides. If `ptr` has memory type cudaMemoryTypeHost then this identifies the device which was current when the allocation was made (and if that device is deinitialized then this allocation will vanish with that device's state).
- devicePointer is the device pointer alias through which the memory referred to by `ptr` may be accessed on the current device. If the memory referred to by `ptr` cannot be accessed directly by the current device then this is NULL.
- hostPointer is the host pointer alias through which the memory referred to by `ptr` may be accessed on the host. If the memory referred to by `ptr` cannot be accessed directly by the host then this is NULL.

Parameters:

attributes - Attributes for the specified pointer

ptr - Pointer to get attributes for

Returns:

cudaSuccess, cudaErrorInvalidDevice

See also:

cudaGetDeviceCount, cudaGetDevice, cudaSetDevice, cudaChooseDevice

5.9 Peer Device Memory Access

Functions

- cudaError_t cudaDeviceCanAccessPeer (int *canAccessPeer, int device, int peerDevice)

  Queries if a device may directly access a peer device's memory.

- cudaError_t cudaDeviceDisablePeerAccess (int peerDevice)

  Disables direct access to memory allocations on a peer device.

- cudaError_t cudaDeviceEnablePeerAccess (int peerDevice, unsigned int flags)

  Enables direct access to memory allocations on a peer device.

5.9.1 Detailed Description

This section describes the peer device memory access functions of the CUDA runtime application programming interface.

5.9.2 Function Documentation

5.9.2.1 cudaError_t cudaDeviceCanAccessPeer (int * *canAccessPeer*, int *device*, int *peerDevice*)

Returns in `*canAccessPeer` a value of 1 if device `device` is capable of directly accessing memory from `peerDevice` and 0 otherwise. If direct access of `peerDevice` from `device` is possible, then access may be enabled by calling cudaDeviceEnablePeerAccess().

Parameters:

canAccessPeer - Returned access capability

device - Device from which allocations on `peerDevice` are to be directly accessed.

peerDevice - Device on which the allocations to be directly accessed by `device` reside.

Returns:

cudaSuccess, cudaErrorInvalidDevice

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceEnablePeerAccess, cudaDeviceDisablePeerAccess

5.9.2.2 cudaError_t cudaDeviceDisablePeerAccess (int *peerDevice*)

Returns cudaErrorPeerAccessNotEnabled if direct access to memory on `peerDevice` has not yet been enabled from the current device.

Parameters:

peerDevice - Peer device to disable direct access to

Returns:

cudaSuccess, cudaErrorPeerAccessNotEnabled, cudaErrorInvalidDevice

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceCanAccessPeer, cudaDeviceEnablePeerAccess

5.9.2.3 cudaError_t cudaDeviceEnablePeerAccess (int *peerDevice*, unsigned int *flags*)

On success, all allocations from `peerDevice` will immediately be accessible by the current device. They will remain accessible until access is explicitly disabled using cudaDeviceDisablePeerAccess() or either device is reset using cudaDeviceReset().

Note that access granted by this call is unidirectional and that in order to access memory on the current device from `peerDevice`, a separate symmetric call to cudaDeviceEnablePeerAccess() is required.

Peer access is not supported in 32 bit applications.

Returns cudaErrorInvalidDevice if cudaDeviceCanAccessPeer() indicates that the current device cannot directly access memory from `peerDevice`.

Returns cudaErrorPeerAccessAlreadyEnabled if direct access of `peerDevice` from the current device has already been enabled.

Returns cudaErrorInvalidValue if `flags` is not 0.

Parameters:

peerDevice - Peer device to enable direct access to from the current device

flags - Reserved for future use and must be set to 0

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorPeerAccessAlreadyEnabled, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaDeviceCanAccessPeer, cudaDeviceDisablePeerAccess

5.10 OpenGL Interoperability

Enumerations

- enum cudaGLDeviceList {

  cudaGLDeviceListAll = 1,

  cudaGLDeviceListCurrentFrame = 2,

  cudaGLDeviceListNextFrame = 3 }

Functions

- cudaError_t cudaGLGetDevices (unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, enum cudaGLDeviceList deviceList)

  Gets the CUDA devices associated with the current OpenGL context.

- cudaError_t cudaGraphicsGLRegisterBuffer (struct cudaGraphicsResource **resource, GLuint buffer, unsigned int flags)

  Registers an OpenGL buffer object.

- cudaError_t cudaGraphicsGLRegisterImage (struct cudaGraphicsResource **resource, GLuint image, GLenum target, unsigned int flags)

  Register an OpenGL texture or renderbuffer object.

- cudaError_t cudaWGLGetDevice (int *device, HGPUNV hGpu)

  Gets the CUDA device associated with hGpu.

5.10.1 Detailed Description

This section describes the OpenGL interoperability functions of the CUDA runtime application programming interface. Note that mapping of OpenGL resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interopability.

5.10.2 Enumeration Type Documentation

5.10.2.1 enum cudaGLDeviceList

CUDA devices corresponding to the current OpenGL context

Enumerator:

cudaGLDeviceListAll The CUDA devices for all GPUs used by the current OpenGL context

cudaGLDeviceListCurrentFrame The CUDA devices for the GPUs used by the current OpenGL context in its currently rendering frame

cudaGLDeviceListNextFrame The CUDA devices for the GPUs to be used by the current OpenGL context in the next frame

5.10.3 Function Documentation

5.10.3.1 cudaError_t cudaGLGetDevices (unsigned int * *pCudaDeviceCount*, int * *pCudaDevices*, unsigned int *cudaDeviceCount*, enum cudaGLDeviceList *deviceList*)

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the current OpenGL context. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the CUDA-compatible devices corresponding to the current OpenGL context. If any of the GPUs being used by the current OpenGL context are not CUDA capable then the call will return cudaErrorNoDevice.

Parameters:

pCudaDeviceCount - Returned number of CUDA devices corresponding to the current OpenGL context

pCudaDevices - Returned CUDA devices corresponding to the current OpenGL context

cudaDeviceCount - The size of the output device array `pCudaDevices`

deviceList - The set of devices to return. This set may be cudaGLDeviceListAll for all devices, cudaGLDeviceListCurrentFrame for the devices used to render the current frame (in SLI), or cudaGLDeviceListNextFrame for the devices used to render the next frame (in SLI).

Returns:

cudaSuccess, cudaErrorNoDevice, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer

5.10.3.2 cudaError_t cudaGraphicsGLRegisterBuffer (struct cudaGraphicsResource ** *resource*, GLuint *buffer*, unsigned int *flags*)

Registers the buffer object specified by `buffer` for access by CUDA. A handle to the registered object is returned as `resource`. The register flags `flags` specify the intended usage, as follows:

- cudaGraphicsRegisterFlagsNone: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- cudaGraphicsRegisterFlagsReadOnly: Specifies that CUDA will not write to this resource.
- cudaGraphicsRegisterFlagsWriteDiscard: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

Parameters:

resource - Pointer to the returned object handle

buffer - name of buffer object to be registered

flags - Register flags

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsResourceGetMappedPointer

5.10.3.3 cudaError_t cudaGraphicsGLRegisterImage (struct cudaGraphicsResource ** *resource*, GLuint *image*, GLenum *target*, unsigned int *flags*)

Registers the texture or renderbuffer object specified by `image` for access by CUDA. A handle to the registered object is returned as `resource`.

`target` must match the type of the object, and must be one of GL_TEXTURE_2D, GL_TEXTURE_RECTANGLE, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_3D, GL_TEXTURE_2D_ARRAY, or GL_RENDERBUFFER.

The register flags `flags` specify the intended usage, as follows:

- cudaGraphicsRegisterFlagsNone: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- cudaGraphicsRegisterFlagsReadOnly: Specifies that CUDA will not write to this resource.
- cudaGraphicsRegisterFlagsWriteDiscard: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.
- cudaGraphicsRegisterFlagsSurfaceLoadStore: Specifies that CUDA will bind this resource to a surface reference.
- cudaGraphicsRegisterFlagsTextureGather: Specifies that CUDA will perform texture gather operations on this resource.

The following image formats are supported. For brevity's sake, the list is abbreviated. For ex., {GL_R, GL_RG} X {8, 16} would expand to the following 4 formats {GL_R8, GL_R16, GL_RG8, GL_RG16} :

- GL_RED, GL_RG, GL_RGBA, GL_LUMINANCE, GL_ALPHA, GL_LUMINANCE_ALPHA, GL_INTENSITY
- {GL_R, GL_RG, GL_RGBA} X {8, 16, 16F, 32F, 8UI, 16UI, 32UI, 8I, 16I, 32I}
- {GL_LUMINANCE, GL_ALPHA, GL_LUMINANCE_ALPHA, GL_INTENSITY} X {8, 16, 16F_ARB, 32F_ARB, 8UI_EXT, 16UI_EXT, 32UI_EXT, 8I_EXT, 16I_EXT, 32I_EXT}

The following image classes are currently disallowed:

- Textures with borders
- Multisampled renderbuffers

Parameters:

resource - Pointer to the returned object handle

image - name of texture or renderbuffer object to be registered

target - Identifies the type of object specified by `image`

flags - Register flags

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray

5.10.3.4 cudaError_t cudaWGLGetDevice (int * *device*, HGPUNV *hGpu*)

Returns the CUDA device associated with a hGpu, if applicable.

Parameters:

device - Returns the device associated with hGpu, or -1 if hGpu is not a compute device.

hGpu - Handle to a GPU, as queried via WGL_NV_gpu_affinity

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

WGL_NV_gpu_affinity

5.11 OpenGL Interoperability [DEPRECATED]

Enumerations

- enum cudaGLMapFlags {
  cudaGLMapFlagsNone = 0,
  cudaGLMapFlagsReadOnly = 1,
  cudaGLMapFlagsWriteDiscard = 2 }

Functions

- cudaError_t cudaGLMapBufferObject (void **devPtr, GLuint bufObj)
  Maps a buffer object for access by CUDA.
- cudaError_t cudaGLMapBufferObjectAsync (void **devPtr, GLuint bufObj, cudaStream_t stream)
  Maps a buffer object for access by CUDA.
- cudaError_t cudaGLRegisterBufferObject (GLuint bufObj)
  Registers a buffer object for access by CUDA.
- cudaError_t cudaGLSetBufferObjectMapFlags (GLuint bufObj, unsigned int flags)
  Set usage flags for mapping an OpenGL buffer.
- cudaError_t cudaGLSetGLDevice (int device)
  Sets a CUDA device to use OpenGL interoperability.
- cudaError_t cudaGLUnmapBufferObject (GLuint bufObj)
  Unmaps a buffer object for access by CUDA.
- cudaError_t cudaGLUnmapBufferObjectAsync (GLuint bufObj, cudaStream_t stream)
  Unmaps a buffer object for access by CUDA.
- cudaError_t cudaGLUnregisterBufferObject (GLuint bufObj)
  Unregisters a buffer object for access by CUDA.

5.11.1 Detailed Description

This section describes deprecated OpenGL interoperability functionality.

5.11.2 Enumeration Type Documentation

5.11.2.1 enum cudaGLMapFlags

CUDA GL Map Flags

Enumerator:

cudaGLMapFlagsNone Default; Assume resource can be read/written

cudaGLMapFlagsReadOnly CUDA kernels will not write to this resource

cudaGLMapFlagsWriteDiscard CUDA kernels will only write to and will not read from this resource

5.11.3 Function Documentation

5.11.3.1 cudaError_t cudaGLMapBufferObject (void ** *devPtr*, GLuint *bufObj*)

Deprecated

This function is deprecated as of CUDA 3.0.

Maps the buffer object of ID `bufObj` into the address space of CUDA and returns in `*devPtr` the base pointer of the resulting mapping. The buffer must have previously been registered by calling cudaGLRegisterBufferObject(). While a buffer is mapped by CUDA, any OpenGL operation which references the buffer will result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

All streams in the current thread are synchronized with the current GL context.

Parameters:

devPtr - Returned device pointer to CUDA object

bufObj - Buffer object ID to map

Returns:

cudaSuccess, cudaErrorMapBufferObjectFailed

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsMapResources

5.11.3.2 cudaError_t cudaGLMapBufferObjectAsync (void ** *devPtr*, GLuint *bufObj*, cudaStream_t *stream*)

Deprecated

This function is deprecated as of CUDA 3.0.

Maps the buffer object of ID `bufObj` into the address space of CUDA and returns in `*devPtr` the base pointer of the resulting mapping. The buffer must have previously been registered by calling cudaGLRegisterBufferObject(). While a buffer is mapped by CUDA, any OpenGL operation which references the buffer will result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Stream /p stream is synchronized with the current GL context.

Parameters:

devPtr - Returned device pointer to CUDA object

bufObj - Buffer object ID to map

stream - Stream to synchronize

Returns:

cudaSuccess, cudaErrorMapBufferObjectFailed

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsMapResources

5.11.3.3 cudaError_t cudaGLRegisterBufferObject (GLuint *bufObj*)

Deprecated

This function is deprecated as of CUDA 3.0.

Registers the buffer object of ID `bufObj` for access by CUDA. This function must be called before CUDA can map the buffer object. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Parameters:

bufObj - Buffer object ID to register

Returns:

cudaSuccess, cudaErrorInitializationError

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsGLRegisterBuffer

5.11.3.4 cudaError_t cudaGLSetBufferObjectMapFlags (GLuint *bufObj*, unsigned int *flags*)

Deprecated

This function is deprecated as of CUDA 3.0.

Set flags for mapping the OpenGL buffer `bufObj`

Changes to flags will take effect the next time `bufObj` is mapped. The `flags` argument may be any of the following:

- cudaGLMapFlagsNone: Specifies no hints about how this buffer will be used. It is therefore assumed that this buffer will be read from and written to by CUDA kernels. This is the default value.
- cudaGLMapFlagsReadOnly: Specifies that CUDA kernels which access this buffer will not write to the buffer.

- cudaGLMapFlagsWriteDiscard: Specifies that CUDA kernels which access this buffer will not read from the buffer and will write over the entire contents of the buffer, so none of the data previously stored in the buffer will be preserved.

If `bufObj` has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned. If `bufObj` is presently mapped for access by CUDA, then cudaErrorUnknown is returned.

Parameters:

bufObj - Registered buffer object to set flags for

flags - Parameters for buffer mapping

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceSetMapFlags

5.11.3.5 cudaError_t cudaGLSetGLDevice (int *device*)

Deprecated

This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with an OpenGL context in order to achieve maximum interoperability performance.

Parameters:

device - Device to use for OpenGL interoperability

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorSetOnActiveProcess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsGLRegisterBuffer, cudaGraphicsGLRegisterImage

5.11.3.6 cudaError_t cudaGLUnmapBufferObject (GLuint *bufObj*)

Deprecated

This function is deprecated as of CUDA 3.0.

Unmaps the buffer object of ID `bufObj` for access by CUDA. When a buffer is unmapped, the base address returned by cudaGLMapBufferObject() is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

All streams in the current thread are synchronized with the current GL context.

Parameters:

bufObj - Buffer object to unmap

Returns:

cudaSuccess, cudaErrorInvalidDevicePointer, cudaErrorUnmapBufferObjectFailed

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnmapResources

5.11.3.7 cudaError_t cudaGLUnmapBufferObjectAsync (GLuint *bufObj*, cudaStream_t *stream*)

Deprecated

This function is deprecated as of CUDA 3.0.

Unmaps the buffer object of ID `bufObj` for access by CUDA. When a buffer is unmapped, the base address returned by cudaGLMapBufferObject() is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Stream /p stream is synchronized with the current GL context.

Parameters:

bufObj - Buffer object to unmap

stream - Stream to synchronize

Returns:

cudaSuccess, cudaErrorInvalidDevicePointer, cudaErrorUnmapBufferObjectFailed

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnmapResources

5.11.3.8 cudaError_t cudaGLUnregisterBufferObject (GLuint *bufObj*)

Deprecated

This function is deprecated as of CUDA 3.0.

Unregisters the buffer object of ID `bufObj` for access by CUDA and releases any CUDA resources associated with the buffer. Once a buffer is unregistered, it may no longer be mapped by CUDA. The GL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Parameters:

bufObj - Buffer object to unregister

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource

5.12 Direct3D 9 Interoperability

Enumerations

- enum cudaD3D9DeviceList {
  cudaD3D9DeviceListAll = 1,
  cudaD3D9DeviceListCurrentFrame = 2,
  cudaD3D9DeviceListNextFrame = 3 }

Functions

- cudaError_t cudaD3D9GetDevice (int *device, const char *pszAdapterName)
  Gets the device number for an adapter.
- cudaError_t cudaD3D9GetDevices (unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 *pD3D9Device, enum cudaD3D9DeviceList deviceList)
  Gets the CUDA devices corresponding to a Direct3D 9 device.
- cudaError_t cudaD3D9GetDirect3DDevice (IDirect3DDevice9 **ppD3D9Device)
  Gets the Direct3D device against which the current CUDA context was created.
- cudaError_t cudaD3D9SetDirect3DDevice (IDirect3DDevice9 *pD3D9Device, int device=-1)
  Sets the Direct3D 9 device to use for interoperability with a CUDA device.
- cudaError_t cudaGraphicsD3D9RegisterResource (struct cudaGraphicsResource **resource, IDirect3DResource9 *pD3DResource, unsigned int flags)
  Register a Direct3D 9 resource for access by CUDA.

5.12.1 Detailed Description

This section describes the Direct3D 9 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 9 resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interopability.

5.12.2 Enumeration Type Documentation

5.12.2.1 enum cudaD3D9DeviceList

CUDA devices corresponding to a D3D9 device

Enumerator:

cudaD3D9DeviceListAll The CUDA devices for all GPUs used by a D3D9 device

cudaD3D9DeviceListCurrentFrame The CUDA devices for the GPUs used by a D3D9 device in its currently rendering frame

cudaD3D9DeviceListNextFrame The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

5.12.3 Function Documentation

5.12.3.1 cudaError_t cudaD3D9GetDevice (int * *device*, const char * *pszAdapterName*)

Returns in `*device` the CUDA-compatible device corresponding to the adapter name `pszAdapterName` obtained from EnumDisplayDevices or IDirect3D9::GetAdapterIdentifier(). If no device on the adapter with name `pszAdapterName` is CUDA-compatible then the call will fail.

Parameters:

device - Returns the device corresponding to pszAdapterName

pszAdapterName - D3D9 adapter to get device for

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaD3D9SetDirect3DDevice, cudaGraphicsD3D9RegisterResource,

5.12.3.2 cudaError_t cudaD3D9GetDevices (unsigned int * *pCudaDeviceCount*, int * *pCudaDevices*, unsigned int *cudaDeviceCount*, IDirect3DDevice9 * *pD3D9Device*, enum cudaD3D9DeviceList *deviceList*)

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the Direct3D 9 device `pD3D9Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 9 device `pD3D9Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return cudaErrorNoDevice.

Parameters:

pCudaDeviceCount - Returned number of CUDA devices corresponding to `pD3D9Device`

pCudaDevices - Returned CUDA devices corresponding to `pD3D9Device`

cudaDeviceCount - The size of the output device array `pCudaDevices`

pD3D9Device - Direct3D 9 device to query for CUDA devices

deviceList - The set of devices to return. This set may be cudaD3D9DeviceListAll for all devices, cudaD3D9DeviceListCurrentFrame for the devices used to render the current frame (in SLI), or cudaD3D9DeviceListNextFrame for the devices used to render the next frame (in SLI).

Returns:

cudaSuccess, cudaErrorNoDevice, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer

5.12.3.3 cudaError_t cudaD3D9GetDirect3DDevice (IDirect3DDevice9 ** *ppD3D9Device*)

Returns in `*ppD3D9Device` the Direct3D device against which this CUDA context was created in cudaD3D9SetDirect3DDevice().

Parameters:

ppD3D9Device - Returns the Direct3D device for this thread

Returns:

cudaSuccess, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaD3D9SetDirect3DDevice

5.12.3.4 cudaError_t cudaD3D9SetDirect3DDevice (IDirect3DDevice9 * *pD3D9Device*, int *device* = -1)

Records `pD3D9Device` as the Direct3D 9 device to use for Direct3D 9 interoperability with the CUDA device `device` and sets `device` as the current device for the calling host thread.

If `device` has already been initialized then this call will fail with the error cudaErrorSetOnActiveProcess. In this case it is necessary to reset `device` using cudaDeviceReset() before Direct3D 9 interoperability on `device` may be enabled.

Successfully initializing CUDA interoperability with `pD3D9Device` will increase the internal reference count on `pD3D9Device`. This reference count will be decremented when `device` is reset using cudaDeviceReset().

Note that this function is never required for correct functionality. Use of this function will result in accelerated interoperability only when the operating system is Windows Vista or Windows 7, and the device `pD3DDdevice` is not an IDirect3DDevice9Ex. In all other cirumstances, this function is not necessary.

Parameters:

pD3D9Device - Direct3D device to use for this thread

device - The CUDA device to use. This device must be among the devices returned when querying cudaD3D9DeviceListAll from cudaD3D9GetDevices, may be set to -1 to automatically select an appropriate CUDA device.

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorSetOnActiveProcess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaD3D9GetDevice, cudaGraphicsD3D9RegisterResource, cudaDeviceReset

5.12.3.5 cudaError_t cudaGraphicsD3D9RegisterResource (struct cudaGraphicsResource ** *resource*, IDirect3DResource9 * *pD3DResource*, unsigned int *flags*)

Registers the Direct3D 9 resource `pD3DResource` for access by CUDA.

If this call is successful then the application will be able to map and unmap this resource until it is unregistered through cudaGraphicsUnregisterResource(). Also on success, this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through cudaGraphicsUnregisterResource().

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following.

- IDirect3DVertexBuffer9: may be accessed through a device pointer
- IDirect3DIndexBuffer9: may be accessed through a device pointer
- IDirect3DSurface9: may be accessed through an array. Only stand-alone objects of type IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- IDirect3DBaseTexture9: individual surfaces on this texture may be accessed through an array.

The `flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- cudaGraphicsRegisterFlagsNone: Specifies no hints about how this resource will be used.
- cudaGraphicsRegisterFlagsSurfaceLoadStore: Specifies that CUDA will bind this resource to a surface reference.
- cudaGraphicsRegisterFlagsTextureGather: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

A complete list of supported formats is as follows:

- D3DFMT_L8
- D3DFMT_L16
- D3DFMT_A8R8G8B8
- D3DFMT_X8R8G8B8
- D3DFMT_G16R16

- D3DFMT_A8B8G8R8
- D3DFMT_A8
- D3DFMT_A8L8
- D3DFMT_Q8W8V8U8
- D3DFMT_V16U16
- D3DFMT_A16B16G16R16F
- D3DFMT_A16B16G16R16
- D3DFMT_R32F
- D3DFMT_G16R16F
- D3DFMT_A32B32G32R32F
- D3DFMT_G32R32F
- D3DFMT_R16F

If `pD3DResource` is of incorrect type or is already registered, then cudaErrorInvalidResourceHandle is returned. If `pD3DResource` cannot be registered, then cudaErrorUnknown is returned.

Parameters:

resource - Pointer to returned resource handle

pD3DResource - Direct3D resource to register

flags - Parameters for resource registration

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaD3D9SetDirect3DDevice, cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer

5.13 Direct3D 9 Interoperability [DEPRECATED]

Enumerations

- enum cudaD3D9MapFlags {
  cudaD3D9MapFlagsNone = 0,
  cudaD3D9MapFlagsReadOnly = 1,
  cudaD3D9MapFlagsWriteDiscard = 2 }
- enum cudaD3D9RegisterFlags {
  cudaD3D9RegisterFlagsNone = 0,
  cudaD3D9RegisterFlagsArray = 1 }

Functions

- cudaError_t cudaD3D9MapResources (int count, IDirect3DResource9 **ppResources)
  Map Direct3D resources for access by CUDA.
- cudaError_t cudaD3D9RegisterResource (IDirect3DResource9 *pResource, unsigned int flags)
  Registers a Direct3D resource for access by CUDA.
- cudaError_t cudaD3D9ResourceGetMappedArray (cudaArray **ppArray, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
  Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.
- cudaError_t cudaD3D9ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
  Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.
- cudaError_t cudaD3D9ResourceGetMappedPointer (void **pPointer, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
  Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.
- cudaError_t cudaD3D9ResourceGetMappedSize (size_t *pSize, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
  Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.
- cudaError_t cudaD3D9ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
  Get the dimensions of a registered Direct3D surface.
- cudaError_t cudaD3D9ResourceSetMapFlags (IDirect3DResource9 *pResource, unsigned int flags)
  Set usage flags for mapping a Direct3D resource.
- cudaError_t cudaD3D9UnmapResources (int count, IDirect3DResource9 **ppResources)
  Unmap Direct3D resources for access by CUDA.
- cudaError_t cudaD3D9UnregisterResource (IDirect3DResource9 *pResource)
  Unregisters a Direct3D resource for access by CUDA.

5.13.1 Detailed Description

This section describes deprecated Direct3D 9 interoperability functions.

5.13.2 Enumeration Type Documentation

5.13.2.1 enum cudaD3D9MapFlags

CUDA D3D9 Map Flags

Enumerator:

cudaD3D9MapFlagsNone Default; Assume resource can be read/written

cudaD3D9MapFlagsReadOnly CUDA kernels will not write to this resource

cudaD3D9MapFlagsWriteDiscard CUDA kernels will only write to and will not read from this resource

5.13.2.2 enum cudaD3D9RegisterFlags

CUDA D3D9 Register Flags

Enumerator:

cudaD3D9RegisterFlagsNone Default; Resource can be accessed througa void∗

cudaD3D9RegisterFlagsArray Resource can be accessed through a CUarray∗

5.13.3 Function Documentation

5.13.3.1 cudaError_t cudaD3D9MapResources (int *count*, IDirect3DResource9 ∗∗ *ppResources*)

Deprecated

This function is deprecated as of CUDA 3.0.

Maps the `count` Direct3D resources in `ppResources` for access by CUDA.

The resources in `ppResources` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before cudaD3D9MapResources() will complete before any CUDA kernels issued after cudaD3D9MapResources() begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries then cudaErrorInvalidResourceHandle is returned. If any of `ppResources` are presently mapped for access by CUDA then cudaErrorUnknown is returned.

Parameters:

count - Number of resources to map for CUDA

ppResources - Resources to map for CUDA

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsMapResources

5.13.3.2 cudaError_t cudaD3D9RegisterResource (IDirect3DResource9 * *pResource*, unsigned int *flags*)

Deprecated

This function is deprecated as of CUDA 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cudaD3D9UnregisterResource(). Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through cudaD3D9UnregisterResource().

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following.

- IDirect3DVertexBuffer9: No notes.
- IDirect3DIndexBuffer9: No notes.
- IDirect3DSurface9: Only stand-alone objects of type IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- IDirect3DBaseTexture9: When a texture is registered, all surfaces associated with all mipmap levels of all faces of the texture will be accessible to CUDA.

The `flags` argument specifies the mechanism through which CUDA will access the Direct3D resource. The following value is allowed:

- cudaD3D9RegisterFlagsNone: Specifies that CUDA will access this resource through a `void*`. The pointer, size, and pitch for each subresource of this resource may be queried through cudaD3D9ResourceGetMappedPointer(), cudaD3D9ResourceGetMappedSize(), and cudaD3D9ResourceGetMappedPitch() respectively. This option is valid for all resource types.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations:

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Any resources allocated in D3DPOOL_SYSTEMMEM or D3DPOOL_MANAGED may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context, then cudaErrorInvalidDevice is returned. If pResource is of incorrect type (e.g, is a non-stand-alone IDirect3DSurface9) or is already registered, then cudaErrorInvalidResourceHandle is returned. If pResource cannot be registered then cudaErrorUnknown is returned.

Parameters:

pResource - Resource to register

flags - Parameters for resource registration

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsD3D9RegisterResource

5.13.3.3 cudaError_t cudaD3D9ResourceGetMappedArray (cudaArray ** *ppArray*, IDirect3DResource9 * *pResource*, unsigned int *face*, unsigned int *level*)

Deprecated

This function is deprecated as of CUDA 3.0.

Returns in *pArray an array through which the subresource of the mapped Direct3D resource pResource, which corresponds to face and level may be accessed. The value set in pArray may change every time that pResource is mapped.

If pResource is not registered then cudaErrorInvalidResourceHandle is returned. If pResource was not registered with usage flags cudaD3D9RegisterFlagsArray, then cudaErrorInvalidResourceHandle is returned. If pResource is not mapped, then cudaErrorUnknown is returned.

For usage requirements of face and level parameters, see cudaD3D9ResourceGetMappedPointer().

Parameters:

ppArray - Returned array corresponding to subresource

pResource - Mapped resource to access

face - Face of resource to access

level - Level of resource to access

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsSubResourceGetMappedArray

5.13.3.4 cudaError_t cudaD3D9ResourceGetMappedPitch (size_t * *pPitch*, size_t * *pPitchSlice*, IDirect3DResource9 * *pResource*, unsigned int *face*, unsigned int *level*)

Deprecated

This function is deprecated as of CUDA 3.0.

Returns in `*pPitch` and `*pPitchSlice` the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level`. The values set in `pPitch` and `pPitchSlice` may change every time that `pResource` is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position **x**, **y** from the base pointer of the surface is:

y * **pitch** + (**bytes per pixel**) * **x**

For a 3D surface, the byte offset of the sample at position **x**, **y**, **z** from the base pointer of the surface is:

z* **slicePitch** + **y** * **pitch** + (**bytes per pixel**) * **x**

Both parameters `pPitch` and `pPitchSlice` are optional and may be set to NULL.

If `pResource` is not of type IDirect3DBaseTexture9 or one of its sub-types or if `pResource` has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned. If `pResource` was not registered with usage flags cudaD3D9RegisterFlagsNone, then cudaErrorInvalidResourceHandle is returned. If `pResource` is not mapped for access by CUDA then cudaErrorUnknown is returned.

For usage requirements of `face` and `level` parameters, see cudaD3D9ResourceGetMappedPointer().

Parameters:

pPitch - Returned pitch of subresource

pPitchSlice - Returned Z-slice pitch of subresource

pResource - Mapped resource to access

face - Face of resource to access

level - Level of resource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceGetMappedPointer

5.13.3.5 cudaError_t cudaD3D9ResourceGetMappedPointer (void ** *pPointer*, IDirect3DResource9 * *pResource*, unsigned int *face*, unsigned int *level*)

Deprecated

This function is deprecated as of CUDA 3.0.

Returns in *pPointer the base pointer of the subresource of the mapped Direct3D resource pResource, which corresponds to face and level. The value set in pPointer may change every time that pResource is mapped.

If pResource is not registered, then cudaErrorInvalidResourceHandle is returned. If pResource was not registered with usage flags cudaD3D9RegisterFlagsNone, then cudaErrorInvalidResourceHandle is returned. If pResource is not mapped, then cudaErrorUnknown is returned.

If pResource is of type IDirect3DCubeTexture9, then face must one of the values enumerated by type D3DCUBEMAP_FACES. For all other types, face must be 0. If face is invalid, then cudaErrorInvalidValue is returned.

If pResource is of type IDirect3DBaseTexture9, then level must correspond to a valid mipmap level. Only mipmap level 0 is supported for now. For all other types level must be 0. If level is invalid, then cudaErrorInvalidValue is returned.

Parameters:

pPointer - Returned pointer corresponding to subresource

pResource - Mapped resource to access

face - Face of resource to access

level - Level of resource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceGetMappedPointer

5.13.3.6 cudaError_t cudaD3D9ResourceGetMappedSize (size_t * *pSize*, IDirect3DResource9 * *pResource*, unsigned int *face*, unsigned int *level*)

Deprecated

This function is deprecated as of CUDA 3.0.

Returns in *pSize the size of the subresource of the mapped Direct3D resource pResource, which corresponds to face and level. The value set in pSize may change every time that pResource is mapped.

If pResource has not been registered for use with CUDA then cudaErrorInvalidResourceHandle is returned. If pResource was not registered with usage flags cudaD3D9RegisterFlagsNone, then cudaErrorInvalidResourceHandle is returned. If pResource is not mapped for access by CUDA then cudaErrorUnknown is returned.

For usage requirements of face and level parameters, see cudaD3D9ResourceGetMappedPointer().

Parameters:

pSize - Returned size of subresource

pResource - Mapped resource to access

face - Face of resource to access

level - Level of resource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceGetMappedPointer

5.13.3.7 cudaError_t cudaD3D9ResourceGetSurfaceDimensions (size_t * *pWidth*, size_t * *pHeight*, size_t * *pDepth*, IDirect3DResource9 * *pResource*, unsigned int *face*, unsigned int *level*)

Deprecated

This function is deprecated as of CUDA 3.0.

Returns in `*pWidth`, `*pHeight`, and `*pDepth` the dimensions of the subresource of the mapped Direct3D resource `pResource` which corresponds to `face` and `level`.

Since anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters `pWidth`, `pHeight`, and `pDepth` are optional. For 2D surfaces, the value returned in `*pDepth` will be 0.

If `pResource` is not of type IDirect3DBaseTexture9 or IDirect3DSurface9 or if `pResource` has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned.

For usage requirements of `face` and `level` parameters, see cudaD3D9ResourceGetMappedPointer.

Parameters:

pWidth - Returned width of surface

pHeight - Returned height of surface

pDepth - Returned depth of surface

pResource - Registered resource to access

face - Face of resource to access

level - Level of resource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsSubResourceGetMappedArray

5.13.3.8 cudaError_t cudaD3D9ResourceSetMapFlags (IDirect3DResource9 * *pResource*, unsigned int *flags*)

Deprecated

This function is deprecated as of CUDA 3.0.

Set flags for mapping the Direct3D resource `pResource`.

Changes to flags will take effect the next time `pResource` is mapped. The `flags` argument may be any of the following:

- cudaD3D9MapFlagsNone: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- cudaD3D9MapFlagsReadOnly: Specifies that CUDA kernels which access this resource will not write to this resource.
- cudaD3D9MapFlagsWriteDiscard: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned. If `pResource` is presently mapped for access by CUDA, then cudaErrorUnknown is returned.

Parameters:

pResource - Registered resource to set flags for

flags - Parameters for resource mapping

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaInteropResourceSetMapFlags

5.13.3.9 cudaError_t cudaD3D9UnmapResources (int *count*, IDirect3DResource9 ** *ppResources*)

Deprecated

This function is deprecated as of CUDA 3.0.

Unmaps the `count` Direct3D resources in `ppResources`.

This function provides the synchronization guarantee that any CUDA kernels issued before cudaD3D9UnmapResources() will complete before any Direct3D calls issued after cudaD3D9UnmapResources() begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then cudaErrorInvalidResourceHandle is returned. If any of `ppResources` are not presently mapped for access by CUDA then cudaErrorUnknown is returned.

Parameters:

count - Number of resources to unmap for CUDA

ppResources - Resources to unmap for CUDA

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnmapResources

5.13.3.10 cudaError_t cudaD3D9UnregisterResource (IDirect3DResource9 * *pResource*)

Deprecated

This function is deprecated as of CUDA 3.0.

Unregisters the Direct3D resource `pResource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then cudaErrorInvalidResourceHandle is returned.

Parameters:

pResource - Resource to unregister

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource

5.14 Direct3D 10 Interoperability

Enumerations

- enum cudaD3D10DeviceList {
  cudaD3D10DeviceListAll = 1,
  cudaD3D10DeviceListCurrentFrame = 2,
  cudaD3D10DeviceListNextFrame = 3 }

Functions

- cudaError_t cudaD3D10GetDevice (int *device, IDXGIAdapter *pAdapter)
  Gets the device number for an adapter.
- cudaError_t cudaD3D10GetDevices (unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device *pD3D10Device, enum cudaD3D10DeviceList deviceList)
  Gets the CUDA devices corresponding to a Direct3D 10 device.
- cudaError_t cudaGraphicsD3D10RegisterResource (struct cudaGraphicsResource **resource, ID3D10Resource *pD3DResource, unsigned int flags)
  Registers a Direct3D 10 resource for access by CUDA.

5.14.1 Detailed Description

This section describes the Direct3D 10 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 10 resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interopability.

5.14.2 Enumeration Type Documentation

5.14.2.1 enum cudaD3D10DeviceList

CUDA devices corresponding to a D3D10 device

Enumerator:

cudaD3D10DeviceListAll The CUDA devices for all GPUs used by a D3D10 device

cudaD3D10DeviceListCurrentFrame The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame

cudaD3D10DeviceListNextFrame The CUDA devices for the GPUs to be used by a D3D10 device in the next frame

5.14.3 Function Documentation

5.14.3.1 cudaError_t cudaD3D10GetDevice (int * *device*, IDXGIAdapter * *pAdapter*)

Returns in `*device` the CUDA-compatible device corresponding to the adapter `pAdapter` obtained from IDXGIFactory::EnumAdapters. This call will succeed only if a device on adapter `pAdapter` is CUDA-compatible.

Parameters:

device - Returns the device corresponding to pAdapter

pAdapter - D3D10 adapter to get device for

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsD3D10RegisterResource,

5.14.3.2 cudaError_t cudaD3D10GetDevices (unsigned int * *pCudaDeviceCount*, int * *pCudaDevices*, unsigned int *cudaDeviceCount*, ID3D10Device * *pD3D10Device*, enum cudaD3D10DeviceList *deviceList*)

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the Direct3D 10 device `pD3D10Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 10 device `pD3D10Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return cudaErrorNoDevice.

Parameters:

pCudaDeviceCount - Returned number of CUDA devices corresponding to `pD3D10Device`

pCudaDevices - Returned CUDA devices corresponding to `pD3D10Device`

cudaDeviceCount - The size of the output device array `pCudaDevices`

pD3D10Device - Direct3D 10 device to query for CUDA devices

deviceList - The set of devices to return. This set may be cudaD3D10DeviceListAll for all devices, cudaD3D10DeviceListCurrentFrame for the devices used to render the current frame (in SLI), or cudaD3D10DeviceListNextFrame for the devices used to render the next frame (in SLI).

Returns:

cudaSuccess, cudaErrorNoDevice, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer

5.14.3.3 cudaError_t cudaGraphicsD3D10RegisterResource (struct cudaGraphicsResource ** *resource*, ID3D10Resource * *pD3DResource*, unsigned int *flags*)

Registers the Direct3D 10 resource `pD3DResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cudaGraphicsUnregisterResource(). Also on success, this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through cudaGraphicsUnregisterResource().

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following.

- ID3D10Buffer: may be accessed via a device pointer
- ID3D10Texture1D: individual subresources of the texture may be accessed via arrays
- ID3D10Texture2D: individual subresources of the texture may be accessed via arrays
- ID3D10Texture3D: individual subresources of the texture may be accessed via arrays

The `flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- cudaGraphicsRegisterFlagsNone: Specifies no hints about how this resource will be used.
- cudaGraphicsRegisterFlagsSurfaceLoadStore: Specifies that CUDA will bind this resource to a surface reference.
- cudaGraphicsRegisterFlagsTextureGather: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

A complete list of supported DXGI formats is as follows. For compactness the notation A_{B,C,D} represents A_B, A_C, and A_D.

- DXGI_FORMAT_A8_UNORM
- DXGI_FORMAT_B8G8R8A8_UNORM
- DXGI_FORMAT_B8G8R8X8_UNORM
- DXGI_FORMAT_R16_FLOAT
- DXGI_FORMAT_R16G16B16A16_{FLOAT,SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R16G16_{FLOAT,SINT,SNORM,UINT,UNORM}

- DXGI_FORMAT_R16_{SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R32_FLOAT
- DXGI_FORMAT_R32G32B32A32_{FLOAT,SINT,UINT}
- DXGI_FORMAT_R32G32_{FLOAT,SINT,UINT}
- DXGI_FORMAT_R32_{SINT,UINT}
- DXGI_FORMAT_R8G8B8A8_{SINT,SNORM,UINT,UNORM,UNORM_SRGB}
- DXGI_FORMAT_R8G8_{SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R8_{SINT,SNORM,UINT,UNORM}

If `pD3DResource` is of incorrect type or is already registered, then cudaErrorInvalidResourceHandle is returned. If `pD3DResource` cannot be registered, then cudaErrorUnknown is returned.

Parameters:

resource - Pointer to returned resource handle

pD3DResource - Direct3D resource to register

flags - Parameters for resource registration

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer

5.15 Direct3D 10 Interoperability [DEPRECATED]

Enumerations

- enum cudaD3D10MapFlags {
  cudaD3D10MapFlagsNone = 0,
  cudaD3D10MapFlagsReadOnly = 1,
  cudaD3D10MapFlagsWriteDiscard = 2 }
- enum cudaD3D10RegisterFlags {
  cudaD3D10RegisterFlagsNone = 0,
  cudaD3D10RegisterFlagsArray = 1 }

Functions

- cudaError_t cudaD3D10GetDirect3DDevice (ID3D10Device **ppD3D10Device)

  Gets the Direct3D device against which the current CUDA context was created.

- cudaError_t cudaD3D10MapResources (int count, ID3D10Resource **ppResources)

  Maps Direct3D Resources for access by CUDA.

- cudaError_t cudaD3D10RegisterResource (ID3D10Resource *pResource, unsigned int flags)

  Registers a Direct3D 10 resource for access by CUDA.

- cudaError_t cudaD3D10ResourceGetMappedArray (cudaArray **ppArray, ID3D10Resource *pResource, unsigned int subResource)

  Gets an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- cudaError_t cudaD3D10ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, ID3D10Resource *pResource, unsigned int subResource)

  Gets the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

- cudaError_t cudaD3D10ResourceGetMappedPointer (void **pPointer, ID3D10Resource *pResource, unsigned int subResource)

  Gets a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- cudaError_t cudaD3D10ResourceGetMappedSize (size_t *pSize, ID3D10Resource *pResource, unsigned int subResource)

  Gets the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

- cudaError_t cudaD3D10ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, ID3D10Resource *pResource, unsigned int subResource)

  Gets the dimensions of a registered Direct3D surface.

- cudaError_t cudaD3D10ResourceSetMapFlags (ID3D10Resource *pResource, unsigned int flags)

  Set usage flags for mapping a Direct3D resource.

- cudaError_t cudaD3D10SetDirect3DDevice (ID3D10Device *pD3D10Device, int device=-1)

Sets the Direct3D 10 device to use for interoperability with a CUDA device.

- cudaError_t cudaD3D10UnmapResources (int count, ID3D10Resource ∗∗ppResources)
  Unmaps Direct3D resources.
- cudaError_t cudaD3D10UnregisterResource (ID3D10Resource ∗pResource)
  Unregisters a Direct3D resource.

5.15.1 Detailed Description

This section describes deprecated Direct3D 10 interoperability functions.

5.15.2 Enumeration Type Documentation

5.15.2.1 enum cudaD3D10MapFlags

CUDA D3D10 Map Flags

Enumerator:

cudaD3D10MapFlagsNone Default; Assume resource can be read/written
cudaD3D10MapFlagsReadOnly CUDA kernels will not write to this resource
cudaD3D10MapFlagsWriteDiscard CUDA kernels will only write to and will not read from this resource

5.15.2.2 enum cudaD3D10RegisterFlags

CUDA D3D10 Register Flags

Enumerator:

cudaD3D10RegisterFlagsNone Default; Resource can be accessed through a void∗
cudaD3D10RegisterFlagsArray Resource can be accessed through a CUarray∗

5.15.3 Function Documentation

5.15.3.1 cudaError_t cudaD3D10GetDirect3DDevice (ID3D10Device ∗∗ *ppD3D10Device*)

Deprecated

This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D10 device in order to achieve maximum interoperability performance.

Parameters:

ppD3D10Device - Returns the Direct3D device for this thread

Returns:

cudaSuccess, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaD3D10SetDirect3DDevice

5.15.3.2 cudaError_t cudaD3D10MapResources (int *count*, ID3D10Resource ** *ppResources*)

Deprecated

This function is deprecated as of CUDA 3.0.

Maps the `count` Direct3D resources in `ppResources` for access by CUDA.

The resources in `ppResources` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before cudaD3D10MapResources() will complete before any CUDA kernels issued after cudaD3D10MapResources() begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries then cudaErrorInvalidResourceHandle is returned. If any of `ppResources` are presently mapped for access by CUDA then cudaErrorUnknown is returned.

Parameters:

count - Number of resources to map for CUDA

ppResources - Resources to map for CUDA

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsMapResources

5.15.3.3 cudaError_t cudaD3D10RegisterResource (ID3D10Resource * *pResource*, unsigned int *flags*)

Deprecated

This function is deprecated as of CUDA 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cudaD3D10UnregisterResource(). Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through cudaD3D10UnregisterResource().

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following:

- ID3D10Buffer: Cannot be used with `flags` set to `cudaD3D10RegisterFlagsArray`.
- ID3D10Texture1D: No restrictions.
- ID3D10Texture2D: No restrictions.
- ID3D10Texture3D: No restrictions.

The `flags` argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- cudaD3D10RegisterFlagsNone: Specifies that CUDA will access this resource through a `void*`. The pointer, size, and pitch for each subresource of this resource may be queried through cudaD3D10ResourceGetMappedPointer(), cudaD3D10ResourceGetMappedSize(), and cudaD3D10ResourceGetMappedPitch() respectively. This option is valid for all resource types.
- cudaD3D10RegisterFlagsArray: Specifies that CUDA will access this resource through a `CUarray` queried on a sub-resource basis through cudaD3D10ResourceGetMappedArray(). This option is only valid for resources of type ID3D10Texture1D, ID3D10Texture2D, and ID3D10Texture3D.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context then cudaErrorInvalidDevice is returned. If `pResource` is of incorrect type or is already registered then cudaErrorInvalidResourceHandle is returned. If `pResource` cannot be registered then cudaErrorUnknown is returned.

Parameters:

pResource - Resource to register

flags - Parameters for resource registration

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsD3D10RegisterResource

5.15.3.4 cudaError_t cudaD3D10ResourceGetMappedArray (cudaArray ** *ppArray*, ID3D10Resource * *pResource*, unsigned int *subResource*)

Deprecated

This function is deprecated as of CUDA 3.0.

Returns in *ppArray an array through which the subresource of the mapped Direct3D resource pResource which corresponds to subResource may be accessed. The value set in ppArray may change every time that pResource is mapped.

If pResource is not registered, then cudaErrorInvalidResourceHandle is returned. If pResource was not registered with usage flags cudaD3D10RegisterFlagsArray, then cudaErrorInvalidResourceHandle is returned. If pResource is not mapped then cudaErrorUnknown is returned.

For usage requirements of the subResource parameter, see cudaD3D10ResourceGetMappedPointer().

Parameters:

ppArray - Returned array corresponding to subresource

pResource - Mapped resource to access

subResource - Subresource of pResource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsSubResourceGetMappedArray

5.15.3.5 cudaError_t cudaD3D10ResourceGetMappedPitch (size_t * *pPitch*, size_t * *pPitchSlice*, ID3D10Resource * *pResource*, unsigned int *subResource*)

Deprecated

This function is deprecated as of CUDA 3.0.

Returns in *pPitch and *pPitchSlice the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource pResource, which corresponds to subResource. The values set in pPitch and pPitchSlice may change every time that pResource is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position **x**, **y** from the base pointer of the surface is:

y * **pitch** + (**bytes per pixel**) * **x**

For a 3D surface, the byte offset of the sample at position **x**, **y**, **z** from the base pointer of the surface is:

z* **slicePitch** + **y** * **pitch** + (**bytes per pixel**) * **x**

Both parameters pPitch and pPitchSlice are optional and may be set to NULL.

If pResource is not of type ID3D10Texture1D, ID3D10Texture2D, or ID3D10Texture3D, or if pResource has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned. If pResource was

not registered with usage flags cudaD3D10RegisterFlagsNone, then cudaErrorInvalidResourceHandle is returned. If pResource is not mapped for access by CUDA then cudaErrorUnknown is returned.

For usage requirements of the subResource parameter see cudaD3D10ResourceGetMappedPointer().

Parameters:

pPitch - Returned pitch of subresource

pPitchSlice - Returned Z-slice pitch of subresource

pResource - Mapped resource to access

subResource - Subresource of pResource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsSubResourceGetMappedArray

5.15.3.6 cudaError_t cudaD3D10ResourceGetMappedPointer (void ** *pPointer*, ID3D10Resource * *pResource*, unsigned int *subResource*)

Deprecated

This function is deprecated as of CUDA 3.0.

Returns in *pPointer the base pointer of the subresource of the mapped Direct3D resource pResource which corresponds to subResource. The value set in pPointer may change every time that pResource is mapped.

If pResource is not registered, then cudaErrorInvalidResourceHandle is returned. If pResource was not registered with usage flags cudaD3D9RegisterFlagsNone, then cudaErrorInvalidResourceHandle is returned. If pResource is not mapped then cudaErrorUnknown is returned.

If pResource is of type ID3D10Buffer then subResource must be 0. If pResource is of any other type, then the value of subResource must come from the subresource calculation in D3D10CalcSubResource().

Parameters:

pPointer - Returned pointer corresponding to subresource

pResource - Mapped resource to access

subResource - Subresource of pResource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceGetMappedPointer

5.15.3.7 cudaError_t cudaD3D10ResourceGetMappedSize (size_t * *pSize*, ID3D10Resource * *pResource*, unsigned int *subResource*)

Deprecated

This function is deprecated as of CUDA 3.0.

Returns in `*pSize` the size of the subresource of the mapped Direct3D resource `pResource` which corresponds to `subResource`. The value set in `pSize` may change every time that `pResource` is mapped.

If `pResource` has not been registered for use with CUDA then cudaErrorInvalidHandle is returned. If `pResource` was not registered with usage flags cudaD3D10RegisterFlagsNone, then cudaErrorInvalidResourceHandle is returned. If `pResource` is not mapped for access by CUDA then cudaErrorUnknown is returned.

For usage requirements of the `subResource` parameter see cudaD3D10ResourceGetMappedPointer().

Parameters:

pSize - Returned size of subresource

pResource - Mapped resource to access

subResource - Subresource of pResource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceGetMappedPointer

5.15.3.8 cudaError_t cudaD3D10ResourceGetSurfaceDimensions (size_t * *pWidth*, size_t * *pHeight*, size_t * *pDepth*, ID3D10Resource * *pResource*, unsigned int *subResource*)

Deprecated

This function is deprecated as of CUDA 3.0.

Returns in `*pWidth`, `*pHeight`, and `*pDepth` the dimensions of the subresource of the mapped Direct3D resource `pResource` which corresponds to `subResource`.

Since anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters `pWidth`, `pHeight`, and `pDepth` are optional. For 2D surfaces, the value returned in `*pDepth` will be 0.

If `pResource` is not of type ID3D10Texture1D, ID3D10Texture2D, or ID3D10Texture3D, or if `pResource` has not been registered for use with CUDA, then cudaErrorInvalidHandle is returned.

For usage requirements of `subResource` parameters see cudaD3D10ResourceGetMappedPointer().

Parameters:

pWidth - Returned width of surface

pHeight - Returned height of surface

pDepth - Returned depth of surface

pResource - Registered resource to access

subResource - Subresource of pResource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsSubResourceGetMappedArray

5.15.3.9 cudaError_t cudaD3D10ResourceSetMapFlags (ID3D10Resource * *pResource*, unsigned int *flags*)

Deprecated

This function is deprecated as of CUDA 3.0.

Set usage flags for mapping the Direct3D resource `pResource`.

Changes to flags will take effect the next time `pResource` is mapped. The `flags` argument may be any of the following:

- cudaD3D10MapFlagsNone: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- cudaD3D10MapFlagsReadOnly: Specifies that CUDA kernels which access this resource will not write to this resource.
- cudaD3D10MapFlagsWriteDiscard: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA then cudaErrorInvalidHandle is returned. If `pResource` is presently mapped for access by CUDA then cudaErrorUnknown is returned.

Parameters:

pResource - Registered resource to set flags for

flags - Parameters for resource mapping

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceSetMapFlags

5.15.3.10 cudaError_t cudaD3D10SetDirect3DDevice (ID3D10Device * *pD3D10Device*, int *device* = -1)

Deprecated

This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D10 device in order to achieve maximum interoperability performance.

Parameters:

pD3D10Device - Direct3D device to use for interoperability

device - The CUDA device to use. This device must be among the devices returned when querying cudaD3D10DeviceListAll from cudaD3D10GetDevices, may be set to -1 to automatically select an appropriate CUDA device.

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorSetOnActiveProcess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaD3D10GetDevice, cudaGraphicsD3D10RegisterResource, cudaDeviceReset

5.15.3.11 cudaError_t cudaD3D10UnmapResources (int *count*, ID3D10Resource ** *ppResources*)

Deprecated

This function is deprecated as of CUDA 3.0.

Unmaps the `count` Direct3D resource in `ppResources`.

This function provides the synchronization guarantee that any CUDA kernels issued before cudaD3D10UnmapResources() will complete before any Direct3D calls issued after cudaD3D10UnmapResources() begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then cudaErrorInvalidResourceHandle is returned. If any of `ppResources` are not presently mapped for access by CUDA then cudaErrorUnknown is returned.

Parameters:

count - Number of resources to unmap for CUDA

ppResources - Resources to unmap for CUDA

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnmapResources

5.15.3.12 cudaError_t cudaD3D10UnregisterResource (ID3D10Resource * *pResource*)

Deprecated

This function is deprecated as of CUDA 3.0.

Unregisters the Direct3D resource `resource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then cudaErrorInvalidResourceHandle is returned.

Parameters:

pResource - Resource to unregister

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource

5.16 Direct3D 11 Interoperability

Enumerations

- enum cudaD3D11DeviceList {
  cudaD3D11DeviceListAll = 1,
  cudaD3D11DeviceListCurrentFrame = 2,
  cudaD3D11DeviceListNextFrame = 3 }

Functions

- cudaError_t cudaD3D11GetDevice (int *device, IDXGIAdapter *pAdapter)
  Gets the device number for an adapter.
- cudaError_t cudaD3D11GetDevices (unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device *pD3D11Device, enum cudaD3D11DeviceList deviceList)
  Gets the CUDA devices corresponding to a Direct3D 11 device.
- cudaError_t cudaGraphicsD3D11RegisterResource (struct cudaGraphicsResource **resource, ID3D11Resource *pD3DResource, unsigned int flags)
  Register a Direct3D 11 resource for access by CUDA.

5.16.1 Detailed Description

This section describes the Direct3D 11 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 11 resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interopability.

5.16.2 Enumeration Type Documentation

5.16.2.1 enum cudaD3D11DeviceList

CUDA devices corresponding to a D3D11 device

Enumerator:

cudaD3D11DeviceListAll The CUDA devices for all GPUs used by a D3D11 device

cudaD3D11DeviceListCurrentFrame The CUDA devices for the GPUs used by a D3D11 device in its currently rendering frame

cudaD3D11DeviceListNextFrame The CUDA devices for the GPUs to be used by a D3D11 device in the next frame

5.16.3 Function Documentation

5.16.3.1 cudaError_t cudaD3D11GetDevice (int * *device*, IDXGIAdapter * *pAdapter*)

Returns in `*device` the CUDA-compatible device corresponding to the adapter `pAdapter` obtained from IDXGIFactory::EnumAdapters. This call will succeed only if a device on adapter `pAdapter` is CUDA-compatible.

Parameters:

device - Returns the device corresponding to pAdapter

pAdapter - D3D11 adapter to get device for

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer

5.16.3.2 cudaError_t cudaD3D11GetDevices (unsigned int * *pCudaDeviceCount*, int * *pCudaDevices*, unsigned int *cudaDeviceCount*, ID3D11Device * *pD3D11Device*, enum cudaD3D11DeviceList *deviceList*)

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the Direct3D 11 device `pD3D11Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 11 device `pD3D11Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return cudaErrorNoDevice.

Parameters:

pCudaDeviceCount - Returned number of CUDA devices corresponding to `pD3D11Device`

pCudaDevices - Returned CUDA devices corresponding to `pD3D11Device`

cudaDeviceCount - The size of the output device array `pCudaDevices`

pD3D11Device - Direct3D 11 device to query for CUDA devices

deviceList - The set of devices to return. This set may be cudaD3D11DeviceListAll for all devices, cudaD3D11DeviceListCurrentFrame for the devices used to render the current frame (in SLI), or cudaD3D11DeviceListNextFrame for the devices used to render the next frame (in SLI).

Returns:

cudaSuccess, cudaErrorNoDevice, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer

5.16.3.3 cudaError_t cudaGraphicsD3D11RegisterResource (struct cudaGraphicsResource ** *resource*, ID3D11Resource * *pD3DResource*, unsigned int *flags*)

Registers the Direct3D 11 resource `pD3DResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cudaGraphicsUnregisterResource(). Also on success, this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through cudaGraphicsUnregisterResource().

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following.

- ID3D11Buffer: may be accessed via a device pointer
- ID3D11Texture1D: individual subresources of the texture may be accessed via arrays
- ID3D11Texture2D: individual subresources of the texture may be accessed via arrays
- ID3D11Texture3D: individual subresources of the texture may be accessed via arrays

The `flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- cudaGraphicsRegisterFlagsNone: Specifies no hints about how this resource will be used.
- cudaGraphicsRegisterFlagsSurfaceLoadStore: Specifies that CUDA will bind this resource to a surface reference.
- cudaGraphicsRegisterFlagsTextureGather: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

A complete list of supported DXGI formats is as follows. For compactness the notation A_{B,C,D} represents A_B, A_C, and A_D.

- DXGI_FORMAT_A8_UNORM
- DXGI_FORMAT_B8G8R8A8_UNORM
- DXGI_FORMAT_B8G8R8X8_UNORM
- DXGI_FORMAT_R16_FLOAT
- DXGI_FORMAT_R16G16B16A16_{FLOAT,SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R16G16_{FLOAT,SINT,SNORM,UINT,UNORM}

- DXGI_FORMAT_R16_{SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R32_FLOAT
- DXGI_FORMAT_R32G32B32A32_{FLOAT,SINT,UINT}
- DXGI_FORMAT_R32G32_{FLOAT,SINT,UINT}
- DXGI_FORMAT_R32_{SINT,UINT}
- DXGI_FORMAT_R8G8B8A8_{SINT,SNORM,UINT,UNORM,UNORM_SRGB}
- DXGI_FORMAT_R8G8_{SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R8_{SINT,SNORM,UINT,UNORM}

If `pD3DResource` is of incorrect type or is already registered, then cudaErrorInvalidResourceHandle is returned. If `pD3DResource` cannot be registered, then cudaErrorUnknown is returned.

Parameters:

resource - Pointer to returned resource handle

pD3DResource - Direct3D resource to register

flags - Parameters for resource registration

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsUnregisterResource, cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsResourceGetMappedPointer

5.17 Direct3D 11 Interoperability [DEPRECATED]

Functions

- cudaError_t cudaD3D11GetDirect3DDevice (ID3D11Device **ppD3D11Device)
  Gets the Direct3D device against which the current CUDA context was created.
- cudaError_t cudaD3D11SetDirect3DDevice (ID3D11Device *pD3D11Device, int device=-1)
  Sets the Direct3D 11 device to use for interoperability with a CUDA device.

5.17.1 Detailed Description

This section describes deprecated Direct3D 11 interoperability functions.

5.17.2 Function Documentation

5.17.2.1 cudaError_t cudaD3D11GetDirect3DDevice (ID3D11Device ** *ppD3D11Device*)

Deprecated

This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D11 device in order to achieve maximum interoperability performance.

Parameters:

ppD3D11Device - Returns the Direct3D device for this thread

Returns:

cudaSuccess, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaD3D11SetDirect3DDevice

5.17.2.2 cudaError_t cudaD3D11SetDirect3DDevice (ID3D11Device * *pD3D11Device*, int *device* = -1)

Deprecated

This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D11 device in order to achieve maximum interoperability performance.

Parameters:

pD3D11Device - Direct3D device to use for interoperability

device - The CUDA device to use. This device must be among the devices returned when querying cudaD3D11DeviceListAll from cudaD3D11GetDevices, may be set to -1 to automatically select an appropriate CUDA device.

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorSetOnActiveProcess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaD3D11GetDevice, cudaGraphicsD3D11RegisterResource, cudaDeviceReset

5.18 VDPAU Interoperability

Functions

- cudaError_t cudaGraphicsVDPAURegisterOutputSurface (struct cudaGraphicsResource **resource, VdpOutputSurface vdpSurface, unsigned int flags)

  Register a VdpOutputSurface object.

- cudaError_t cudaGraphicsVDPAURegisterVideoSurface (struct cudaGraphicsResource **resource, VdpVideoSurface vdpSurface, unsigned int flags)

  Register a VdpVideoSurface object.

- cudaError_t cudaVDPAUGetDevice (int *device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

  Gets the CUDA device associated with a VdpDevice.

- cudaError_t cudaVDPAUSetVDPAUDevice (int device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

  Sets a CUDA device to use VDPAU interoperability.

5.18.1 Detailed Description

This section describes the VDPAU interoperability functions of the CUDA runtime application programming interface.

5.18.2 Function Documentation

5.18.2.1 cudaError_t cudaGraphicsVDPAURegisterOutputSurface (struct cudaGraphicsResource ** *resource*, VdpOutputSurface *vdpSurface*, unsigned int *flags*)

Registers the VdpOutputSurface specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `resource`. The surface's intended usage is specified using `flags`, as follows:

- cudaGraphicsMapFlagsNone: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- cudaGraphicsMapFlagsReadOnly: Specifies that CUDA will not write to this resource.
- cudaGraphicsMapFlagsWriteDiscard: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

Parameters:

resource - Pointer to the returned object handle

vdpSurface - VDPAU object to be registered

flags - Map flags

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaVDPAUSetVDPAUDevice, cudaGraphicsUnregisterResource, cudaGraphicsSubResourceGetMappedArray

5.18.2.2 cudaError_t cudaGraphicsVDPAURegisterVideoSurface (struct cudaGraphicsResource ** *resource*, VdpVideoSurface *vdpSurface*, unsigned int *flags*)

Registers the VdpVideoSurface specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `resource`. The surface's intended usage is specified using `flags`, as follows:

- cudaGraphicsMapFlagsNone: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- cudaGraphicsMapFlagsReadOnly: Specifies that CUDA will not write to this resource.
- cudaGraphicsMapFlagsWriteDiscard: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

Parameters:

resource - Pointer to the returned object handle

vdpSurface - VDPAU object to be registered

flags - Map flags

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaVDPAUSetVDPAUDevice, cudaGraphicsUnregisterResource, cudaGraphicsSubResourceGetMappedArray

5.18.2.3 cudaError_t cudaVDPAUGetDevice (int * *device*, VdpDevice *vdpDevice*, VdpGetProcAddress * *vdpGetProcAddress*)

Returns the CUDA device associated with a VdpDevice, if applicable.

Parameters:

device - Returns the device associated with vdpDevice, or -1 if the device associated with vdpDevice is not a compute device.

vdpDevice - A VdpDevice handle

vdpGetProcAddress - VDPAU's VdpGetProcAddress function pointer

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaVDPAUSetVDPAUDevice

5.18.2.4 cudaError_t cudaVDPAUSetVDPAUDevice (int *device*, VdpDevice *vdpDevice*, VdpGetProcAddress * *vdpGetProcAddress*)

Records `vdpDevice` as the VdpDevice for VDPAU interoperability with the CUDA device `device` and sets `device` as the current device for the calling host thread.

If `device` has already been initialized then this call will fail with the error cudaErrorSetOnActiveProcess. In this case it is necessary to reset `device` using cudaDeviceReset() before VDPAU interoperability on `device` may be enabled.

Parameters:

device - Device to use for VDPAU interoperability

vdpDevice - The VdpDevice to interoperate with

vdpGetProcAddress - VDPAU's VdpGetProcAddress function pointer

Returns:

cudaSuccess, cudaErrorInvalidDevice, cudaErrorSetOnActiveProcess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsVDPAURegisterVideoSurface, cudaGraphicsVDPAURegisterOutputSurface, cudaDeviceReset

5.19 Graphics Interoperability

Functions

- cudaError_t cudaGraphicsMapResources (int count, cudaGraphicsResource_t *resources, cudaStream_t stream=0)

  Map graphics resources for access by CUDA.

- cudaError_t cudaGraphicsResourceGetMappedMipmappedArray (cudaMipmappedArray_t *mipmappedArray, cudaGraphicsResource_t resource)

  Get a mipmapped array through which to access a mapped graphics resource.

- cudaError_t cudaGraphicsResourceGetMappedPointer (void **devPtr, size_t *size, cudaGraphicsResource_t resource)

  Get an device pointer through which to access a mapped graphics resource.

- cudaError_t cudaGraphicsResourceSetMapFlags (cudaGraphicsResource_t resource, unsigned int flags)

  Set usage flags for mapping a graphics resource.

- cudaError_t cudaGraphicsSubResourceGetMappedArray (cudaArray_t *array, cudaGraphicsResource_t resource, unsigned int arrayIndex, unsigned int mipLevel)

  Get an array through which to access a subresource of a mapped graphics resource.

- cudaError_t cudaGraphicsUnmapResources (int count, cudaGraphicsResource_t *resources, cudaStream_t stream=0)

  Unmap graphics resources.

- cudaError_t cudaGraphicsUnregisterResource (cudaGraphicsResource_t resource)

  Unregisters a graphics resource for access by CUDA.

5.19.1 Detailed Description

This section describes the graphics interoperability functions of the CUDA runtime application programming interface.

5.19.2 Function Documentation

5.19.2.1 cudaError_t cudaGraphicsMapResources (int *count*, cudaGraphicsResource_t * *resources*, cudaStream_t *stream* = `0`)

Maps the `count` graphics resources in `resources` for access by CUDA.

The resources in `resources` may be accessed by CUDA until they are unmapped. The graphics API from which `resources` were registered should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any graphics calls issued before cudaGraphicsMapResources() will complete before any subsequent CUDA work issued in `stream` begins.

If `resources` contains any duplicate entries then cudaErrorInvalidResourceHandle is returned. If any of `resources` are presently mapped for access by CUDA then cudaErrorUnknown is returned.

Parameters:

count - Number of resources to map

resources - Resources to map for CUDA

stream - Stream for synchronization

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceGetMappedPointer, cudaGraphicsSubResourceGetMappedArray, cudaGraphicsUnmapResources

5.19.2.2 cudaError_t cudaGraphicsResourceGetMappedMipmappedArray (cudaMipmappedArray_t * *mipmappedArray*, cudaGraphicsResource_t *resource*)

Returns in `*mipmappedArray` a mipmapped array through which the mapped graphics resource `resource` may be accessed. The value set in `mipmappedArray` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and cudaErrorUnknown is returned. If `resource` is not mapped then cudaErrorUnknown is returned.

Parameters:

mipmappedArray - Returned mipmapped array through which `resource` may be accessed

resource - Mapped resource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceGetMappedPointer

5.19.2.3 cudaError_t cudaGraphicsResourceGetMappedPointer (void ** *devPtr*, size_t * *size*, cudaGraphicsResource_t *resource*)

Returns in `*devPtr` a pointer through which the mapped graphics resource `resource` may be accessed. Returns in `*size` the size of the memory in bytes which may be accessed from that pointer. The value set in `devPtr` may change every time that `resource` is mapped.

If `resource` is not a buffer then it cannot be accessed via a pointer and cudaErrorUnknown is returned. If `resource` is not mapped then cudaErrorUnknown is returned. *

Parameters:

devPtr - Returned pointer through which `resource` may be accessed

size - Returned size of the buffer accessible starting at `*devPtr`

resource - Mapped resource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsMapResources, cudaGraphicsSubResourceGetMappedArray

5.19.2.4 cudaError_t cudaGraphicsResourceSetMapFlags (cudaGraphicsResource_t *resource*, unsigned int *flags*)

Set `flags` for mapping the graphics resource `resource`.

Changes to `flags` will take effect the next time `resource` is mapped. The `flags` argument may be any of the following:

- cudaGraphicsMapFlagsNone: Specifies no hints about how `resource` will be used. It is therefore assumed that CUDA may read from or write to `resource`.
- cudaGraphicsMapFlagsReadOnly: Specifies that CUDA will not write to `resource`.
- cudaGraphicsMapFlagsWriteDiscard: Specifies CUDA will not read from `resource` and will write over the entire contents of `resource`, so none of the data previously stored in `resource` will be preserved.

If `resource` is presently mapped for access by CUDA then cudaErrorUnknown is returned. If `flags` is not one of the above values then cudaErrorInvalidValue is returned.

Parameters:

resource - Registered resource to set flags for

flags - Parameters for resource mapping

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsMapResources

5.19.2.5 cudaError_t cudaGraphicsSubResourceGetMappedArray (cudaArray_t * *array*, cudaGraphicsResource_t *resource*, unsigned int *arrayIndex*, unsigned int *mipLevel*)

Returns in `*array` an array through which the subresource of the mapped graphics resource `resource` which corresponds to array index `arrayIndex` and mipmap level `mipLevel` may be accessed. The value set in `array` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and cudaErrorUnknown is returned. If `arrayIndex` is not a valid array index for `resource` then cudaErrorInvalidValue is returned. If `mipLevel` is not a valid mipmap level for `resource` then cudaErrorInvalidValue is returned. If `resource` is not mapped then cudaErrorUnknown is returned.

Parameters:

array - Returned array through which a subresource of `resource` may be accessed

resource - Mapped resource to access

arrayIndex - Array index for array textures or cubemap face index as defined by cudaGraphicsCubeFace for cubemap textures for the subresource to access

mipLevel - Mipmap level for the subresource to access

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsResourceGetMappedPointer

5.19.2.6 cudaError_t cudaGraphicsUnmapResources (int *count*, cudaGraphicsResource_t * *resources*, cudaStream_t *stream* = `0`)

Unmaps the `count` graphics resources in `resources`.

Once unmapped, the resources in `resources` may not be accessed by CUDA until they are mapped again.

This function provides the synchronization guarantee that any CUDA work issued in `stream` before cudaGraphicsUnmapResources() will complete before any subsequently issued graphics work begins.

If `resources` contains any duplicate entries then cudaErrorInvalidResourceHandle is returned. If any of `resources` are not presently mapped for access by CUDA then cudaErrorUnknown is returned.

Parameters:

count - Number of resources to unmap

resources - Resources to unmap

stream - Stream for synchronization

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsMapResources

5.19.2.7 cudaError_t cudaGraphicsUnregisterResource (cudaGraphicsResource_t *resource*)

Unregisters the graphics resource `resource` so it is not accessible by CUDA unless registered again.

If `resource` is invalid then cudaErrorInvalidResourceHandle is returned.

Parameters:

resource - Resource to unregister

Returns:

cudaSuccess, cudaErrorInvalidResourceHandle, cudaErrorUnknown

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGraphicsD3D9RegisterResource, cudaGraphicsD3D10RegisterResource, cudaGraphicsD3D11RegisterResource, cudaGraphicsGLRegisterBuffer, cudaGraphicsGLRegisterImage

5.20 Texture Reference Management

Functions

- cudaError_t cudaBindTexture (size_t *offset, const struct textureReference *texref, const void *devPtr, const struct cudaChannelFormatDesc *desc, size_t size=UINT_MAX)

  Binds a memory area to a texture.

- cudaError_t cudaBindTexture2D (size_t *offset, const struct textureReference *texref, const void *devPtr, const struct cudaChannelFormatDesc *desc, size_t width, size_t height, size_t pitch)

  Binds a 2D memory area to a texture.

- cudaError_t cudaBindTextureToArray (const struct textureReference *texref, cudaArray_const_t array, const struct cudaChannelFormatDesc *desc)

  Binds an array to a texture.

- cudaError_t cudaBindTextureToMipmappedArray (const struct textureReference *texref, cudaMipmappedArray_const_t mipmappedArray, const struct cudaChannelFormatDesc *desc)

  Binds a mipmapped array to a texture.

- struct cudaChannelFormatDesc cudaCreateChannelDesc (int x, int y, int z, int w, enum cudaChannelFormatKind f)

  Returns a channel descriptor using the specified format.

- cudaError_t cudaGetChannelDesc (struct cudaChannelFormatDesc *desc, cudaArray_const_t array)

  Get the channel descriptor of an array.

- cudaError_t cudaGetTextureAlignmentOffset (size_t *offset, const struct textureReference *texref)

  Get the alignment offset of a texture.

- cudaError_t cudaGetTextureReference (const struct textureReference **texref, const void *symbol)

  Get the texture reference associated with a symbol.

- cudaError_t cudaUnbindTexture (const struct textureReference *texref)

  Unbinds a texture.

5.20.1 Detailed Description

This section describes the low level texture reference management functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module.

5.20.2 Function Documentation

5.20.2.1 cudaError_t cudaBindTexture (size_t * *offset*, const struct textureReference * *texref*, const void * *devPtr*, const struct cudaChannelFormatDesc * *desc*, size_t *size* = UINT_MAX)

Binds `size` bytes of the memory area pointed to by `devPtr` to the texture reference `texref`. `desc` describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to `texref` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, cudaBindTexture() returns in `*offset` a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the tex1Dfetch() function. If the device memory pointer was returned from cudaMalloc(), the offset is guaranteed to be 0 and NULL may be passed as the `offset` parameter.

The total number of elements (or texels) in the linear address range cannot exceed cudaDeviceProp::maxTexture1DLinear[0]. The number of elements is computed as (`size` / elementSize), where elementSize is determined from `desc`.

Parameters:

offset - Offset in bytes

texref - Texture to bind

devPtr - Memory area on device

desc - Channel format

size - Size of the memory area pointed to by devPtr

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture2D (C API), cudaBindTextureToArray (C API), cudaUnbindTexture (C API), cudaGetTextureAlignmentOffset (C API)

5.20.2.2 cudaError_t cudaBindTexture2D (size_t * *offset*, const struct textureReference * *texref*, const void * *devPtr*, const struct cudaChannelFormatDesc * *desc*, size_t *width*, size_t *height*, size_t *pitch*)

Binds the 2D memory area pointed to by `devPtr` to the texture reference `texref`. The size of the area is constrained by `width` in texel units, `height` in texel units, and `pitch` in byte units. `desc` describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to `texref` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, cudaBindTexture2D() returns in `*offset` a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the tex2D() function. If the device memory pointer was returned from cudaMalloc(), the offset is guaranteed to be 0 and NULL may be passed as the `offset` parameter.

`width` and `height`, which are specified in elements (or texels), cannot exceed cudaDeviceProp::maxTexture2DLinear[0] and cudaDeviceProp::maxTexture2DLinear[1] respectively. `pitch`, which is specified in bytes, cannot exceed cudaDeviceProp::maxTexture2DLinear[2].

The driver returns cudaErrorInvalidValue if `pitch` is not a multiple of cudaDeviceProp::texturePitchAlignment.

Parameters:

offset - Offset in bytes

texref - Texture reference to bind

devPtr - 2D memory area on device

desc - Channel format

width - Width in texel units

height - Height in texel units

pitch - Pitch in bytes

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C API), cudaBindTextureToArray (C API), cudaGetTextureAlignmentOffset (C API)

5.20.2.3 cudaError_t cudaBindTextureToArray (const struct textureReference * *texref*, cudaArray_const_t *array*, const struct cudaChannelFormatDesc * *desc*)

Binds the CUDA array `array` to the texture reference `texref`. `desc` describes how the memory is interpreted when fetching values from the texture. Any CUDA array previously bound to `texref` is unbound.

Parameters:

texref - Texture to bind

array - Memory array on device

desc - Channel format

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API), cudaBindTexture2D (C API), cudaBindTextureToArray (C++ API), cudaUnbindTexture (C API), cudaGetTextureAlignmentOffset (C API)

5.20.2.4 cudaError_t cudaBindTextureToMipmappedArray (const struct textureReference * *texref*, cudaMipmappedArray_const_t *mipmappedArray*, const struct cudaChannelFormatDesc * *desc*)

Binds the CUDA mipmapped array `mipmappedArray` to the texture reference `texref`. `desc` describes how the memory is interpreted when fetching values from the texture. Any CUDA mipmapped array previously bound to `texref` is unbound.

Parameters:

texref - Texture to bind

mipmappedArray - Memory mipmapped array on device

desc - Channel format

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API), cudaBindTexture2D (C API), cudaBindTextureToArray (C++ API), cudaUnbindTexture (C API), cudaGetTextureAlignmentOffset (C API)

5.20.2.5 struct cudaChannelFormatDesc cudaCreateChannelDesc (int *x*, int *y*, int *z*, int *w*, enum cudaChannelFormatKind *f*) [read]

Returns a channel descriptor with format `f` and number of bits of each component `x`, `y`, `z`, and `w`. The cudaChannelFormatDesc is defined as:

```
struct cudaChannelFormatDesc {
  int x, y, z, w;
  enum cudaChannelFormatKind f;
};
```

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned, cudaChannelFormatKindUnsigned, or cudaChannelFormatKindFloat.

Parameters:

x - X component

y - Y component

z - Z component

w - W component

f - Channel format

Returns:

Channel descriptor with format `f`

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API), cudaBindTexture2D (C API), cudaBindTextureToArray (C API), cudaUnbindTexture (C API), cudaGetTextureAlignmentOffset (C API)

5.20.2.6 cudaError_t cudaGetChannelDesc (struct cudaChannelFormatDesc * *desc*, cudaArray_const_t *array*)

Returns in `*desc` the channel descriptor of the CUDA array `array`.

Parameters:

desc - Channel format

array - Memory array on device

Returns:

cudaSuccess, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C API), cudaGetTextureReference, cudaBindTexture (C API), cudaBindTexture2D (C API), cudaBindTextureToArray (C API), cudaUnbindTexture (C API), cudaGetTextureAlignmentOffset (C API)

5.20.2.7 cudaError_t cudaGetTextureAlignmentOffset (size_t * *offset*, const struct textureReference * *texref*)

Returns in `*offset` the offset that was returned when texture reference `texref` was bound.

Parameters:

offset - Offset of texture reference in bytes

texref - Texture to get offset of

Returns:

cudaSuccess, cudaErrorInvalidTexture, cudaErrorInvalidTextureBinding

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API), cudaBindTexture2D (C API), cudaBindTextureToArray (C API), cudaUnbindTexture (C API), cudaGetTextureAlignmentOffset (C++ API)

5.20.2.8 cudaError_t cudaGetTextureReference (const struct textureReference ** *texref*, const void * *symbol*)

Returns in `*texref` the structure associated to the texture reference defined by symbol `symbol`.

Parameters:

texref - Texture reference associated with symbol

symbol - Texture to get reference for

Returns:

cudaSuccess, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.
Use of a string naming a variable as the `symbol` paramater was removed in CUDA 5.0.

See also:

cudaCreateChannelDesc (C API), cudaGetChannelDesc, cudaGetTextureAlignmentOffset (C API), cudaBindTexture (C API), cudaBindTexture2D (C API), cudaBindTextureToArray (C API), cudaUnbindTexture (C API)

5.20.2.9 cudaError_t cudaUnbindTexture (const struct textureReference * *texref*)

Unbinds the texture bound to `texref`.

Parameters:

texref - Texture to unbind

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API), cudaBindTexture2D (C API), cudaBindTextureToArray (C API), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C API)

5.21 Surface Reference Management

Functions

- cudaError_t cudaBindSurfaceToArray (const struct surfaceReference *surfref, cudaArray_const_t array, const struct cudaChannelFormatDesc *desc)

  Binds an array to a surface.

- cudaError_t cudaGetSurfaceReference (const struct surfaceReference **surfref, const void *symbol)

  Get the surface reference associated with a symbol.

5.21.1 Detailed Description

This section describes the low level surface reference management functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the C++ API Routines module.

5.21.2 Function Documentation

5.21.2.1 cudaError_t cudaBindSurfaceToArray (const struct surfaceReference * *surfref*, cudaArray_const_t *array*, const struct cudaChannelFormatDesc * *desc*)

Binds the CUDA array `array` to the surface reference `surfref`. `desc` describes how the memory is interpreted when fetching values from the surface. Any CUDA array previously bound to `surfref` is unbound.

Parameters:

surfref - Surface to bind

array - Memory array on device

desc - Channel format

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSurface

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaBindSurfaceToArray (C++ API), cudaBindSurfaceToArray (C++ API, inherited channel descriptor), cudaGetSurfaceReference

5.21.2.2 cudaError_t cudaGetSurfaceReference (const struct surfaceReference ** *surfref*, const void * *symbol*)

Returns in `*surfref` the structure associated to the surface reference defined by symbol `symbol`.

Parameters:

surfref - Surface reference associated with symbol

symbol - Surface to get reference for

Returns:

cudaSuccess, cudaErrorInvalidSurface

Note:

Note that this function may also return error codes from previous, asynchronous launches.
Use of a string naming a variable as the `symbol` paramater was removed in CUDA 5.0.

See also:

cudaBindSurfaceToArray (C API)

5.22 Texture Object Management

Functions

- cudaError_t cudaCreateTextureObject (cudaTextureObject_t *pTexObject, const struct cudaResourceDesc *pResDesc, const struct cudaTextureDesc *pTexDesc, const struct cudaResourceViewDesc *pResViewDesc)

  Creates a texture object.

- cudaError_t cudaDestroyTextureObject (cudaTextureObject_t texObject)

  Destroys a texture object.

- cudaError_t cudaGetTextureObjectResourceDesc (struct cudaResourceDesc *pResDesc, cudaTextureObject_t texObject)

  Returns a texture object's resource descriptor.

- cudaError_t cudaGetTextureObjectResourceViewDesc (struct cudaResourceViewDesc *pResViewDesc, cudaTextureObject_t texObject)

  Returns a texture object's resource view descriptor.

- cudaError_t cudaGetTextureObjectTextureDesc (struct cudaTextureDesc *pTexDesc, cudaTextureObject_t texObject)

  Returns a texture object's texture descriptor.

5.22.1 Detailed Description

This section describes the low level texture object management functions of the CUDA runtime application programming interface. The texture object API is only supported on devices of compute capability 3.0 or higher.

5.22.2 Function Documentation

5.22.2.1 cudaError_t cudaCreateTextureObject (cudaTextureObject_t * *pTexObject*, const struct cudaResourceDesc * *pResDesc*, const struct cudaTextureDesc * *pTexDesc*, const struct cudaResourceViewDesc * *pResViewDesc*)

Creates a texture object and returns it in `pTexObject`. `pResDesc` describes the data to texture from. `pTexDesc` describes how the data should be sampled. `pResViewDesc` is an optional argument that specifies an alternate format for the data described by `pResDesc`, and also describes the subresource region to restrict access to when texturing. `pResViewDesc` can only be specified if the type of resource is a CUDA array or a CUDA mipmapped array.

Texture objects are only supported on devices of compute capability 3.0 or higher.

The cudaResourceDesc structure is defined as:

```
struct cudaResourceDesc {
    enum cudaResourceType resType;

    union {
        struct {
            cudaArray_t array;
        } array;
        struct {
            cudaMipmappedArray_t mipmap;
        } mipmap;
```

```
            struct {
                void *devPtr;
                struct cudaChannelFormatDesc desc;
                size_t sizeInBytes;
            } linear;
            struct {
                void *devPtr;
                struct cudaChannelFormatDesc desc;
                size_t width;
                size_t height;
                size_t pitchInBytes;
            } pitch2D;
        } res;
    };
```

where:

- cudaResourceDesc::resType specifies the type of resource to texture from. CUresourceType is defined as:

```
    enum cudaResourceType {
        cudaResourceTypeArray          = 0x00,
        cudaResourceTypeMipmappedArray = 0x01,
        cudaResourceTypeLinear         = 0x02,
        cudaResourceTypePitch2D        = 0x03
    };
```

  If cudaResourceDesc::resType is set to cudaResourceTypeArray, cudaResourceDesc::res::array::array must be set to a valid CUDA array handle.

  If cudaResourceDesc::resType is set to cudaResourceTypeMipmappedArray, cudaResourceDesc::res::mipmap::mipmap must be set to a valid CUDA mipmapped array handle.

  If cudaResourceDesc::resType is set to cudaResourceTypeLinear, cudaResourceDesc::res::linear::devPtr must be set to a valid device pointer, that is aligned to cudaDeviceProp::textureAlignment. cudaResourceDesc::res::linear::desc describes the format and the number of components per array element. cudaResourceDesc::res::linear::sizeInBytes specifies the size of the array in bytes. The total number of elements in the linear address range cannot exceed cudaDeviceProp::maxTexture1DLinear. The number of elements is computed as (sizeInBytes / sizeof(desc)).

  If cudaResourceDesc::resType is set to cudaResourceTypePitch2D, cudaResourceDesc::res::pitch2D::devPtr must be set to a valid device pointer, that is aligned to cudaDeviceProp::textureAlignment. cudaResourceDesc::res::pitch2D::desc describes the format and the number of components per array element. cudaResourceDesc::res::pitch2D::width and cudaResourceDesc::res::pitch2D::height specify the width and height of the array in elements, and cannot exceed cudaDeviceProp::maxTexture2DLinear[0] and cudaDeviceProp::maxTexture2DLinear[1] respectively. cudaResourceDesc::res::pitch2D::pitchInBytes specifies the pitch between two rows in bytes and has to be aligned to cudaDeviceProp::texturePitchAlignment. Pitch cannot exceed cudaDeviceProp::maxTexture2DLinear[2].

The cudaTextureDesc struct is defined as

```
    struct cudaTextureDesc {
        enum cudaTextureAddressMode addressMode[3];
        enum cudaTextureFilterMode  filterMode;
        enum cudaTextureReadMode    readMode;
```

```
    int                         sRGB;
    int                         normalizedCoords;
    unsigned int                maxAnisotropy;
    enum cudaTextureFilterMode  mipmapFilterMode;
    float                       mipmapLevelBias;
    float                       minMipmapLevelClamp;
    float                       maxMipmapLevelClamp;
};
```

where

- cudaTextureDesc::addressMode specifies the addressing mode for each dimension of the texture data. cudaTextureAddressMode is defined as:

  ```
  enum cudaTextureAddressMode {
      cudaAddressModeWrap   = 0,
      cudaAddressModeClamp  = 1,
      cudaAddressModeMirror = 2,
      cudaAddressModeBorder = 3
  };
  ```

  This is ignored if cudaResourceDesc::resType is cudaResourceTypeLinear. Also, if cudaTextureDesc::normalizedCoords is set to zero, the only supported address mode is cudaAddressModeClamp.

- cudaTextureDesc::filterMode specifies the filtering mode to be used when fetching from the texture. cudaTextureFilterMode is defined as:

  ```
  enum cudaTextureFilterMode {
      cudaFilterModePoint  = 0,
      cudaFilterModeLinear = 1
  };
  ```

  This is ignored if cudaResourceDesc::resType is cudaResourceTypeLinear.

- cudaTextureDesc::readMode specifies whether integer data should be converted to floating point or not. cudaTextureReadMode is defined as:

  ```
  enum cudaTextureReadMode {
      cudaReadModeElementType     = 0,
      cudaReadModeNormalizedFloat = 1
  };
  ```

  Note that this applies only to 8-bit and 16-bit integer formats. 32-bit integer format would not be promoted, regardless of whether or not this cudaTextureDesc::readMode is set cudaReadModeNormalizedFloat is specified.

- cudaTextureDesc::sRGB specifies whether sRGB to linear conversion should be performed during texture fetch.

- cudaTextureDesc::normalizedCoords specifies whether the texture coordinates will be normalized or not.

- cudaTextureDesc::maxAnisotropy specifies the maximum anistropy ratio to be used when doing anisotropic filtering. This value will be clamped to the range [1,16].

- cudaTextureDesc::mipmapFilterMode specifies the filter mode when the calculated mipmap level lies between two defined mipmap levels.

- cudaTextureDesc::mipmapLevelBias specifies the offset to be applied to the calculated mipmap level.

- cudaTextureDesc::minMipmapLevelClamp specifies the lower end of the mipmap level range to clamp access to.

- cudaTextureDesc::maxMipmapLevelClamp specifies the upper end of the mipmap level range to clamp access to.

The cudaResourceViewDesc struct is defined as

```
struct cudaResourceViewDesc {
    enum cudaResourceViewFormat format;
    size_t                      width;
    size_t                      height;
    size_t                      depth;
    unsigned int                firstMipmapLevel;
    unsigned int                lastMipmapLevel;
    unsigned int                firstLayer;
    unsigned int                lastLayer;
};
```

where:

- cudaResourceViewDesc::format specifies how the data contained in the CUDA array or CUDA mipmapped array should be interpreted. Note that this can incur a change in size of the texture data. If the resource view format is a block compressed format, then the underlying CUDA array or CUDA mipmapped array has to have a 32-bit unsigned integer format with 2 or 4 channels, depending on the block compressed format. For ex., BC1 and BC4 require the underlying CUDA array to have a 32-bit unsigned int with 2 channels. The other BC formats require the underlying resource to have the same 32-bit unsigned int format but with 4 channels.

- cudaResourceViewDesc::width specifies the new width of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original width of the resource. For non block compressed formats, this value has to be equal to that of the original resource.

- cudaResourceViewDesc::height specifies the new height of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original height of the resource. For non block compressed formats, this value has to be equal to that of the original resource.

- cudaResourceViewDesc::depth specifies the new depth of the texture data. This value has to be equal to that of the original resource.

- cudaResourceViewDesc::firstMipmapLevel specifies the most detailed mipmap level. This will be the new mipmap level zero. For non-mipmapped resources, this value has to be zero.cudaTextureDesc::minMipmapLevelClamp and cudaTextureDesc::maxMipmapLevelClamp will be relative to this value. For ex., if the firstMipmapLevel is set to 2, and a minMipmapLevelClamp of 1.2 is specified, then the actual minimum mipmap level clamp will be 3.2.

- cudaResourceViewDesc::lastMipmapLevel specifies the least detailed mipmap level. For non-mipmapped resources, this value has to be zero.

- cudaResourceViewDesc::firstLayer specifies the first layer index for layered textures. This will be the new layer zero. For non-layered resources, this value has to be zero.

- cudaResourceViewDesc::lastLayer specifies the last layer index for layered textures. For non-layered resources, this value has to be zero.

Parameters:

pTexObject - Texture object to create

pResDesc - Resource descriptor

pTexDesc - Texture descriptor

pResViewDesc - Resource view descriptor

Returns:

cudaSuccess, cudaErrorInvalidValue

See also:

cudaDestroyTextureObject

5.22.2.2 cudaError_t cudaDestroyTextureObject (cudaTextureObject_t *texObject*)

Destroys the texture object specified by `texObject`.

Parameters:

texObject - Texture object to destroy

Returns:

cudaSuccess, cudaErrorInvalidValue

See also:

cudaCreateTextureObject

5.22.2.3 cudaError_t cudaGetTextureObjectResourceDesc (struct cudaResourceDesc * *pResDesc*, cudaTextureObject_t *texObject*)

Returns the resource descriptor for the texture object specified by `texObject`.

Parameters:

pResDesc - Resource descriptor

texObject - Texture object

Returns:

cudaSuccess, cudaErrorInvalidValue

See also:

cudaCreateTextureObject

5.22.2.4 cudaError_t cudaGetTextureObjectResourceViewDesc (struct cudaResourceViewDesc * *pResViewDesc*, cudaTextureObject_t *texObject*)

Returns the resource view descriptor for the texture object specified by `texObject`. If no resource view was specified, cudaErrorInvalidValue is returned.

Parameters:

pResViewDesc - Resource view descriptor

texObject - Texture object

Returns:

cudaSuccess, cudaErrorInvalidValue

See also:

cudaCreateTextureObject

5.22.2.5 cudaError_t cudaGetTextureObjectTextureDesc (struct cudaTextureDesc * *pTexDesc*, cudaTextureObject_t *texObject*)

Returns the texture descriptor for the texture object specified by `texObject`.

Parameters:

pTexDesc - Texture descriptor

texObject - Texture object

Returns:

cudaSuccess, cudaErrorInvalidValue

See also:

cudaCreateTextureObject

5.23 Surface Object Management

Functions

- cudaError_t cudaCreateSurfaceObject (cudaSurfaceObject_t *pSurfObject, const struct cudaResourceDesc *pResDesc)

  Creates a surface object.

- cudaError_t cudaDestroySurfaceObject (cudaSurfaceObject_t surfObject)

  Destroys a surface object.

- cudaError_t cudaGetSurfaceObjectResourceDesc (struct cudaResourceDesc *pResDesc, cudaSurfaceObject_t surfObject)

  Returns a surface object's resource descriptor Returns the resource descriptor for the surface object specified by `surfObject`.

5.23.1 Detailed Description

This section describes the low level texture object management functions of the CUDA runtime application programming interface. The surface object API is only supported on devices of compute capability 3.0 or higher.

5.23.2 Function Documentation

5.23.2.1 cudaError_t cudaCreateSurfaceObject (cudaSurfaceObject_t * *pSurfObject*, const struct cudaResourceDesc * *pResDesc*)

Creates a surface object and returns it in `pSurfObject`. `pResDesc` describes the data to perform surface load/stores on. cudaResourceDesc::resType must be cudaResourceTypeArray and cudaResourceDesc::res::array::array must be set to a valid CUDA array handle.

Surface objects are only supported on devices of compute capability 3.0 or higher.

Parameters:

pSurfObject - Surface object to create

pResDesc - Resource descriptor

Returns:

cudaSuccess, cudaErrorInvalidValue

See also:

cudaDestroySurfaceObject

5.23.2.2 cudaError_t cudaDestroySurfaceObject (cudaSurfaceObject_t *surfObject*)

Destroys the surface object specified by `surfObject`.

Parameters:

surfObject - Surface object to destroy

Returns:

cudaSuccess, cudaErrorInvalidValue

See also:

cudaCreateSurfaceObject

5.23.2.3 cudaError_t cudaGetSurfaceObjectResourceDesc (struct cudaResourceDesc * *pResDesc*, cudaSurfaceObject_t *surfObject*)

Parameters:

pResDesc - Resource descriptor

surfObject - Surface object

Returns:

cudaSuccess, cudaErrorInvalidValue

See also:

cudaCreateSurfaceObject

5.24 Version Management

Functions

- cudaError_t cudaDriverGetVersion (int *driverVersion)

  Returns the CUDA driver version.

- __cudart_builtin__ cudaError_t cudaRuntimeGetVersion (int *runtimeVersion)

  Returns the CUDA Runtime version.

5.24.1 Function Documentation

5.24.1.1 cudaError_t cudaDriverGetVersion (int * *driverVersion*)

Returns in `*driverVersion` the version number of the installed CUDA driver. If no driver is installed, then 0 is returned as the driver version (via `driverVersion`). This function automatically returns cudaErrorInvalidValue if the `driverVersion` argument is NULL.

Parameters:

driverVersion - Returns the CUDA driver version.

Returns:

cudaSuccess, cudaErrorInvalidValue

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaRuntimeGetVersion

5.24.1.2 __cudart_builtin__ cudaError_t cudaRuntimeGetVersion (int * *runtimeVersion*)

Returns in `*runtimeVersion` the version number of the installed CUDA Runtime. This function automatically returns cudaErrorInvalidValue if the `runtimeVersion` argument is NULL.

Parameters:

runtimeVersion - Returns the CUDA Runtime version.

Returns:

cudaSuccess, cudaErrorInvalidValue

See also:

cudaDriverGetVersion

5.25 C++ API Routines

C++-style interface built on top of CUDA runtime API.

Functions

- template<class T , int dim>
  cudaError_t cudaBindSurfaceToArray (const struct surface< T, dim > &surf, cudaArray_const_t array)

  [C++ API] Binds an array to a surface

- template<class T , int dim>
  cudaError_t cudaBindSurfaceToArray (const struct surface< T, dim > &surf, cudaArray_const_t array, const struct cudaChannelFormatDesc &desc)

  [C++ API] Binds an array to a surface

- template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTexture (size_t *offset, const struct texture< T, dim, readMode > &tex, const void *devPtr, size_t size=UINT_MAX)

  [C++ API] Binds a memory area to a texture

- template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTexture (size_t *offset, const struct texture< T, dim, readMode > &tex, const void *devPtr, const struct cudaChannelFormatDesc &desc, size_t size=UINT_MAX)

  [C++ API] Binds a memory area to a texture

- template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTexture2D (size_t *offset, const struct texture< T, dim, readMode > &tex, const void *devPtr, size_t width, size_t height, size_t pitch)

  [C++ API] Binds a 2D memory area to a texture

- template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTexture2D (size_t *offset, const struct texture< T, dim, readMode > &tex, const void *devPtr, const struct cudaChannelFormatDesc &desc, size_t width, size_t height, size_t pitch)

  [C++ API] Binds a 2D memory area to a texture

- template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTextureToArray (const struct texture< T, dim, readMode > &tex, cudaArray_const_t array)

  [C++ API] Binds an array to a texture

- template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTextureToArray (const struct texture< T, dim, readMode > &tex, cudaArray_const_t array, const struct cudaChannelFormatDesc &desc)

  [C++ API] Binds an array to a texture

- template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTextureToMipmappedArray (const struct texture< T, dim, readMode > &tex, cudaMipmappedArray_const_t mipmappedArray)

  [C++ API] Binds a mipmapped array to a texture

- template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaBindTextureToMipmappedArray (const struct texture< T, dim, readMode > &tex, cudaMipmappedArray_const_t mipmappedArray, const struct cudaChannelFormatDesc &desc)

  [C++ API] Binds a mipmapped array to a texture

- template<class T >
  cudaChannelFormatDesc cudaCreateChannelDesc (void)

  [C++ API] Returns a channel descriptor using the specified format

- cudaError_t cudaEventCreate (cudaEvent_t *event, unsigned int flags)

  [C++ API] Creates an event object with the specified flags

- template<class T >
  cudaError_t cudaFuncGetAttributes (struct cudaFuncAttributes *attr, T *entry)

  [C++ API] Find out attributes for a given function

- template<class T >
  cudaError_t cudaFuncSetCacheConfig (T *func, enum cudaFuncCache cacheConfig)

  [C++ API] Sets the preferred cache configuration for a device function

- template<class T >
  cudaError_t cudaGetSymbolAddress (void **devPtr, const T &symbol)

  [C++ API] Finds the address associated with a CUDA symbol

- template<class T >
  cudaError_t cudaGetSymbolSize (size_t *size, const T &symbol)

  [C++ API] Finds the size of the object associated with a CUDA symbol

- template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaGetTextureAlignmentOffset (size_t *offset, const struct texture< T, dim, readMode > &tex)

  [C++ API] Get the alignment offset of a texture

- template<class T >
  cudaError_t cudaLaunch (T *func)

  [C++ API] Launches a device function

- cudaError_t cudaMallocHost (void **ptr, size_t size, unsigned int flags)

  [C++ API] Allocates page-locked memory on the host

- template<class T >
  cudaError_t cudaMemcpyFromSymbol (void *dst, const T &symbol, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyDeviceToHost)

  [C++ API] Copies data from the given symbol on the device

- template<class T >
  cudaError_t cudaMemcpyFromSymbolAsync (void *dst, const T &symbol, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyDeviceToHost, cudaStream_t stream=0)

  [C++ API] Copies data from the given symbol on the device

- template<class T >
  cudaError_t cudaMemcpyToSymbol (const T &symbol, const void *src, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyHostToDevice)

[C++ API] Copies data to the given symbol on the device

- template<class T >
  cudaError_t cudaMemcpyToSymbolAsync (const T &symbol, const void *src, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyHostToDevice, cudaStream_t stream=0)

  [C++ API] Copies data to the given symbol on the device

- template<class T >
  cudaError_t cudaSetupArgument (T arg, size_t offset)

  [C++ API] Configure a device launch

- template<class T , int dim, enum cudaTextureReadMode readMode>
  cudaError_t cudaUnbindTexture (const struct texture< T, dim, readMode > &tex)

  [C++ API] Unbinds a texture

5.25.1 Detailed Description

This section describes the C++ high level API functions of the CUDA runtime application programming interface. To use these functions, your application needs to be compiled with the `nvcc` compiler.

5.25.2 Function Documentation

5.25.2.1 template<class T , int dim> cudaError_t cudaBindSurfaceToArray (const struct surface< T, dim > & *surf*, cudaArray_const_t *array*)

Binds the CUDA array `array` to the surface reference `surf`. The channel descriptor is inherited from the CUDA array. Any CUDA array previously bound to `surf` is unbound.

Parameters:

surf - Surface to bind

array - Memory array on device

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSurface

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaBindSurfaceToArray (C API), cudaBindSurfaceToArray (C++ API)

5.25.2.2 template<class T , int dim> cudaError_t cudaBindSurfaceToArray (const struct surface< T, dim > & *surf*, cudaArray_const_t *array*, const struct cudaChannelFormatDesc & *desc*)

Binds the CUDA array `array` to the surface reference `surf`. `desc` describes how the memory is interpreted when dealing with the surface. Any CUDA array previously bound to `surf` is unbound.

Parameters:

surf - Surface to bind

array - Memory array on device

desc - Channel format

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSurface

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaBindSurfaceToArray (C API), cudaBindSurfaceToArray (C++ API, inherited channel descriptor)

5.25.2.3 template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTexture (size_t * *offset*, const struct texture< T, dim, readMode > & *tex*, const void * *devPtr*, size_t *size* = UINT_MAX)

Binds `size` bytes of the memory area pointed to by `devPtr` to texture reference `tex`. The channel descriptor is inherited from the texture reference type. The `offset` parameter is an optional byte offset as with the low-level cudaBindTexture(size_t*, const struct textureReference*, const void*, const struct cudaChannelFormatDesc*, size_t) function. Any memory previously bound to `tex` is unbound.

Parameters:

offset - Offset in bytes

tex - Texture to bind

devPtr - Memory area on device

size - Size of the memory area pointed to by devPtr

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API), cudaBindTexture (C++ API), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C++ API), cudaBindTextureToArray (C++ API, inherited channel descriptor), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)

5.25.2.4 template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTexture (size_t * *offset*, const struct texture< T, dim, readMode > & *tex*, const void * *devPtr*, const struct cudaChannelFormatDesc & *desc*, size_t *size* = UINT_MAX)

Binds `size` bytes of the memory area pointed to by `devPtr` to texture reference `tex`. `desc` describes how the memory is interpreted when fetching values from the texture. The `offset` parameter is an optional byte offset as with the low-level cudaBindTexture() function. Any memory previously bound to `tex` is unbound.

Parameters:

offset - Offset in bytes

tex - Texture to bind

devPtr - Memory area on device

desc - Channel format

size - Size of the memory area pointed to by devPtr

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C++ API), cudaBindTextureToArray (C++ API, inherited channel descriptor), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)

5.25.2.5 template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTexture2D (size_t * *offset*, const struct texture< T, dim, readMode > & *tex*, const void * *devPtr*, size_t *width*, size_t *height*, size_t *pitch*)

Binds the 2D memory area pointed to by `devPtr` to the texture reference `tex`. The size of the area is constrained by `width` in texel units, `height` in texel units, and `pitch` in byte units. The channel descriptor is inherited from the texture reference type. Any memory previously bound to `tex` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, cudaBindTexture2D() returns in `*offset` a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the tex2D() function. If the device memory pointer was returned from cudaMalloc(), the offset is guaranteed to be 0 and NULL may be passed as the `offset` parameter.

Parameters:

offset - Offset in bytes

tex - Texture reference to bind

devPtr - 2D memory area on device

width - Width in texel units

height - Height in texel units

pitch - Pitch in bytes

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C API), cudaBindTexture2D (C++ API), cudaBindTextureToArray (C++ API), cudaBindTextureToArray (C++ API, inherited channel descriptor), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)

5.25.2.6 template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTexture2D (size_t * *offset*, const struct texture< T, dim, readMode > & *tex*, const void * *devPtr*, const struct cudaChannelFormatDesc & *desc*, size_t *width*, size_t *height*, size_t *pitch*)

Binds the 2D memory area pointed to by `devPtr` to the texture reference `tex`. The size of the area is constrained by `width` in texel units, `height` in texel units, and `pitch` in byte units. `desc` describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to `tex` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, cudaBindTexture2D() returns in `*offset` a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the tex2D() function. If the device memory pointer was returned from cudaMalloc(), the offset is guaranteed to be 0 and NULL may be passed as the `offset` parameter.

Parameters:

offset - Offset in bytes

tex - Texture reference to bind

devPtr - 2D memory area on device

desc - Channel format

width - Width in texel units

height - Height in texel units

pitch - Pitch in bytes

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C++ API), cudaBindTextureToArray (C++ API, inherited channel descriptor), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)

5.25.2.7 template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTextureToArray (const struct texture< T, dim, readMode > & *tex*, cudaArray_const_t *array*)

Binds the CUDA array `array` to the texture reference `tex`. The channel descriptor is inherited from the CUDA array. Any CUDA array previously bound to `tex` is unbound.

Parameters:

tex - Texture to bind

array - Memory array on device

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C API), cudaBindTextureToArray (C++ API), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)

5.25.2.8 template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTextureToArray (const struct texture< T, dim, readMode > & *tex*, cudaArray_const_t *array*, const struct cudaChannelFormatDesc & *desc*)

Binds the CUDA array `array` to the texture reference `tex`. `desc` describes how the memory is interpreted when fetching values from the texture. Any CUDA array previously bound to `tex` is unbound.

Parameters:

tex - Texture to bind

array - Memory array on device

desc - Channel format

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C API), cudaBindTextureToArray (C++ API, inherited channel descriptor), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)

5.25.2.9 template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTextureToMipmappedArray (const struct texture< T, dim, readMode > & *tex*, cudaMipmappedArray_const_t *mipmappedArray*)

Binds the CUDA mipmapped array `mipmappedArray` to the texture reference `tex`. The channel descriptor is inherited from the CUDA array. Any CUDA mipmapped array previously bound to `tex` is unbound.

Parameters:

tex - Texture to bind

mipmappedArray - Memory mipmapped array on device

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C API), cudaBindTextureToArray (C++ API), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)

5.25.2.10 template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaBindTextureToMipmappedArray (const struct texture< T, dim, readMode > & *tex*, cudaMipmappedArray_const_t *mipmappedArray*, const struct cudaChannelFormatDesc & *desc*)

Binds the CUDA mipmapped array `mipmappedArray` to the texture reference `tex`. `desc` describes how the memory is interpreted when fetching values from the texture. Any CUDA mipmapped array previously bound to `tex` is unbound.

Parameters:

tex - Texture to bind

mipmappedArray - Memory mipmapped array on device

desc - Channel format

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C API), cudaBindTextureToArray (C++ API, inherited channel descriptor), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C++ API)

5.25.2.11 template<class T > cudaChannelFormatDesc cudaCreateChannelDesc (void)

Returns a channel descriptor with format `f` and number of bits of each component `x`, `y`, `z`, and `w`. The cudaChannelFormatDesc is defined as:

```
struct cudaChannelFormatDesc {
  int x, y, z, w;
  enum cudaChannelFormatKind f;
};
```

where cudaChannelFormatKind is one of cudaChannelFormatKindSigned, cudaChannelFormatKindUnsigned, or cudaChannelFormatKindFloat.

Returns:

Channel descriptor with format `f`

See also:

cudaCreateChannelDesc (Low level), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (High level), cudaBindTexture (High level, inherited channel descriptor), cudaBindTexture2D (High level), cudaBindTextureToArray (High level), cudaBindTextureToArray (High level, inherited channel descriptor), cudaUnbindTexture (High level), cudaGetTextureAlignmentOffset (High level)

5.25.2.12 cudaError_t cudaEventCreate (cudaEvent_t * *event*, unsigned int *flags*)

Creates an event object with the specified flags. Valid flags include:

- cudaEventDefault: Default event creation flag.
- cudaEventBlockingSync: Specifies that event should use blocking synchronization. A host thread that uses cudaEventSynchronize() to wait on an event created with this flag will block until the event actually completes.
- cudaEventDisableTiming: Specifies that the created event does not need to record timing data. Events created with this flag specified and the cudaEventBlockingSync flag not specified will provide the best performance when used with cudaStreamWaitEvent() and cudaEventQuery().

Parameters:

event - Newly created event

flags - Flags for new event

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidValue, cudaErrorLaunchFailure, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaEventCreate (C API), cudaEventCreateWithFlags, cudaEventRecord, cudaEventQuery, cudaEventSynchronize, cudaEventDestroy, cudaEventElapsedTime, cudaStreamWaitEvent

5.25.2.13 template<class T > cudaError_t cudaFuncGetAttributes (struct cudaFuncAttributes * *attr*, T * *entry*)

This function obtains the attributes of a function specified via `entry`. The parameter `entry` must be a pointer to a function that executes on the device. The parameter specified by `entry` must be declared as a `__global__` function. The fetched attributes are placed in `attr`. If the specified function does not exist, then cudaErrorInvalidDeviceFunction is returned.

Note that some function attributes such as maxThreadsPerBlock may vary based on the device that is currently being used.

Parameters:

attr - Return pointer to function's attributes

entry - Function to get attributes of

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidDeviceFunction

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaConfigureCall, cudaFuncSetCacheConfig (C++ API), cudaFuncGetAttributes (C API), cudaLaunch (C++ API), cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaSetupArgument (C++ API)

5.25.2.14 template<class T > cudaError_t cudaFuncSetCacheConfig (T * *func*, enum cudaFuncCache *cacheConfig*)

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `cacheConfig` the preferred cache configuration for the function specified via `func`. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute `func`.

`func` must be a pointer to a function that executes on the device. The parameter specified by `func` must be declared as a `__global__` function. If the specified function does not exist, then cudaErrorInvalidDeviceFunction is returned.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- cudaFuncCachePreferNone: no preference for shared memory or L1 (default)
- cudaFuncCachePreferShared: prefer larger shared memory and smaller L1 cache
- cudaFuncCachePreferL1: prefer larger L1 cache and smaller shared memory

Parameters:

func - device function pointer

cacheConfig - Requested cache configuration

Returns:

cudaSuccess, cudaErrorInitializationError, cudaErrorInvalidDeviceFunction

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaConfigureCall, cudaFuncSetCacheConfig (C API), cudaFuncGetAttributes (C++ API), cudaLaunch (C API), cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaSetupArgument (C++ API), cudaThreadGetCacheConfig, cudaThreadSetCacheConfig

5.25.2.15 template<class T > cudaError_t cudaGetSymbolAddress (void ** *devPtr*, const T & *symbol*)

Returns in `*devPtr` the address of symbol `symbol` on the device. `symbol` can either be a variable that resides in global or constant memory space. If `symbol` cannot be found, or if `symbol` is not declared in the global or constant memory space, `*devPtr` is unchanged and the error cudaErrorInvalidSymbol is returned.

Parameters:

devPtr - Return device pointer associated with symbol

symbol - Device symbol reference

Returns:

cudaSuccess, cudaErrorInvalidSymbol

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGetSymbolAddress (C API), cudaGetSymbolSize (C++ API)

5.25.2.16 template<class T > cudaError_t cudaGetSymbolSize (size_t * *size*, const T & *symbol*)

Returns in `*size` the size of symbol `symbol`. `symbol` must be a variable that resides in global or constant memory space. If `symbol` cannot be found, or if `symbol` is not declared in global or constant memory space, `*size` is unchanged and the error cudaErrorInvalidSymbol is returned.

Parameters:

size - Size of object associated with symbol

symbol - Device symbol reference

Returns:

cudaSuccess, cudaErrorInvalidSymbol

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaGetSymbolAddress (C++ API), cudaGetSymbolSize (C API)

5.25.2.17 template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaGetTextureAlignmentOffset (size_t * *offset*, const struct texture< T, dim, readMode > & *tex*)

Returns in `*offset` the offset that was returned when texture reference `tex` was bound.

Parameters:

offset - Offset of texture reference in bytes

tex - Texture to get offset of

Returns:

cudaSuccess, cudaErrorInvalidTexture, cudaErrorInvalidTextureBinding

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C++ API), cudaBindTextureToArray (C++ API, inherited channel descriptor), cudaUnbindTexture (C++ API), cudaGetTextureAlignmentOffset (C API)

5.25.2.18 template<class T > cudaError_t cudaLaunch (T * *func*)

Launches the function `entry` on the device. The parameter `entry` must be a function that executes on the device. The parameter specified by `entry` must be declared as a `__global__` function. cudaLaunch() must be preceded by a call to cudaConfigureCall() since it pops the data that was pushed by cudaConfigureCall() from the execution stack.

Parameters:

entry - Device function pointer to execute

Returns:

cudaSuccess, cudaErrorInvalidDeviceFunction, cudaErrorInvalidConfiguration, cudaErrorLaunchFailure, cudaErrorLaunchTimeout, cudaErrorLaunchOutOfResources, cudaErrorSharedObjectSymbolNotFound, cudaErrorSharedObjectInitFailed

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaConfigureCall, cudaFuncSetCacheConfig (C++ API), cudaFuncGetAttributes (C++ API), cudaLaunch (C API), cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaSetupArgument (C++ API), cudaThreadGetCacheConfig, cudaThreadSetCacheConfig

5.25.2.19 cudaError_t cudaMallocHost (void ** *ptr*, size_t *size*, unsigned int *flags*)

Allocates `size` bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as cudaMemcpy(). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc(). Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- cudaHostAllocDefault: This flag's value is defined to be 0.
- cudaHostAllocPortable: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- cudaHostAllocMapped: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling cudaHostGetDevicePointer().
- cudaHostAllocWriteCombined: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host->device transfers.

All of these flags are orthogonal to one another: a developer may allocate memory that is portable, mapped and/or write-combined with no restrictions.

cudaSetDeviceFlags() must have been called with the cudaDeviceMapHost flag in order for the cudaHostAllocMapped flag to have any effect.

The cudaHostAllocMapped flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to cudaHostGetDevicePointer() because the memory may be mapped into other CUDA contexts via the cudaHostAllocPortable flag.

Memory allocated by this function must be freed with cudaFreeHost().

Parameters:

ptr - Device pointer to allocated memory

size - Requested allocation size in bytes

flags - Requested properties of allocated memory

Returns:

cudaSuccess, cudaErrorMemoryAllocation

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaSetDeviceFlags, cudaMallocHost (C API), cudaFreeHost, cudaHostAlloc

5.25.2.20 template<class T > cudaError_t cudaMemcpyFromSymbol (void * *dst*, const T & *symbol*, size_t *count*, size_t *offset* = 0, enum cudaMemcpyKind *kind* = cudaMemcpyDeviceToHost)

Copies `count` bytes from the memory area `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either cudaMemcpyDeviceToHost or cudaMemcpyDeviceToDevice.

Parameters:

dst - Destination memory address

symbol - Device symbol reference

count - Size in bytes to copy

offset - Offset from start of symbol in bytes

kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.
Use of a string naming a variable as the `symbol` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.25.2.21 template<class T > cudaError_t cudaMemcpyFromSymbolAsync (void * *dst*, const T & *symbol*, size_t *count*, size_t *offset* = 0, enum cudaMemcpyKind *kind* = cudaMemcpyDeviceToHost, cudaStream_t *stream* = 0)

Copies `count` bytes from the memory area `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either cudaMemcpyDeviceToHost or cudaMemcpyDeviceToDevice.

cudaMemcpyFromSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is cudaMemcpyDeviceToHost and `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

dst - Destination memory address

symbol - Device symbol reference

count - Size in bytes to copy

offset - Offset from start of symbol in bytes

kind - Type of transfer

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.
Use of a string naming a variable as the `symbol` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync

5.25.2.22 template<class T > cudaError_t cudaMemcpyToSymbol (const T & *symbol*, const void * *src*, size_t *count*, size_t *offset* = `0`, enum cudaMemcpyKind *kind* = `cudaMemcpyHostToDevice`)

Copies `count` bytes from the memory area pointed to by `src` to the memory area `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either cudaMemcpyHostToDevice or cudaMemcpyDeviceToDevice.

Parameters:

symbol - Device symbol reference

src - Source memory address

count - Size in bytes to copy

offset - Offset from start of symbol in bytes

kind - Type of transfer

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits synchronous behavior for most use cases.
Use of a string naming a variable as the `symbol` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyToSymbolAsync, cudaMemcpyFromSymbolAsync

5.25.2.23 template<class T > cudaError_t cudaMemcpyToSymbolAsync (const T & *symbol*, const void * *src*, size_t *count*, size_t *offset* = 0, enum cudaMemcpyKind *kind* = cudaMemcpyHostToDevice, cudaStream_t *stream* = 0)

Copies `count` bytes from the memory area pointed to by `src` to the memory area `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either cudaMemcpyHostToDevice or cudaMemcpyDeviceToDevice.

cudaMemcpyToSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is cudaMemcpyHostToDevice and `stream` is non-zero, the copy may overlap with operations in other streams.

Parameters:

symbol - Device symbol reference

src - Source memory address

count - Size in bytes to copy

offset - Offset from start of symbol in bytes

kind - Type of transfer

stream - Stream identifier

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Note:

Note that this function may also return error codes from previous, asynchronous launches.
This function exhibits asynchronous behavior for most use cases.
Use of a string naming a variable as the `symbol` paramater was deprecated in CUDA 4.1 and removed in CUDA 5.0.

See also:

cudaMemcpy, cudaMemcpy2D, cudaMemcpyToArray, cudaMemcpy2DToArray, cudaMemcpyFromArray, cudaMemcpy2DFromArray, cudaMemcpyArrayToArray, cudaMemcpy2DArrayToArray, cudaMemcpyToSymbol, cudaMemcpyFromSymbol, cudaMemcpyAsync, cudaMemcpy2DAsync, cudaMemcpyToArrayAsync, cudaMemcpy2DToArrayAsync, cudaMemcpyFromArrayAsync, cudaMemcpy2DFromArrayAsync, cudaMemcpyFromSymbolAsync

5.25.2.24 template<class T > cudaError_t cudaSetupArgument (T *arg*, size_t *offset*)

Pushes `size` bytes of the argument pointed to by `arg` at `offset` bytes from the start of the parameter passing area, which starts at offset 0. The arguments are stored in the top of the execution stack. cudaSetupArgument() must be preceded by a call to cudaConfigureCall().

Parameters:

arg - Argument to push for a kernel launch

offset - Offset in argument stack to push new arg

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaConfigureCall, cudaFuncGetAttributes (C++ API), cudaLaunch (C++ API), cudaSetDoubleForDevice, cudaSetDoubleForHost, cudaSetupArgument (C API)

5.25.2.25 template<class T , int dim, enum cudaTextureReadMode readMode> cudaError_t cudaUnbindTexture (const struct texture< T, dim, readMode > & *tex*)

Unbinds the texture bound to `tex`.

Parameters:

tex - Texture to unbind

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaCreateChannelDesc (C++ API), cudaGetChannelDesc, cudaGetTextureReference, cudaBindTexture (C++ API), cudaBindTexture (C++ API, inherited channel descriptor), cudaBindTexture2D (C++ API), cudaBindTexture2D (C++ API, inherited channel descriptor), cudaBindTextureToArray (C++ API), cudaBindTextureToArray (C++ API, inherited channel descriptor), cudaUnbindTexture (C API), cudaGetTextureAlignmentOffset (C++ API)

5.26 Interactions with the CUDA Driver API

Interactions between the CUDA Driver API and the CUDA Runtime API.

This section describes the interactions between the CUDA Driver API and the CUDA Runtime API

5.26.1 Primary Contexts

There exists a one to one relationship between CUDA devices in the CUDA Runtime API and CUcontext s in the CUDA Driver API within a process. The specific context which the CUDA Runtime API uses for a device is called the device's primary context. From the perspective of the CUDA Runtime API, a device and its primary context are synonymous.

5.26.2 Initialization and Tear-Down

CUDA Runtime API calls operate on the CUDA Driver API CUcontext which is current to to the calling host thread.

The function cudaSetDevice() makes the primary context for the specified device current to the calling thread by calling cuCtxSetCurrent().

The CUDA Runtime API will automatically initialize the primary context for a device at the first CUDA Runtime API call which requires an active context. If no CUcontext is current to the calling thread when a CUDA Runtime API call which requires an active context is made, then the primary context for a device will be selected, made current to the calling thread, and initialized.

The context which the CUDA Runtime API initializes will be initialized using the parameters specified by the CUDA Runtime API functions cudaSetDeviceFlags(), cudaD3D9SetDirect3DDevice(), cudaD3D10SetDirect3DDevice(), cudaD3D11SetDirect3DDevice(), cudaGLSetGLDevice(), and cudaVDPAUSetVDPAUDevice(). Note that these functions will fail with cudaErrorSetOnActiveProcess if they are called when the primary context for the specified device has already been initialized. (or if the current device has already been initialized, in the case of cudaSetDeviceFlags()).

Primary contexts will remain active until they are explicitly deinitialized using cudaDeviceReset(). The function cudaDeviceReset() will deinitialize the primary context for the calling thread's current device immediately. The context will remain current to all of the threads that it was current to. The next CUDA Runtime API call on any thread which requires an active context will trigger the reinitialization of that device's primary context.

Note that there is no reference counting of the primary context's lifetime. It is recommended that the primary context not be deinitialized except just before exit or to recover from an unspecified launch failure.

5.26.3 Context Interoperability

Note that the use of multiple CUcontext s per device within a single process will substantially degrade performance and is strongly discouraged. Instead, it is highly recommended that the implicit one-to-one device-to-context mapping for the process provided by the CUDA Runtime API be used.

If a non-primary CUcontext created by the CUDA Driver API is current to a thread then the CUDA Runtime API calls to that thread will operate on that CUcontext, with some exceptions listed below. Interoperability between data types is discussed in the following sections.

The function cudaPointerGetAttributes() will return the error cudaErrorIncompatibleDriverContext if the pointer being queried was allocated by a non-primary context. The function cudaDeviceEnablePeerAccess() and the rest of the peer access API may not be called when a non-primary CUcontext is current. To use the pointer query and peer access APIs with a context created using the CUDA Driver API, it is necessary that the CUDA Driver API be used to access these features.

All CUDA Runtime API state (e.g, global variables' addresses and values) travels with its underlying CUcontext. In particular, if a CUcontext is moved from one thread to another then all CUDA Runtime API state will move to that thread as well.

Please note that attaching to legacy contexts (those with a version of 3010 as returned by cuCtxGetApiVersion()) is not possible. The CUDA Runtime will return cudaErrorIncompatibleDriverContext in such cases.

5.26.4 Interactions between CUstream and cudaStream_t

The types CUstream and cudaStream_t are identical and may be used interchangeably.

5.26.5 Interactions between CUevent and cudaEvent_t

The types CUevent and cudaEvent_t are identical and may be used interchangeably.

5.26.6 Interactions between CUarray and cudaArray_t

The types CUarray and struct cudaArray * represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a CUarray in a CUDA Runtime API function which takes a struct cudaArray *, it is necessary to explicitly cast the CUarray to a struct cudaArray *.

In order to use a struct cudaArray * in a CUDA Driver API function which takes a CUarray, it is necessary to explicitly cast the struct cudaArray * to a CUarray .

5.26.7 Interactions between CUgraphicsResource and cudaGraphicsResource_t

The types CUgraphicsResource and cudaGraphicsResource_t represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a CUgraphicsResource in a CUDA Runtime API function which takes a cudaGraphicsResource_t, it is necessary to explicitly cast the CUgraphicsResource to a cudaGraphicsResource_t.

In order to use a cudaGraphicsResource_t in a CUDA Driver API function which takes a CUgraphicsResource, it is necessary to explicitly cast the cudaGraphicsResource_t to a CUgraphicsResource.

5.27 Profiler Control

Functions

- cudaError_t cudaProfilerInitialize (const char *configFile, const char *outputFile, cudaOutputMode_t outputMode)

  Initialize the CUDA profiler.

- cudaError_t cudaProfilerStart (void)

  Enable profiling.

- cudaError_t cudaProfilerStop (void)

  Disable profiling.

5.27.1 Detailed Description

This section describes the profiler control functions of the CUDA runtime application programming interface.

5.27.2 Function Documentation

5.27.2.1 cudaError_t cudaProfilerInitialize (const char * *configFile*, const char * *outputFile*, cudaOutputMode_t *outputMode*)

Using this API user can initialize the CUDA profiler by specifying the configuration file, output file and output file format. This API is generally used to profile different set of counters by looping the kernel launch. The `configFile` parameter can be used to select profiling options including profiler counters. Refer to the "Compute Command Line Profiler User Guide" for supported profiler options and counters.

Limitation: The CUDA profiler cannot be initialized with this API if another profiling tool is already active, as indicated by the cudaErrorProfilerDisabled return code.

Typical usage of the profiling APIs is as follows:

for each set of counters/options

{

cudaProfilerInitialize(); //Initialize profiling,set the counters/options in the config file

...

cudaProfilerStart();

// code to be profiled

cudaProfilerStop();

...

cudaProfilerStart();

// code to be profiled

cudaProfilerStop();

...

}

Parameters:

configFile - Name of the config file that lists the counters/options for profiling.

outputFile - Name of the outputFile where the profiling results will be stored.

outputMode - outputMode, can be cudaKeyValuePair OR cudaCSV.

Returns:

cudaSuccess, cudaErrorInvalidValue, cudaErrorProfilerDisabled

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaProfilerStart, cudaProfilerStop

5.27.2.2 cudaError_t cudaProfilerStart (void)

Enables profile collection by the active profiling tool. If profiling is already enabled, then cudaProfilerStart() has no effect.

cudaProfilerStart and cudaProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaProfilerInitialize, cudaProfilerStop

5.27.2.3 cudaError_t cudaProfilerStop (void)

Disables profile collection by the active profiling tool. If profiling is already disabled, then cudaProfilerStop() has no effect.

cudaProfilerStart and cudaProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.

Returns:

cudaSuccess

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cudaProfilerInitialize, cudaProfilerStart

5.28 Data types used by CUDA Runtime

Data Structures

- struct cudaChannelFormatDesc
- struct cudaDeviceProp
- struct cudaExtent
- struct cudaFuncAttributes
- struct cudaIpcEventHandle_st
- struct cudaIpcMemHandle_st
- struct cudaMemcpy3DParms
- struct cudaMemcpy3DPeerParms
- struct cudaPitchedPtr
- struct cudaPointerAttributes
- struct cudaPos
- struct cudaResourceDesc
- struct cudaResourceViewDesc
- struct cudaTextureDesc
- struct surfaceReference
- struct textureReference

Defines

- #define CUDA_IPC_HANDLE_SIZE 64
- #define cudaArrayCubemap 0x04
- #define cudaArrayDefault 0x00
- #define cudaArrayLayered 0x01
- #define cudaArraySurfaceLoadStore 0x02
- #define cudaArrayTextureGather 0x08
- #define cudaDeviceBlockingSync 0x04
- #define cudaDeviceLmemResizeToMax 0x10
- #define cudaDeviceMapHost 0x08
- #define cudaDeviceMask 0x1f
- #define cudaDevicePropDontCare
- #define cudaDeviceScheduleAuto 0x00
- #define cudaDeviceScheduleBlockingSync 0x04
- #define cudaDeviceScheduleMask 0x07
- #define cudaDeviceScheduleSpin 0x01
- #define cudaDeviceScheduleYield 0x02
- #define cudaEventBlockingSync 0x01
- #define cudaEventDefault 0x00
- #define cudaEventDisableTiming 0x02
- #define cudaEventInterprocess 0x04
- #define cudaHostAllocDefault 0x00
- #define cudaHostAllocMapped 0x02
- #define cudaHostAllocPortable 0x01
- #define cudaHostAllocWriteCombined 0x04
- #define cudaHostRegisterDefault 0x00
- #define cudaHostRegisterMapped 0x02
- #define cudaHostRegisterPortable 0x01

- #define cudaIpcMemLazyEnablePeerAccess 0x01
- #define cudaPeerAccessDefault 0x00
- #define cudaStreamDefault 0x00
- #define cudaStreamNonBlocking 0x01

Typedefs

- typedef struct cudaArray * cudaArray_const_t
- typedef struct cudaArray * cudaArray_t
- typedef enum cudaError cudaError_t
- typedef struct CUevent_st * cudaEvent_t
- typedef struct cudaGraphicsResource * cudaGraphicsResource_t
- typedef struct cudaIpcEventHandle_st cudaIpcEventHandle_t
- typedef struct cudaIpcMemHandle_st cudaIpcMemHandle_t
- typedef struct cudaMipmappedArray * cudaMipmappedArray_const_t
- typedef struct cudaMipmappedArray * cudaMipmappedArray_t
- typedef enum cudaOutputMode cudaOutputMode_t
- typedef struct CUstream_st * cudaStream_t
- typedef unsigned long long cudaSurfaceObject_t
- typedef unsigned long long cudaTextureObject_t
- typedef struct CUuuid_st cudaUUID_t

Enumerations

- enum cudaChannelFormatKind {
  cudaChannelFormatKindSigned = 0,
  cudaChannelFormatKindUnsigned = 1,
  cudaChannelFormatKindFloat = 2,
  cudaChannelFormatKindNone = 3 }
- enum cudaComputeMode {
  cudaComputeModeDefault = 0,
  cudaComputeModeExclusive = 1,
  cudaComputeModeProhibited = 2,
  cudaComputeModeExclusiveProcess = 3 }
- enum cudaDeviceAttr {
  cudaDevAttrMaxThreadsPerBlock = 1,
  cudaDevAttrMaxBlockDimX = 2,
  cudaDevAttrMaxBlockDimY = 3,
  cudaDevAttrMaxBlockDimZ = 4,
  cudaDevAttrMaxGridDimX = 5,
  cudaDevAttrMaxGridDimY = 6,
  cudaDevAttrMaxGridDimZ = 7,
  cudaDevAttrMaxSharedMemoryPerBlock = 8,
  cudaDevAttrTotalConstantMemory = 9,
  cudaDevAttrWarpSize = 10,

cudaDevAttrMaxPitch = 11,
cudaDevAttrMaxRegistersPerBlock = 12,
cudaDevAttrClockRate = 13,
cudaDevAttrTextureAlignment = 14,
cudaDevAttrGpuOverlap = 15,
cudaDevAttrMultiProcessorCount = 16,
cudaDevAttrKernelExecTimeout = 17,
cudaDevAttrIntegrated = 18,
cudaDevAttrCanMapHostMemory = 19,
cudaDevAttrComputeMode = 20,
cudaDevAttrMaxTexture1DWidth = 21,
cudaDevAttrMaxTexture2DWidth = 22,
cudaDevAttrMaxTexture2DHeight = 23,
cudaDevAttrMaxTexture3DWidth = 24,
cudaDevAttrMaxTexture3DHeight = 25,
cudaDevAttrMaxTexture3DDepth = 26,
cudaDevAttrMaxTexture2DLayeredWidth = 27,
cudaDevAttrMaxTexture2DLayeredHeight = 28,
cudaDevAttrMaxTexture2DLayeredLayers = 29,
cudaDevAttrSurfaceAlignment = 30,
cudaDevAttrConcurrentKernels = 31,
cudaDevAttrEccEnabled = 32,
cudaDevAttrPciBusId = 33,
cudaDevAttrPciDeviceId = 34,
cudaDevAttrTccDriver = 35,
cudaDevAttrMemoryClockRate = 36,
cudaDevAttrGlobalMemoryBusWidth = 37,
cudaDevAttrL2CacheSize = 38,
cudaDevAttrMaxThreadsPerMultiProcessor = 39,
cudaDevAttrAsyncEngineCount = 40,
cudaDevAttrUnifiedAddressing = 41,
cudaDevAttrMaxTexture1DLayeredWidth = 42,
cudaDevAttrMaxTexture1DLayeredLayers = 43,
cudaDevAttrMaxTexture2DGatherWidth = 45,
cudaDevAttrMaxTexture2DGatherHeight = 46,
cudaDevAttrMaxTexture3DWidthAlt = 47,
cudaDevAttrMaxTexture3DHeightAlt = 48,
cudaDevAttrMaxTexture3DDepthAlt = 49,
cudaDevAttrPciDomainId = 50,
cudaDevAttrTexturePitchAlignment = 51,
cudaDevAttrMaxTextureCubemapWidth = 52,

cudaDevAttrMaxTextureCubemapLayeredWidth = 53,
cudaDevAttrMaxTextureCubemapLayeredLayers = 54,
cudaDevAttrMaxSurface1DWidth = 55,
cudaDevAttrMaxSurface2DWidth = 56,
cudaDevAttrMaxSurface2DHeight = 57,
cudaDevAttrMaxSurface3DWidth = 58,
cudaDevAttrMaxSurface3DHeight = 59,
cudaDevAttrMaxSurface3DDepth = 60,
cudaDevAttrMaxSurface1DLayeredWidth = 61,
cudaDevAttrMaxSurface1DLayeredLayers = 62,
cudaDevAttrMaxSurface2DLayeredWidth = 63,
cudaDevAttrMaxSurface2DLayeredHeight = 64,
cudaDevAttrMaxSurface2DLayeredLayers = 65,
cudaDevAttrMaxSurfaceCubemapWidth = 66,
cudaDevAttrMaxSurfaceCubemapLayeredWidth = 67,
cudaDevAttrMaxSurfaceCubemapLayeredLayers = 68,
cudaDevAttrMaxTexture1DLinearWidth = 69,
cudaDevAttrMaxTexture2DLinearWidth = 70,
cudaDevAttrMaxTexture2DLinearHeight = 71,
cudaDevAttrMaxTexture2DLinearPitch = 72,
cudaDevAttrMaxTexture2DMipmappedWidth = 73,
cudaDevAttrMaxTexture2DMipmappedHeight = 74,
cudaDevAttrComputeCapabilityMajor = 75,
cudaDevAttrComputeCapabilityMinor = 76,
cudaDevAttrMaxTexture1DMipmappedWidth = 77 }

- enum cudaError {
  cudaSuccess = 0,
  cudaErrorMissingConfiguration = 1,
  cudaErrorMemoryAllocation = 2,
  cudaErrorInitializationError = 3,
  cudaErrorLaunchFailure = 4,
  cudaErrorPriorLaunchFailure = 5,
  cudaErrorLaunchTimeout = 6,
  cudaErrorLaunchOutOfResources = 7,
  cudaErrorInvalidDeviceFunction = 8,
  cudaErrorInvalidConfiguration = 9,
  cudaErrorInvalidDevice = 10,
  cudaErrorInvalidValue = 11,
  cudaErrorInvalidPitchValue = 12,
  cudaErrorInvalidSymbol = 13,
  cudaErrorMapBufferObjectFailed = 14,

cudaErrorUnmapBufferObjectFailed = 15,
cudaErrorInvalidHostPointer = 16,
cudaErrorInvalidDevicePointer = 17,
cudaErrorInvalidTexture = 18,
cudaErrorInvalidTextureBinding = 19,
cudaErrorInvalidChannelDescriptor = 20,
cudaErrorInvalidMemcpyDirection = 21,
cudaErrorAddressOfConstant = 22,
cudaErrorTextureFetchFailed = 23,
cudaErrorTextureNotBound = 24,
cudaErrorSynchronizationError = 25,
cudaErrorInvalidFilterSetting = 26,
cudaErrorInvalidNormSetting = 27,
cudaErrorMixedDeviceExecution = 28,
cudaErrorCudartUnloading = 29,
cudaErrorUnknown = 30,
cudaErrorNotYetImplemented = 31,
cudaErrorMemoryValueTooLarge = 32,
cudaErrorInvalidResourceHandle = 33,
cudaErrorNotReady = 34,
cudaErrorInsufficientDriver = 35,
cudaErrorSetOnActiveProcess = 36,
cudaErrorInvalidSurface = 37,
cudaErrorNoDevice = 38,
cudaErrorECCUncorrectable = 39,
cudaErrorSharedObjectSymbolNotFound = 40,
cudaErrorSharedObjectInitFailed = 41,
cudaErrorUnsupportedLimit = 42,
cudaErrorDuplicateVariableName = 43,
cudaErrorDuplicateTextureName = 44,
cudaErrorDuplicateSurfaceName = 45,
cudaErrorDevicesUnavailable = 46,
cudaErrorInvalidKernelImage = 47,
cudaErrorNoKernelImageForDevice = 48,
cudaErrorIncompatibleDriverContext = 49,
cudaErrorPeerAccessAlreadyEnabled = 50,
cudaErrorPeerAccessNotEnabled = 51,
cudaErrorDeviceAlreadyInUse = 54,
cudaErrorProfilerDisabled = 55,
cudaErrorProfilerNotInitialized = 56,
cudaErrorProfilerAlreadyStarted = 57,

cudaErrorProfilerAlreadyStopped = 58,
cudaErrorAssert = 59,
cudaErrorTooManyPeers = 60,
cudaErrorHostMemoryAlreadyRegistered = 61,
cudaErrorHostMemoryNotRegistered = 62,
cudaErrorOperatingSystem = 63,
cudaErrorPeerAccessUnsupported = 64,
cudaErrorLaunchMaxDepthExceeded = 65,
cudaErrorLaunchFileScopedTex = 66,
cudaErrorLaunchFileScopedSurf = 67,
cudaErrorSyncDepthExceeded = 68,
cudaErrorLaunchPendingCountExceeded = 69,
cudaErrorNotPermitted = 70,
cudaErrorNotSupported = 71,
cudaErrorStartupFailure = 0x7f,
cudaErrorApiFailureBase = 10000 }

- enum cudaFuncCache {
  cudaFuncCachePreferNone = 0,
  cudaFuncCachePreferShared = 1,
  cudaFuncCachePreferL1 = 2,
  cudaFuncCachePreferEqual = 3 }
- enum cudaGraphicsCubeFace {
  cudaGraphicsCubeFacePositiveX = 0x00,
  cudaGraphicsCubeFaceNegativeX = 0x01,
  cudaGraphicsCubeFacePositiveY = 0x02,
  cudaGraphicsCubeFaceNegativeY = 0x03,
  cudaGraphicsCubeFacePositiveZ = 0x04,
  cudaGraphicsCubeFaceNegativeZ = 0x05 }
- enum cudaGraphicsMapFlags {
  cudaGraphicsMapFlagsNone = 0,
  cudaGraphicsMapFlagsReadOnly = 1,
  cudaGraphicsMapFlagsWriteDiscard = 2 }
- enum cudaGraphicsRegisterFlags {
  cudaGraphicsRegisterFlagsNone = 0,
  cudaGraphicsRegisterFlagsReadOnly = 1,
  cudaGraphicsRegisterFlagsWriteDiscard = 2,
  cudaGraphicsRegisterFlagsSurfaceLoadStore = 4,
  cudaGraphicsRegisterFlagsTextureGather = 8 }
- enum cudaLimit {
  cudaLimitStackSize = 0x00,
  cudaLimitPrintfFifoSize = 0x01,
  cudaLimitMallocHeapSize = 0x02,
  cudaLimitDevRuntimeSyncDepth = 0x03,
  cudaLimitDevRuntimePendingLaunchCount = 0x04 }

- enum cudaMemcpyKind {
  cudaMemcpyHostToHost = 0,
  cudaMemcpyHostToDevice = 1,
  cudaMemcpyDeviceToHost = 2,
  cudaMemcpyDeviceToDevice = 3,
  cudaMemcpyDefault = 4 }
- enum cudaMemoryType {
  cudaMemoryTypeHost = 1,
  cudaMemoryTypeDevice = 2 }
- enum cudaOutputMode {
  cudaKeyValuePair = 0x00,
  cudaCSV = 0x01 }
- enum cudaResourceType {
  cudaResourceTypeArray = 0x00,
  cudaResourceTypeMipmappedArray = 0x01,
  cudaResourceTypeLinear = 0x02,
  cudaResourceTypePitch2D = 0x03 }
- enum cudaResourceViewFormat {
  cudaResViewFormatNone = 0x00,
  cudaResViewFormatUnsignedChar1 = 0x01,
  cudaResViewFormatUnsignedChar2 = 0x02,
  cudaResViewFormatUnsignedChar4 = 0x03,
  cudaResViewFormatSignedChar1 = 0x04,
  cudaResViewFormatSignedChar2 = 0x05,
  cudaResViewFormatSignedChar4 = 0x06,
  cudaResViewFormatUnsignedShort1 = 0x07,
  cudaResViewFormatUnsignedShort2 = 0x08,
  cudaResViewFormatUnsignedShort4 = 0x09,
  cudaResViewFormatSignedShort1 = 0x0a,
  cudaResViewFormatSignedShort2 = 0x0b,
  cudaResViewFormatSignedShort4 = 0x0c,
  cudaResViewFormatUnsignedInt1 = 0x0d,
  cudaResViewFormatUnsignedInt2 = 0x0e,
  cudaResViewFormatUnsignedInt4 = 0x0f,
  cudaResViewFormatSignedInt1 = 0x10,
  cudaResViewFormatSignedInt2 = 0x11,
  cudaResViewFormatSignedInt4 = 0x12,
  cudaResViewFormatHalf1 = 0x13,
  cudaResViewFormatHalf2 = 0x14,
  cudaResViewFormatHalf4 = 0x15,
  cudaResViewFormatFloat1 = 0x16,
  cudaResViewFormatFloat2 = 0x17,

cudaResViewFormatFloat4 = 0x18,
cudaResViewFormatUnsignedBlockCompressed1 = 0x19,
cudaResViewFormatUnsignedBlockCompressed2 = 0x1a,
cudaResViewFormatUnsignedBlockCompressed3 = 0x1b,
cudaResViewFormatUnsignedBlockCompressed4 = 0x1c,
cudaResViewFormatSignedBlockCompressed4 = 0x1d,
cudaResViewFormatUnsignedBlockCompressed5 = 0x1e,
cudaResViewFormatSignedBlockCompressed5 = 0x1f,
cudaResViewFormatUnsignedBlockCompressed6H = 0x20,
cudaResViewFormatSignedBlockCompressed6H = 0x21,
cudaResViewFormatUnsignedBlockCompressed7 = 0x22 }

- enum cudaSharedMemConfig
- enum cudaSurfaceBoundaryMode {
  cudaBoundaryModeZero = 0,
  cudaBoundaryModeClamp = 1,
  cudaBoundaryModeTrap = 2 }
- enum cudaSurfaceFormatMode {
  cudaFormatModeForced = 0,
  cudaFormatModeAuto = 1 }
- enum cudaTextureAddressMode {
  cudaAddressModeWrap = 0,
  cudaAddressModeClamp = 1,
  cudaAddressModeMirror = 2,
  cudaAddressModeBorder = 3 }
- enum cudaTextureFilterMode {
  cudaFilterModePoint = 0,
  cudaFilterModeLinear = 1 }
- enum cudaTextureReadMode {
  cudaReadModeElementType = 0,
  cudaReadModeNormalizedFloat = 1 }

5.28.1 Define Documentation

5.28.1.1 #define CUDA_IPC_HANDLE_SIZE 64

CUDA IPC Handle Size

5.28.1.2 #define cudaArrayCubemap 0x04

Must be set in cudaMalloc3DArray to create a cubemap CUDA array

5.28.1.3 #define cudaArrayDefault 0x00

Default CUDA array allocation flag

5.28.1.4 #define cudaArrayLayered 0x01

Must be set in cudaMalloc3DArray to create a layered CUDA array

5.28.1.5 #define cudaArraySurfaceLoadStore 0x02

Must be set in cudaMallocArray or cudaMalloc3DArray in order to bind surfaces to the CUDA array

5.28.1.6 #define cudaArrayTextureGather 0x08

Must be set in cudaMallocArray or cudaMalloc3DArray in order to perform texture gather operations on the CUDA array

5.28.1.7 #define cudaDeviceBlockingSync 0x04

Device flag - Use blocking synchronization

Deprecated

This flag was deprecated as of CUDA 4.0 and replaced with cudaDeviceScheduleBlockingSync.

5.28.1.8 #define cudaDeviceLmemResizeToMax 0x10

Device flag - Keep local memory allocation after launch

5.28.1.9 #define cudaDeviceMapHost 0x08

Device flag - Support mapped pinned allocations

5.28.1.10 #define cudaDeviceMask 0x1f

Device flags mask

5.28.1.11 #define cudaDevicePropDontCare

Empty device properties

5.28.1.12 #define cudaDeviceScheduleAuto 0x00

Device flag - Automatic scheduling

5.28.1.13 #define cudaDeviceScheduleBlockingSync 0x04

Device flag - Use blocking synchronization

5.28.1.14 #define cudaDeviceScheduleMask 0x07

Device schedule flags mask

5.28.1.15 #define cudaDeviceScheduleSpin 0x01

Device flag - Spin default scheduling

5.28.1.16 #define cudaDeviceScheduleYield 0x02

Device flag - Yield default scheduling

5.28.1.17 #define cudaEventBlockingSync 0x01

Event uses blocking synchronization

5.28.1.18 #define cudaEventDefault 0x00

Default event flag

5.28.1.19 #define cudaEventDisableTiming 0x02

Event will not record timing data

5.28.1.20 #define cudaEventInterprocess 0x04

Event is suitable for interprocess use. cudaEventDisableTiming must be set

5.28.1.21 #define cudaHostAllocDefault 0x00

Default page-locked allocation flag

5.28.1.22 #define cudaHostAllocMapped 0x02

Map allocation into device space

5.28.1.23 #define cudaHostAllocPortable 0x01

Pinned memory accessible by all CUDA contexts

5.28.1.24 #define cudaHostAllocWriteCombined 0x04

Write-combined memory

5.28.1.25 #define cudaHostRegisterDefault 0x00

Default host memory registration flag

5.28.1.26 #define cudaHostRegisterMapped 0x02

Map registered memory into device space

5.28.1.27 #define cudaHostRegisterPortable 0x01

Pinned memory accessible by all CUDA contexts

5.28.1.28 #define cudaIpcMemLazyEnablePeerAccess 0x01

Automatically enable peer access between remote devices as needed

5.28.1.29 #define cudaPeerAccessDefault 0x00

Default peer addressing enable flag

5.28.1.30 #define cudaStreamDefault 0x00

Default stream flag

5.28.1.31 #define cudaStreamNonBlocking 0x01

Stream does not synchronize with stream 0 (the NULL stream)

5.28.2 Typedef Documentation

5.28.2.1 typedef struct cudaArray* cudaArray_const_t

CUDA array (as source copy argument)

5.28.2.2 typedef struct cudaArray* cudaArray_t

CUDA array

5.28.2.3 typedef enum cudaError cudaError_t

CUDA Error types

5.28.2.4 typedef struct CUevent_st* cudaEvent_t

CUDA event types

5.28.2.5 typedef struct cudaGraphicsResource* cudaGraphicsResource_t

CUDA graphics resource types

5.28.2.6 typedef struct cudaIpcEventHandle_st cudaIpcEventHandle_t

CUDA IPC event handle

5.28.2.7 typedef struct cudaIpcMemHandle_st cudaIpcMemHandle_t

CUDA IPC memory handle

5.28.2.8 typedef struct cudaMipmappedArray* cudaMipmappedArray_const_t

CUDA mipmapped array (as source argument)

5.28.2.9 typedef struct cudaMipmappedArray* cudaMipmappedArray_t

CUDA mipmapped array

5.28.2.10 typedef enum cudaOutputMode cudaOutputMode_t

CUDA output file modes

5.28.2.11 typedef struct CUstream_st* cudaStream_t

CUDA stream

5.28.2.12 typedef unsigned long long cudaSurfaceObject_t

CUDA Surface object

5.28.2.13 typedef unsigned long long cudaTextureObject_t

CUDA texture object

5.28.2.14 typedef struct CUuuid_st cudaUUID_t

CUDA UUID types

5.28.3 Enumeration Type Documentation

5.28.3.1 enum cudaChannelFormatKind

Channel format kind

Enumerator:

cudaChannelFormatKindSigned Signed channel format
cudaChannelFormatKindUnsigned Unsigned channel format
cudaChannelFormatKindFloat Float channel format
cudaChannelFormatKindNone No channel format

5.28.3.2 enum cudaComputeMode

CUDA device compute modes

Enumerator:

cudaComputeModeDefault Default compute mode (Multiple threads can use cudaSetDevice() with this device)

cudaComputeModeExclusive Compute-exclusive-thread mode (Only one thread in one process will be able to use cudaSetDevice() with this device)

cudaComputeModeProhibited Compute-prohibited mode (No threads can use cudaSetDevice() with this device)

cudaComputeModeExclusiveProcess Compute-exclusive-process mode (Many threads in one process will be able to use cudaSetDevice() with this device)

5.28.3.3 enum cudaDeviceAttr

CUDA device attributes

Enumerator:

cudaDevAttrMaxThreadsPerBlock Maximum number of threads per block

cudaDevAttrMaxBlockDimX Maximum block dimension X

cudaDevAttrMaxBlockDimY Maximum block dimension Y

cudaDevAttrMaxBlockDimZ Maximum block dimension Z

cudaDevAttrMaxGridDimX Maximum grid dimension X

cudaDevAttrMaxGridDimY Maximum grid dimension Y

cudaDevAttrMaxGridDimZ Maximum grid dimension Z

cudaDevAttrMaxSharedMemoryPerBlock Maximum shared memory available per block in bytes

cudaDevAttrTotalConstantMemory Memory available on device for __constant__ variables in a CUDA C kernel in bytes

cudaDevAttrWarpSize Warp size in threads

cudaDevAttrMaxPitch Maximum pitch in bytes allowed by memory copies

cudaDevAttrMaxRegistersPerBlock Maximum number of 32-bit registers available per block

cudaDevAttrClockRate Peak clock frequency in kilohertz

cudaDevAttrTextureAlignment Alignment requirement for textures

cudaDevAttrGpuOverlap Device can possibly copy memory and execute a kernel concurrently

cudaDevAttrMultiProcessorCount Number of multiprocessors on device

cudaDevAttrKernelExecTimeout Specifies whether there is a run time limit on kernels

cudaDevAttrIntegrated Device is integrated with host memory

cudaDevAttrCanMapHostMemory Device can map host memory into CUDA address space

cudaDevAttrComputeMode Compute mode (See cudaComputeMode for details)

cudaDevAttrMaxTexture1DWidth Maximum 1D texture width

cudaDevAttrMaxTexture2DWidth Maximum 2D texture width

cudaDevAttrMaxTexture2DHeight Maximum 2D texture height

cudaDevAttrMaxTexture3DWidth Maximum 3D texture width

cudaDevAttrMaxTexture3DHeight Maximum 3D texture height

cudaDevAttrMaxTexture3DDepth Maximum 3D texture depth

cudaDevAttrMaxTexture2DLayeredWidth Maximum 2D layered texture width

cudaDevAttrMaxTexture2DLayeredHeight Maximum 2D layered texture height

cudaDevAttrMaxTexture2DLayeredLayers Maximum layers in a 2D layered texture

cudaDevAttrSurfaceAlignment Alignment requirement for surfaces

cudaDevAttrConcurrentKernels Device can possibly execute multiple kernels concurrently

cudaDevAttrEccEnabled Device has ECC support enabled

cudaDevAttrPciBusId PCI bus ID of the device

cudaDevAttrPciDeviceId PCI device ID of the device

cudaDevAttrTccDriver Device is using TCC driver model

cudaDevAttrMemoryClockRate Peak memory clock frequency in kilohertz

cudaDevAttrGlobalMemoryBusWidth Global memory bus width in bits

cudaDevAttrL2CacheSize Size of L2 cache in bytes

cudaDevAttrMaxThreadsPerMultiProcessor Maximum resident threads per multiprocessor

cudaDevAttrAsyncEngineCount Number of asynchronous engines

cudaDevAttrUnifiedAddressing Device shares a unified address space with the host

cudaDevAttrMaxTexture1DLayeredWidth Maximum 1D layered texture width

cudaDevAttrMaxTexture1DLayeredLayers Maximum layers in a 1D layered texture

cudaDevAttrMaxTexture2DGatherWidth Maximum 2D texture width if cudaArrayTextureGather is set

cudaDevAttrMaxTexture2DGatherHeight Maximum 2D texture height if cudaArrayTextureGather is set

cudaDevAttrMaxTexture3DWidthAlt Alternate maximum 3D texture width

cudaDevAttrMaxTexture3DHeightAlt Alternate maximum 3D texture height

cudaDevAttrMaxTexture3DDepthAlt Alternate maximum 3D texture depth

cudaDevAttrPciDomainId PCI domain ID of the device

cudaDevAttrTexturePitchAlignment Pitch alignment requirement for textures

cudaDevAttrMaxTextureCubemapWidth Maximum cubemap texture width/height

cudaDevAttrMaxTextureCubemapLayeredWidth Maximum cubemap layered texture width/height

cudaDevAttrMaxTextureCubemapLayeredLayers Maximum layers in a cubemap layered texture

cudaDevAttrMaxSurface1DWidth Maximum 1D surface width

cudaDevAttrMaxSurface2DWidth Maximum 2D surface width

cudaDevAttrMaxSurface2DHeight Maximum 2D surface height

cudaDevAttrMaxSurface3DWidth Maximum 3D surface width

cudaDevAttrMaxSurface3DHeight Maximum 3D surface height

cudaDevAttrMaxSurface3DDepth Maximum 3D surface depth

cudaDevAttrMaxSurface1DLayeredWidth Maximum 1D layered surface width

cudaDevAttrMaxSurface1DLayeredLayers Maximum layers in a 1D layered surface

cudaDevAttrMaxSurface2DLayeredWidth Maximum 2D layered surface width

cudaDevAttrMaxSurface2DLayeredHeight Maximum 2D layered surface height

cudaDevAttrMaxSurface2DLayeredLayers Maximum layers in a 2D layered surface

cudaDevAttrMaxSurfaceCubemapWidth Maximum cubemap surface width

cudaDevAttrMaxSurfaceCubemapLayeredWidth Maximum cubemap layered surface width

cudaDevAttrMaxSurfaceCubemapLayeredLayers Maximum layers in a cubemap layered surface

cudaDevAttrMaxTexture1DLinearWidth Maximum 1D linear texture width

cudaDevAttrMaxTexture2DLinearWidth Maximum 2D linear texture width

cudaDevAttrMaxTexture2DLinearHeight Maximum 2D linear texture height

cudaDevAttrMaxTexture2DLinearPitch Maximum 2D linear texture pitch in bytes

cudaDevAttrMaxTexture2DMipmappedWidth Maximum mipmapped 2D texture width

cudaDevAttrMaxTexture2DMipmappedHeight Maximum mipmapped 2D texture height

cudaDevAttrComputeCapabilityMajor Major compute capability version number

cudaDevAttrComputeCapabilityMinor Minor compute capability version number

cudaDevAttrMaxTexture1DMipmappedWidth Maximum mipmapped 1D texture width

5.28.3.4 enum cudaError

CUDA error types

Enumerator:

cudaSuccess The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see cudaEventQuery() and cudaStreamQuery()).

cudaErrorMissingConfiguration The device function being invoked (usually via cudaLaunch()) was not previously configured via the cudaConfigureCall() function.

cudaErrorMemoryAllocation The API call failed because it was unable to allocate enough memory to perform the requested operation.

cudaErrorInitializationError The API call failed because the CUDA driver and runtime could not be initialized.

cudaErrorLaunchFailure An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. The device cannot be used until cudaThreadExit() is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.

cudaErrorPriorLaunchFailure This indicated that a previous kernel launch failed. This was previously used for device emulation of kernel launches.

Deprecated

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

cudaErrorLaunchTimeout This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device property kernelExecTimeoutEnabled for more information. The device cannot be used until cudaThreadExit() is called. All existing device memory allocations are invalid and must be reconstructed if the program is to continue using CUDA.

cudaErrorLaunchOutOfResources This indicates that a launch did not occur because it did not have appropriate resources. Although this error is similar to cudaErrorInvalidConfiguration, this error usually indicates that the user has attempted to pass too many arguments to the device kernel, or the kernel launch specifies too many threads for the kernel's register count.

cudaErrorInvalidDeviceFunction The requested device function does not exist or is not compiled for the proper device architecture.

cudaErrorInvalidConfiguration This indicates that a kernel launch is requesting resources that can never be satisfied by the current device. Requesting more shared memory per block than the device supports will trigger this error, as will requesting too many threads or blocks. See cudaDeviceProp for more device limitations.

cudaErrorInvalidDevice This indicates that the device ordinal supplied by the user does not correspond to a valid CUDA device.

cudaErrorInvalidValue This indicates that one or more of the parameters passed to the API call is not within an acceptable range of values.

cudaErrorInvalidPitchValue This indicates that one or more of the pitch-related parameters passed to the API call is not within the acceptable range for pitch.

cudaErrorInvalidSymbol This indicates that the symbol name/identifier passed to the API call is not a valid name or identifier.

cudaErrorMapBufferObjectFailed This indicates that the buffer object could not be mapped.

cudaErrorUnmapBufferObjectFailed This indicates that the buffer object could not be unmapped.

cudaErrorInvalidHostPointer This indicates that at least one host pointer passed to the API call is not a valid host pointer.

cudaErrorInvalidDevicePointer This indicates that at least one device pointer passed to the API call is not a valid device pointer.

cudaErrorInvalidTexture This indicates that the texture passed to the API call is not a valid texture.

cudaErrorInvalidTextureBinding This indicates that the texture binding is not valid. This occurs if you call cudaGetTextureAlignmentOffset() with an unbound texture.

cudaErrorInvalidChannelDescriptor This indicates that the channel descriptor passed to the API call is not valid. This occurs if the format is not one of the formats specified by cudaChannelFormatKind, or if one of the dimensions is invalid.

cudaErrorInvalidMemcpyDirection This indicates that the direction of the memcpy passed to the API call is not one of the types specified by cudaMemcpyKind.

cudaErrorAddressOfConstant This indicated that the user has taken the address of a constant variable, which was forbidden up until the CUDA 3.1 release.

Deprecated

This error return is deprecated as of CUDA 3.1. Variables in constant memory may now have their address taken by the runtime via cudaGetSymbolAddress().

cudaErrorTextureFetchFailed This indicated that a texture fetch was not able to be performed. This was previously used for device emulation of texture operations.

Deprecated

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

cudaErrorTextureNotBound This indicated that a texture was not bound for access. This was previously used for device emulation of texture operations.

Deprecated

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

cudaErrorSynchronizationError This indicated that a synchronization operation had failed. This was previously used for some device emulation functions.

Deprecated

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

cudaErrorInvalidFilterSetting This indicates that a non-float texture was being accessed with linear filtering. This is not supported by CUDA.

cudaErrorInvalidNormSetting This indicates that an attempt was made to read a non-float texture as a normalized float. This is not supported by CUDA.

cudaErrorMixedDeviceExecution Mixing of device and device emulation code was not allowed.

Deprecated

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

cudaErrorCudartUnloading This indicates that a CUDA Runtime API call cannot be executed because it is being called during process shut down, at a point in time after CUDA driver has been unloaded.

cudaErrorUnknown This indicates that an unknown internal error has occurred.

cudaErrorNotYetImplemented This indicates that the API call is not yet implemented. Production releases of CUDA will never return this error.

Deprecated

This error return is deprecated as of CUDA 4.1.

cudaErrorMemoryValueTooLarge This indicated that an emulated device pointer exceeded the 32-bit address range.

Deprecated

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

cudaErrorInvalidResourceHandle This indicates that a resource handle passed to the API call was not valid. Resource handles are opaque types like cudaStream_t and cudaEvent_t.

cudaErrorNotReady This indicates that asynchronous operations issued previously have not completed yet. This result is not actually an error, but must be indicated differently than cudaSuccess (which indicates completion). Calls that may return this value include cudaEventQuery() and cudaStreamQuery().

cudaErrorInsufficientDriver This indicates that the installed NVIDIA CUDA driver is older than the CUDA runtime library. This is not a supported configuration. Users should install an updated NVIDIA display driver to allow the application to run.

cudaErrorSetOnActiveProcess This indicates that the user has called cudaSetValidDevices(), cudaSetDeviceFlags(), cudaD3D9SetDirect3DDevice(), cudaD3D10SetDirect3DDevice, cudaD3D11SetDirect3DDevice(), or cudaVDPAUSetVDPAUDevice() after initializing the CUDA runtime by calling non-device management operations (allocating memory and launching kernels are examples of non-device management operations). This error can also be returned if using runtime/driver interoperability and there is an existing CUcontext active on the host thread.

cudaErrorInvalidSurface This indicates that the surface passed to the API call is not a valid surface.

cudaErrorNoDevice This indicates that no CUDA-capable devices were detected by the installed CUDA driver.

cudaErrorECCUncorrectable This indicates that an uncorrectable ECC error was detected during execution.

cudaErrorSharedObjectSymbolNotFound This indicates that a link to a shared object failed to resolve.

cudaErrorSharedObjectInitFailed This indicates that initialization of a shared object failed.

cudaErrorUnsupportedLimit This indicates that the cudaLimit passed to the API call is not supported by the active device.

cudaErrorDuplicateVariableName This indicates that multiple global or constant variables (across separate CUDA source files in the application) share the same string name.

cudaErrorDuplicateTextureName This indicates that multiple textures (across separate CUDA source files in the application) share the same string name.

cudaErrorDuplicateSurfaceName This indicates that multiple surfaces (across separate CUDA source files in the application) share the same string name.

cudaErrorDevicesUnavailable This indicates that all CUDA devices are busy or unavailable at the current time. Devices are often busy/unavailable due to use of cudaComputeModeExclusive, cudaComputeModeProhibited or when long running CUDA kernels have filled up the GPU and are blocking new work from starting. They can also be unavailable due to memory constraints on a device that already has active CUDA work being performed.

cudaErrorInvalidKernelImage This indicates that the device kernel image is invalid.

cudaErrorNoKernelImageForDevice This indicates that there is no kernel image available that is suitable for the device. This can occur when a user specifies code generation options for a particular CUDA source file that do not include the corresponding device configuration.

cudaErrorIncompatibleDriverContext This indicates that the current context is not compatible with this the CUDA Runtime. This can only occur if you are using CUDA Runtime/Driver interoperability and have created an existing Driver context using the driver API. The Driver context may be incompatible either because the Driver context was created using an older version of the API, because the Runtime API call expects a primary driver context and the Driver context is not primary, or because the Driver context has been destroyed. Please see Interactions with the CUDA Driver API" for more information.

cudaErrorPeerAccessAlreadyEnabled This error indicates that a call to cudaDeviceEnablePeerAccess() is trying to re-enable peer addressing on from a context which has already had peer addressing enabled.

cudaErrorPeerAccessNotEnabled This error indicates that cudaDeviceDisablePeerAccess() is trying to disable peer addressing which has not been enabled yet via cudaDeviceEnablePeerAccess().

cudaErrorDeviceAlreadyInUse This indicates that a call tried to access an exclusive-thread device that is already in use by a different thread.

cudaErrorProfilerDisabled This indicates profiler is not initialized for this run. This can happen when the application is running with external profiling tools like visual profiler.

Deprecated

cudaErrorProfilerNotInitialized This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via cudaProfilerStart or cudaProfilerStop without initialization.

Deprecated

cudaErrorProfilerAlreadyStarted This error return is deprecated as of CUDA 5.0. It is no longer an error to call cudaProfilerStart() when profiling is already enabled.

Deprecated

cudaErrorProfilerAlreadyStopped This error return is deprecated as of CUDA 5.0. It is no longer an error to call cudaProfilerStop() when profiling is already disabled.

cudaErrorAssert An assert triggered in device code during kernel execution. The device cannot be used again until cudaThreadExit() is called. All existing allocations are invalid and must be reconstructed if the program is to continue using CUDA.

cudaErrorTooManyPeers This error indicates that the hardware resources required to enable peer access have been exhausted for one or more of the devices passed to cudaEnablePeerAccess().

cudaErrorHostMemoryAlreadyRegistered This error indicates that the memory range passed to cudaHostRegister() has already been registered.

cudaErrorHostMemoryNotRegistered This error indicates that the pointer passed to cudaHostUnregister() does not correspond to any currently registered memory region.

cudaErrorOperatingSystem This error indicates that an OS call failed.

cudaErrorPeerAccessUnsupported This error indicates that P2P access is not supported across the given devices.

cudaErrorLaunchMaxDepthExceeded This error indicates that a device runtime grid launch did not occur because the depth of the child grid would exceed the maximum supported number of nested grid launches.

cudaErrorLaunchFileScopedTex This error indicates that a grid launch did not occur because the kernel uses file-scoped textures which are unsupported by the device runtime. Kernels launched via the device runtime only support textures created with the Texture Object API's.

cudaErrorLaunchFileScopedSurf This error indicates that a grid launch did not occur because the kernel uses file-scoped surfaces which are unsupported by the device runtime. Kernels launched via the device runtime only support surfaces created with the Surface Object API's.

cudaErrorSyncDepthExceeded This error indicates that a call to cudaDeviceSynchronize made from the device runtime failed because the call was made at grid depth greater than than either the default (2 levels of grids) or user specified device limit cudaLimitDevRuntimeSyncDepth. To be able to synchronize on launched grids at a greater depth successfully, the maximum nested depth at which cudaDeviceSynchronize will be called must be specified with the cudaLimitDevRuntimeSyncDepth limit to the cudaDeviceSetLimit api before the host-side launch of a kernel using the device runtime. Keep in mind that additional levels of sync depth require the runtime to reserve large amounts of device memory that cannot be used for user allocations.

cudaErrorLaunchPendingCountExceeded This error indicates that a device runtime grid launch failed because the launch would exceed the limit cudaLimitDevRuntimePendingLaunchCount. For this launch to proceed successfully, cudaDeviceSetLimit must be called to set the cudaLimitDevRuntimePendingLaunchCount to be higher than the upper bound of outstanding launches that can be issued to the device runtime. Keep in mind that raising the limit of pending device runtime launches will require the runtime to reserve device memory that cannot be used for user allocations.

cudaErrorNotPermitted This error indicates the attempted operation is not permitted.

cudaErrorNotSupported This error indicates the attempted operation is not supported on the current system or device.

cudaErrorStartupFailure This indicates an internal startup failure in the CUDA runtime.

cudaErrorApiFailureBase Any unhandled CUDA driver error is added to this value and returned via the runtime. Production releases of CUDA should not return such errors.

Deprecated

This error return is deprecated as of CUDA 4.1.

5.28.3.5 enum cudaFuncCache

CUDA function cache configurations

Enumerator:

cudaFuncCachePreferNone Default function cache configuration, no preference

cudaFuncCachePreferShared Prefer larger shared memory and smaller L1 cache

cudaFuncCachePreferL1 Prefer larger L1 cache and smaller shared memory

cudaFuncCachePreferEqual Prefer equal size L1 cache and shared memory

5.28.3.6 enum cudaGraphicsCubeFace

CUDA graphics interop array indices for cube maps

Enumerator:

cudaGraphicsCubeFacePositiveX Positive X face of cubemap

cudaGraphicsCubeFaceNegativeX Negative X face of cubemap

cudaGraphicsCubeFacePositiveY Positive Y face of cubemap

cudaGraphicsCubeFaceNegativeY Negative Y face of cubemap
cudaGraphicsCubeFacePositiveZ Positive Z face of cubemap
cudaGraphicsCubeFaceNegativeZ Negative Z face of cubemap

5.28.3.7 enum cudaGraphicsMapFlags

CUDA graphics interop map flags

Enumerator:

cudaGraphicsMapFlagsNone Default; Assume resource can be read/written
cudaGraphicsMapFlagsReadOnly CUDA will not write to this resource
cudaGraphicsMapFlagsWriteDiscard CUDA will only write to and will not read from this resource

5.28.3.8 enum cudaGraphicsRegisterFlags

CUDA graphics interop register flags

Enumerator:

cudaGraphicsRegisterFlagsNone Default
cudaGraphicsRegisterFlagsReadOnly CUDA will not write to this resource
cudaGraphicsRegisterFlagsWriteDiscard CUDA will only write to and will not read from this resource
cudaGraphicsRegisterFlagsSurfaceLoadStore CUDA will bind this resource to a surface reference
cudaGraphicsRegisterFlagsTextureGather CUDA will perform texture gather operations on this resource

5.28.3.9 enum cudaLimit

CUDA Limits

Enumerator:

cudaLimitStackSize GPU thread stack size
cudaLimitPrintfFifoSize GPU printf/fprintf FIFO size
cudaLimitMallocHeapSize GPU malloc heap size
cudaLimitDevRuntimeSyncDepth GPU device runtime synchronize depth
cudaLimitDevRuntimePendingLaunchCount GPU device runtime pending launch count

5.28.3.10 enum cudaMemcpyKind

CUDA memory copy types

Enumerator:

cudaMemcpyHostToHost Host -> Host
cudaMemcpyHostToDevice Host -> Device
cudaMemcpyDeviceToHost Device -> Host
cudaMemcpyDeviceToDevice Device -> Device
cudaMemcpyDefault Default based unified virtual address space

5.28.3.11 enum cudaMemoryType

CUDA memory types

Enumerator:

cudaMemoryTypeHost Host memory

cudaMemoryTypeDevice Device memory

5.28.3.12 enum cudaOutputMode

CUDA Profiler Output modes

Enumerator:

cudaKeyValuePair Output mode Key-Value pair format.

cudaCSV Output mode Comma separated values format.

5.28.3.13 enum cudaResourceType

CUDA resource types

Enumerator:

cudaResourceTypeArray Array resource

cudaResourceTypeMipmappedArray Mipmapped array resource

cudaResourceTypeLinear Linear resource

cudaResourceTypePitch2D Pitch 2D resource

5.28.3.14 enum cudaResourceViewFormat

CUDA texture resource view formats

Enumerator:

cudaResViewFormatNone No resource view format (use underlying resource format)

cudaResViewFormatUnsignedChar1 1 channel unsigned 8-bit integers

cudaResViewFormatUnsignedChar2 2 channel unsigned 8-bit integers

cudaResViewFormatUnsignedChar4 4 channel unsigned 8-bit integers

cudaResViewFormatSignedChar1 1 channel signed 8-bit integers

cudaResViewFormatSignedChar2 2 channel signed 8-bit integers

cudaResViewFormatSignedChar4 4 channel signed 8-bit integers

cudaResViewFormatUnsignedShort1 1 channel unsigned 16-bit integers

cudaResViewFormatUnsignedShort2 2 channel unsigned 16-bit integers

cudaResViewFormatUnsignedShort4 4 channel unsigned 16-bit integers

cudaResViewFormatSignedShort1 1 channel signed 16-bit integers

cudaResViewFormatSignedShort2 2 channel signed 16-bit integers

cudaResViewFormatSignedShort4 4 channel signed 16-bit integers
cudaResViewFormatUnsignedInt1 1 channel unsigned 32-bit integers
cudaResViewFormatUnsignedInt2 2 channel unsigned 32-bit integers
cudaResViewFormatUnsignedInt4 4 channel unsigned 32-bit integers
cudaResViewFormatSignedInt1 1 channel signed 32-bit integers
cudaResViewFormatSignedInt2 2 channel signed 32-bit integers
cudaResViewFormatSignedInt4 4 channel signed 32-bit integers
cudaResViewFormatHalf1 1 channel 16-bit floating point
cudaResViewFormatHalf2 2 channel 16-bit floating point
cudaResViewFormatHalf4 4 channel 16-bit floating point
cudaResViewFormatFloat1 1 channel 32-bit floating point
cudaResViewFormatFloat2 2 channel 32-bit floating point
cudaResViewFormatFloat4 4 channel 32-bit floating point
cudaResViewFormatUnsignedBlockCompressed1 Block compressed 1
cudaResViewFormatUnsignedBlockCompressed2 Block compressed 2
cudaResViewFormatUnsignedBlockCompressed3 Block compressed 3
cudaResViewFormatUnsignedBlockCompressed4 Block compressed 4 unsigned
cudaResViewFormatSignedBlockCompressed4 Block compressed 4 signed
cudaResViewFormatUnsignedBlockCompressed5 Block compressed 5 unsigned
cudaResViewFormatSignedBlockCompressed5 Block compressed 5 signed
cudaResViewFormatUnsignedBlockCompressed6H Block compressed 6 unsigned half-float
cudaResViewFormatSignedBlockCompressed6H Block compressed 6 signed half-float
cudaResViewFormatUnsignedBlockCompressed7 Block compressed 7

5.28.3.15 enum cudaSharedMemConfig

CUDA shared memory configuration

5.28.3.16 enum cudaSurfaceBoundaryMode

CUDA Surface boundary modes

Enumerator:

cudaBoundaryModeZero Zero boundary mode
cudaBoundaryModeClamp Clamp boundary mode
cudaBoundaryModeTrap Trap boundary mode

5.28.3.17 enum cudaSurfaceFormatMode

CUDA Surface format modes

Enumerator:

cudaFormatModeForced Forced format mode
cudaFormatModeAuto Auto format mode

5.28.3.18 enum cudaTextureAddressMode

CUDA texture address modes

Enumerator:

cudaAddressModeWrap Wrapping address mode

cudaAddressModeClamp Clamp to edge address mode

cudaAddressModeMirror Mirror address mode

cudaAddressModeBorder Border address mode

5.28.3.19 enum cudaTextureFilterMode

CUDA texture filter modes

Enumerator:

cudaFilterModePoint Point filter mode

cudaFilterModeLinear Linear filter mode

5.28.3.20 enum cudaTextureReadMode

CUDA texture read modes

Enumerator:

cudaReadModeElementType Read texture as specified element type

cudaReadModeNormalizedFloat Read texture as normalized float

5.29 Data types used by CUDA driver

Data Structures

- struct CUDA_ARRAY3D_DESCRIPTOR_st
- struct CUDA_ARRAY_DESCRIPTOR_st
- struct CUDA_MEMCPY2D_st
- struct CUDA_MEMCPY3D_PEER_st
- struct CUDA_MEMCPY3D_st
- struct CUDA_POINTER_ATTRIBUTE_P2P_TOKENS_st
- struct CUDA_RESOURCE_DESC_st
- struct CUDA_RESOURCE_VIEW_DESC_st
- struct CUDA_TEXTURE_DESC_st
- struct CUdevprop_st
- struct CUipcEventHandle_st
- struct CUipcMemHandle_st

Defines

- #define CU_IPC_HANDLE_SIZE 64
- #define CU_LAUNCH_PARAM_BUFFER_POINTER ((void∗)0x01)
- #define CU_LAUNCH_PARAM_BUFFER_SIZE ((void∗)0x02)
- #define CU_LAUNCH_PARAM_END ((void∗)0x00)
- #define CU_MEMHOSTALLOC_DEVICEMAP 0x02
- #define CU_MEMHOSTALLOC_PORTABLE 0x01
- #define CU_MEMHOSTALLOC_WRITECOMBINED 0x04
- #define CU_MEMHOSTREGISTER_DEVICEMAP 0x02
- #define CU_MEMHOSTREGISTER_PORTABLE 0x01
- #define CU_PARAM_TR_DEFAULT -1
- #define CU_TRSA_OVERRIDE_FORMAT 0x01
- #define CU_TRSF_NORMALIZED_COORDINATES 0x02
- #define CU_TRSF_READ_AS_INTEGER 0x01
- #define CU_TRSF_SRGB 0x10
- #define CUDA_ARRAY3D_2DARRAY 0x01
- #define CUDA_ARRAY3D_CUBEMAP 0x04
- #define CUDA_ARRAY3D_LAYERED 0x01
- #define CUDA_ARRAY3D_SURFACE_LDST 0x02
- #define CUDA_ARRAY3D_TEXTURE_GATHER 0x08
- #define CUDA_VERSION 5000

Typedefs

- typedef enum CUaddress_mode_enum CUaddress_mode
- typedef struct CUarray_st ∗ CUarray
- typedef enum CUarray_cubemap_face_enum CUarray_cubemap_face
- typedef enum CUarray_format_enum CUarray_format
- typedef enum CUcomputemode_enum CUcomputemode
- typedef struct CUctx_st ∗ CUcontext
- typedef enum CUctx_flags_enum CUctx_flags

- typedef struct CUDA_ARRAY3D_DESCRIPTOR_st CUDA_ARRAY3D_DESCRIPTOR
- typedef struct CUDA_ARRAY_DESCRIPTOR_st CUDA_ARRAY_DESCRIPTOR
- typedef struct CUDA_MEMCPY2D_st CUDA_MEMCPY2D
- typedef struct CUDA_MEMCPY3D_st CUDA_MEMCPY3D
- typedef struct CUDA_MEMCPY3D_PEER_st CUDA_MEMCPY3D_PEER
- typedef struct CUDA_POINTER_ATTRIBUTE_P2P_TOKENS_st CUDA_POINTER_ATTRIBUTE_P2P_TOKENS
- typedef struct CUDA_RESOURCE_DESC_st CUDA_RESOURCE_DESC
- typedef struct CUDA_RESOURCE_VIEW_DESC_st CUDA_RESOURCE_VIEW_DESC
- typedef struct CUDA_TEXTURE_DESC_st CUDA_TEXTURE_DESC
- typedef int CUdevice
- typedef enum CUdevice_attribute_enum CUdevice_attribute
- typedef unsigned int CUdeviceptr
- typedef struct CUdevprop_st CUdevprop
- typedef struct CUevent_st * CUevent
- typedef enum CUevent_flags_enum CUevent_flags
- typedef enum CUfilter_mode_enum CUfilter_mode
- typedef enum CUfunc_cache_enum CUfunc_cache
- typedef struct CUfunc_st * CUfunction
- typedef enum CUfunction_attribute_enum CUfunction_attribute
- typedef enum CUgraphicsMapResourceFlags_enum CUgraphicsMapResourceFlags
- typedef enum CUgraphicsRegisterFlags_enum CUgraphicsRegisterFlags
- typedef struct CUgraphicsResource_st * CUgraphicsResource
- typedef struct CUipcEventHandle_st CUipcEventHandle
- typedef enum CUipcMem_flags_enum CUipcMem_flags
- typedef struct CUipcMemHandle_st CUipcMemHandle
- typedef enum CUjit_fallback_enum CUjit_fallback
- typedef enum CUjit_option_enum CUjit_option
- typedef enum CUjit_target_enum CUjit_target
- typedef enum CUlimit_enum CUlimit
- typedef enum CUmemorytype_enum CUmemorytype
- typedef struct CUmipmappedArray_st * CUmipmappedArray
- typedef struct CUmod_st * CUmodule
- typedef enum CUpointer_attribute_enum CUpointer_attribute
- typedef enum CUresourcetype_enum CUresourcetype
- typedef enum CUresourceViewFormat_enum CUresourceViewFormat
- typedef enum cudaError_enum CUresult
- typedef enum CUsharedconfig_enum CUsharedconfig
- typedef struct CUstream_st * CUstream
- typedef enum CUstream_flags_enum CUstream_flags
- typedef void(CUDA_CB * CUstreamCallback)(CUstream hStream, CUresult status, void *userData)
- typedef unsigned long long CUsurfObject
- typedef struct CUsurfref_st * CUsurfref
- typedef unsigned long long CUtexObject
- typedef struct CUtexref_st * CUtexref

Enumerations

- enum CUaddress_mode_enum {
  CU_TR_ADDRESS_MODE_WRAP = 0,
  CU_TR_ADDRESS_MODE_CLAMP = 1,
  CU_TR_ADDRESS_MODE_MIRROR = 2,
  CU_TR_ADDRESS_MODE_BORDER = 3 }
- enum CUarray_cubemap_face_enum {
  CU_CUBEMAP_FACE_POSITIVE_X = 0x00,
  CU_CUBEMAP_FACE_NEGATIVE_X = 0x01,
  CU_CUBEMAP_FACE_POSITIVE_Y = 0x02,
  CU_CUBEMAP_FACE_NEGATIVE_Y = 0x03,
  CU_CUBEMAP_FACE_POSITIVE_Z = 0x04,
  CU_CUBEMAP_FACE_NEGATIVE_Z = 0x05 }
- enum CUarray_format_enum {
  CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
  CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
  CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
  CU_AD_FORMAT_SIGNED_INT8 = 0x08,
  CU_AD_FORMAT_SIGNED_INT16 = 0x09,
  CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
  CU_AD_FORMAT_HALF = 0x10,
  CU_AD_FORMAT_FLOAT = 0x20 }
- enum CUcomputemode_enum {
  CU_COMPUTEMODE_DEFAULT = 0,
  CU_COMPUTEMODE_EXCLUSIVE = 1,
  CU_COMPUTEMODE_PROHIBITED = 2,
  CU_COMPUTEMODE_EXCLUSIVE_PROCESS = 3 }
- enum CUctx_flags_enum {
  CU_CTX_SCHED_AUTO = 0x00,
  CU_CTX_SCHED_SPIN = 0x01,
  CU_CTX_SCHED_YIELD = 0x02,
  CU_CTX_SCHED_BLOCKING_SYNC = 0x04,
  CU_CTX_BLOCKING_SYNC = 0x04 ,
  CU_CTX_MAP_HOST = 0x08,
  CU_CTX_LMEM_RESIZE_TO_MAX = 0x10 }
- enum cudaError_enum {
  CUDA_SUCCESS = 0,
  CUDA_ERROR_INVALID_VALUE = 1,
  CUDA_ERROR_OUT_OF_MEMORY = 2,
  CUDA_ERROR_NOT_INITIALIZED = 3,
  CUDA_ERROR_DEINITIALIZED = 4,

CUDA_ERROR_PROFILER_DISABLED = 5,
CUDA_ERROR_PROFILER_NOT_INITIALIZED = 6,
CUDA_ERROR_PROFILER_ALREADY_STARTED = 7,
CUDA_ERROR_PROFILER_ALREADY_STOPPED = 8,
CUDA_ERROR_NO_DEVICE = 100,
CUDA_ERROR_INVALID_DEVICE = 101,
CUDA_ERROR_INVALID_IMAGE = 200,
CUDA_ERROR_INVALID_CONTEXT = 201,
CUDA_ERROR_CONTEXT_ALREADY_CURRENT = 202,
CUDA_ERROR_MAP_FAILED = 205,
CUDA_ERROR_UNMAP_FAILED = 206,
CUDA_ERROR_ARRAY_IS_MAPPED = 207,
CUDA_ERROR_ALREADY_MAPPED = 208,
CUDA_ERROR_NO_BINARY_FOR_GPU = 209,
CUDA_ERROR_ALREADY_ACQUIRED = 210,
CUDA_ERROR_NOT_MAPPED = 211,
CUDA_ERROR_NOT_MAPPED_AS_ARRAY = 212,
CUDA_ERROR_NOT_MAPPED_AS_POINTER = 213,
CUDA_ERROR_ECC_UNCORRECTABLE = 214,
CUDA_ERROR_UNSUPPORTED_LIMIT = 215,
CUDA_ERROR_CONTEXT_ALREADY_IN_USE = 216,
CUDA_ERROR_PEER_ACCESS_UNSUPPORTED = 217,
CUDA_ERROR_INVALID_SOURCE = 300,
CUDA_ERROR_FILE_NOT_FOUND = 301,
CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND = 302,
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED = 303,
CUDA_ERROR_OPERATING_SYSTEM = 304,
CUDA_ERROR_INVALID_HANDLE = 400,
CUDA_ERROR_NOT_FOUND = 500,
CUDA_ERROR_NOT_READY = 600,
CUDA_ERROR_LAUNCH_FAILED = 700,
CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES = 701,
CUDA_ERROR_LAUNCH_TIMEOUT = 702,
CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING = 703,
CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED = 704,
CUDA_ERROR_PEER_ACCESS_NOT_ENABLED = 705,
CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE = 708,
CUDA_ERROR_CONTEXT_IS_DESTROYED = 709,
CUDA_ERROR_ASSERT = 710,
CUDA_ERROR_TOO_MANY_PEERS = 711,
CUDA_ERROR_HOST_MEMORY_ALREADY_REGISTERED = 712,

CUDA_ERROR_HOST_MEMORY_NOT_REGISTERED = 713,
CUDA_ERROR_NOT_PERMITTED = 800,
CUDA_ERROR_NOT_SUPPORTED = 801,
CUDA_ERROR_UNKNOWN = 999 }

- enum CUdevice_attribute_enum {
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK = 1,
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X = 2,
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y = 3,
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z = 4,
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X = 5,
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y = 6,
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z = 7,
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK = 8,
CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK = 8,
CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY = 9,
CU_DEVICE_ATTRIBUTE_WARP_SIZE = 10,
CU_DEVICE_ATTRIBUTE_MAX_PITCH = 11,
CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK = 12,
CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK = 12,
CU_DEVICE_ATTRIBUTE_CLOCK_RATE = 13,
CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT = 14,
CU_DEVICE_ATTRIBUTE_GPU_OVERLAP = 15,
CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT = 16,
CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT = 17,
CU_DEVICE_ATTRIBUTE_INTEGRATED = 18,
CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY = 19,
CU_DEVICE_ATTRIBUTE_COMPUTE_MODE = 20,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH = 21,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH = 22,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT = 23,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH = 24,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT = 25,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH = 26,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH = 27,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT = 28,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS = 29,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH = 27,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT = 28,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES = 29,
CU_DEVICE_ATTRIBUTE_SURFACE_ALIGNMENT = 30,
CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS = 31,

CU_DEVICE_ATTRIBUTE_ECC_ENABLED = 32,
CU_DEVICE_ATTRIBUTE_PCI_BUS_ID = 33,
CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID = 34,
CU_DEVICE_ATTRIBUTE_TCC_DRIVER = 35,
CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE = 36,
CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH = 37,
CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE = 38,
CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR = 39,
CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT = 40,
CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING = 41,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH = 42,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS = 43,
CU_DEVICE_ATTRIBUTE_CAN_TEX2D_GATHER = 44,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_WIDTH = 45,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_HEIGHT = 46,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH_ALTERNATE = 47,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE = 48,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE = 49,
CU_DEVICE_ATTRIBUTE_PCI_DOMAIN_ID = 50,
CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT = 51,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_WIDTH = 52,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH = 53,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_LAYERS = 54,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_WIDTH = 55,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_WIDTH = 56,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_HEIGHT = 57,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_WIDTH = 58,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_HEIGHT = 59,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_DEPTH = 60,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_WIDTH = 61,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_LAYERS = 62,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_WIDTH = 63,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_HEIGHT = 64,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_LAYERS = 65,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_WIDTH = 66,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH = 67,
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS = 68,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH = 69,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH = 70,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT = 71,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH = 72,

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH = 73,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT = 74,
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR = 75,
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR = 76,
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH = 77 }

- enum CUevent_flags_enum {
  CU_EVENT_DEFAULT = 0x0,
  CU_EVENT_BLOCKING_SYNC = 0x1,
  CU_EVENT_DISABLE_TIMING = 0x2,
  CU_EVENT_INTERPROCESS = 0x4 }
- enum CUfilter_mode_enum {
  CU_TR_FILTER_MODE_POINT = 0,
  CU_TR_FILTER_MODE_LINEAR = 1 }
- enum CUfunc_cache_enum {
  CU_FUNC_CACHE_PREFER_NONE = 0x00,
  CU_FUNC_CACHE_PREFER_SHARED = 0x01,
  CU_FUNC_CACHE_PREFER_L1 = 0x02,
  CU_FUNC_CACHE_PREFER_EQUAL = 0x03 }
- enum CUfunction_attribute_enum {
  CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK = 0,
  CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES = 1,
  CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES = 2,
  CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES = 3,
  CU_FUNC_ATTRIBUTE_NUM_REGS = 4,
  CU_FUNC_ATTRIBUTE_PTX_VERSION = 5,
  CU_FUNC_ATTRIBUTE_BINARY_VERSION = 6 }
- enum CUgraphicsMapResourceFlags_enum
- enum CUgraphicsRegisterFlags_enum
- enum CUipcMem_flags_enum { CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS = 0x1 }
- enum CUjit_fallback_enum {
  CU_PREFER_PTX = 0,
  CU_PREFER_BINARY }
- enum CUjit_option_enum {
  CU_JIT_MAX_REGISTERS = 0,
  CU_JIT_THREADS_PER_BLOCK,
  CU_JIT_WALL_TIME,
  CU_JIT_INFO_LOG_BUFFER,
  CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES,
  CU_JIT_ERROR_LOG_BUFFER,
  CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES,
  CU_JIT_OPTIMIZATION_LEVEL,
  CU_JIT_TARGET_FROM_CUCONTEXT,
  CU_JIT_TARGET,
  CU_JIT_FALLBACK_STRATEGY }

- enum CUjit_target_enum {
  CU_TARGET_COMPUTE_10 = 0,
  CU_TARGET_COMPUTE_11,
  CU_TARGET_COMPUTE_12,
  CU_TARGET_COMPUTE_13,
  CU_TARGET_COMPUTE_20,
  CU_TARGET_COMPUTE_21,
  CU_TARGET_COMPUTE_30,
  CU_TARGET_COMPUTE_35 }
- enum CUlimit_enum {
  CU_LIMIT_STACK_SIZE = 0x00,
  CU_LIMIT_PRINTF_FIFO_SIZE = 0x01,
  CU_LIMIT_MALLOC_HEAP_SIZE = 0x02,
  CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH = 0x03,
  CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT = 0x04 }
- enum CUmemorytype_enum {
  CU_MEMORYTYPE_HOST = 0x01,
  CU_MEMORYTYPE_DEVICE = 0x02,
  CU_MEMORYTYPE_ARRAY = 0x03,
  CU_MEMORYTYPE_UNIFIED = 0x04 }
- enum CUpointer_attribute_enum {
  CU_POINTER_ATTRIBUTE_CONTEXT = 1,
  CU_POINTER_ATTRIBUTE_MEMORY_TYPE = 2,
  CU_POINTER_ATTRIBUTE_DEVICE_POINTER = 3,
  CU_POINTER_ATTRIBUTE_HOST_POINTER = 4,
  CU_POINTER_ATTRIBUTE_P2P_TOKENS = 5 }
- enum CUresourcetype_enum {
  CU_RESOURCE_TYPE_ARRAY = 0x00,
  CU_RESOURCE_TYPE_MIPMAPPED_ARRAY = 0x01,
  CU_RESOURCE_TYPE_LINEAR = 0x02,
  CU_RESOURCE_TYPE_PITCH2D = 0x03 }
- enum CUresourceViewFormat_enum {
  CU_RES_VIEW_FORMAT_NONE = 0x00,
  CU_RES_VIEW_FORMAT_UINT_1X8 = 0x01,
  CU_RES_VIEW_FORMAT_UINT_2X8 = 0x02,
  CU_RES_VIEW_FORMAT_UINT_4X8 = 0x03,
  CU_RES_VIEW_FORMAT_SINT_1X8 = 0x04,
  CU_RES_VIEW_FORMAT_SINT_2X8 = 0x05,
  CU_RES_VIEW_FORMAT_SINT_4X8 = 0x06,
  CU_RES_VIEW_FORMAT_UINT_1X16 = 0x07,
  CU_RES_VIEW_FORMAT_UINT_2X16 = 0x08,
  CU_RES_VIEW_FORMAT_UINT_4X16 = 0x09,

CU_RES_VIEW_FORMAT_SINT_1X16 = 0x0a,
CU_RES_VIEW_FORMAT_SINT_2X16 = 0x0b,
CU_RES_VIEW_FORMAT_SINT_4X16 = 0x0c,
CU_RES_VIEW_FORMAT_UINT_1X32 = 0x0d,
CU_RES_VIEW_FORMAT_UINT_2X32 = 0x0e,
CU_RES_VIEW_FORMAT_UINT_4X32 = 0x0f,
CU_RES_VIEW_FORMAT_SINT_1X32 = 0x10,
CU_RES_VIEW_FORMAT_SINT_2X32 = 0x11,
CU_RES_VIEW_FORMAT_SINT_4X32 = 0x12,
CU_RES_VIEW_FORMAT_FLOAT_1X16 = 0x13,
CU_RES_VIEW_FORMAT_FLOAT_2X16 = 0x14,
CU_RES_VIEW_FORMAT_FLOAT_4X16 = 0x15,
CU_RES_VIEW_FORMAT_FLOAT_1X32 = 0x16,
CU_RES_VIEW_FORMAT_FLOAT_2X32 = 0x17,
CU_RES_VIEW_FORMAT_FLOAT_4X32 = 0x18,
CU_RES_VIEW_FORMAT_UNSIGNED_BC1 = 0x19,
CU_RES_VIEW_FORMAT_UNSIGNED_BC2 = 0x1a,
CU_RES_VIEW_FORMAT_UNSIGNED_BC3 = 0x1b,
CU_RES_VIEW_FORMAT_UNSIGNED_BC4 = 0x1c,
CU_RES_VIEW_FORMAT_SIGNED_BC4 = 0x1d,
CU_RES_VIEW_FORMAT_UNSIGNED_BC5 = 0x1e,
CU_RES_VIEW_FORMAT_SIGNED_BC5 = 0x1f,
CU_RES_VIEW_FORMAT_UNSIGNED_BC6H = 0x20,
CU_RES_VIEW_FORMAT_SIGNED_BC6H = 0x21,
CU_RES_VIEW_FORMAT_UNSIGNED_BC7 = 0x22 }

- enum CUsharedconfig_enum {
  CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE = 0x00,
  CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE = 0x01,
  CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE = 0x02 }
- enum CUstream_flags_enum {
  CU_STREAM_DEFAULT = 0x0,
  CU_STREAM_NON_BLOCKING = 0x1 }

5.29.1 Define Documentation

5.29.1.1 #define CU_IPC_HANDLE_SIZE 64

CUDA IPC handle size

5.29.1.2 #define CU_LAUNCH_PARAM_BUFFER_POINTER ((void∗)0x01)

Indicator that the next value in the `extra` parameter to cuLaunchKernel will be a pointer to a buffer containing all kernel parameters used for launching kernel `f`. This buffer needs to honor all alignment/padding requirements of the individual parameters. If CU_LAUNCH_PARAM_BUFFER_SIZE is not also specified in the `extra` array, then CU_LAUNCH_PARAM_BUFFER_POINTER will have no effect.

5.29.1.3 #define CU_LAUNCH_PARAM_BUFFER_SIZE ((void*)0x02)

Indicator that the next value in the `extra` parameter to cuLaunchKernel will be a pointer to a size_t which contains the size of the buffer specified with CU_LAUNCH_PARAM_BUFFER_POINTER. It is required that CU_LAUNCH_PARAM_BUFFER_POINTER also be specified in the `extra` array if the value associated with CU_LAUNCH_PARAM_BUFFER_SIZE is not zero.

5.29.1.4 #define CU_LAUNCH_PARAM_END ((void*)0x00)

End of array terminator for the `extra` parameter to cuLaunchKernel

5.29.1.5 #define CU_MEMHOSTALLOC_DEVICEMAP 0x02

If set, host memory is mapped into CUDA address space and cuMemHostGetDevicePointer() may be called on the host pointer. Flag for cuMemHostAlloc()

5.29.1.6 #define CU_MEMHOSTALLOC_PORTABLE 0x01

If set, host memory is portable between CUDA contexts. Flag for cuMemHostAlloc()

5.29.1.7 #define CU_MEMHOSTALLOC_WRITECOMBINED 0x04

If set, host memory is allocated as write-combined - fast to write, faster to DMA, slow to read except via SSE4 streaming load instruction (MOVNTDQA). Flag for cuMemHostAlloc()

5.29.1.8 #define CU_MEMHOSTREGISTER_DEVICEMAP 0x02

If set, host memory is mapped into CUDA address space and cuMemHostGetDevicePointer() may be called on the host pointer. Flag for cuMemHostRegister()

5.29.1.9 #define CU_MEMHOSTREGISTER_PORTABLE 0x01

If set, host memory is portable between CUDA contexts. Flag for cuMemHostRegister()

5.29.1.10 #define CU_PARAM_TR_DEFAULT -1

For texture references loaded into the module, use default texunit from texture reference.

5.29.1.11 #define CU_TRSA_OVERRIDE_FORMAT 0x01

Override the texref format with a format inferred from the array. Flag for cuTexRefSetArray()

5.29.1.12 #define CU_TRSF_NORMALIZED_COORDINATES 0x02

Use normalized texture coordinates in the range [0,1) instead of [0,dim). Flag for cuTexRefSetFlags()

5.29.1.13 #define CU_TRSF_READ_AS_INTEGER 0x01

Read the texture as integers rather than promoting the values to floats in the range [0,1]. Flag for cuTexRefSetFlags()

5.29.1.14 #define CU_TRSF_SRGB 0x10

Perform sRGB->linear conversion during texture read. Flag for cuTexRefSetFlags()

5.29.1.15 #define CUDA_ARRAY3D_2DARRAY 0x01

Deprecated, use CUDA_ARRAY3D_LAYERED

5.29.1.16 #define CUDA_ARRAY3D_CUBEMAP 0x04

If set, the CUDA array is a collection of six 2D arrays, representing faces of a cube. The width of such a CUDA array must be equal to its height, and Depth must be six. If CUDA_ARRAY3D_LAYERED flag is also set, then the CUDA array is a collection of cubemaps and Depth must be a multiple of six.

5.29.1.17 #define CUDA_ARRAY3D_LAYERED 0x01

If set, the CUDA array is a collection of layers, where each layer is either a 1D or a 2D array and the Depth member of CUDA_ARRAY3D_DESCRIPTOR specifies the number of layers, not the depth of a 3D array.

5.29.1.18 #define CUDA_ARRAY3D_SURFACE_LDST 0x02

This flag must be set in order to bind a surface reference to the CUDA array

5.29.1.19 #define CUDA_ARRAY3D_TEXTURE_GATHER 0x08

This flag must be set in order to perform texture gather operations on a CUDA array.

5.29.1.20 #define CUDA_VERSION 5000

CUDA API version number

5.29.2 Typedef Documentation

5.29.2.1 typedef enum CUaddress_mode_enum CUaddress_mode

Texture reference addressing modes

5.29.2.2 typedef struct CUarray_st* CUarray

CUDA array

5.29.2.3 typedef enum CUarray_cubemap_face_enum CUarray_cubemap_face

Array indices for cube faces

5.29.2.4 typedef enum CUarray_format_enum CUarray_format

Array formats

5.29.2.5 typedef enum CUcomputemode_enum CUcomputemode

Compute Modes

5.29.2.6 typedef struct CUctx_st* CUcontext

CUDA context

5.29.2.7 typedef enum CUctx_flags_enum CUctx_flags

Context creation flags

5.29.2.8 typedef struct CUDA_ARRAY3D_DESCRIPTOR_st CUDA_ARRAY3D_DESCRIPTOR

3D array descriptor

5.29.2.9 typedef struct CUDA_ARRAY_DESCRIPTOR_st CUDA_ARRAY_DESCRIPTOR

Array descriptor

5.29.2.10 typedef struct CUDA_MEMCPY2D_st CUDA_MEMCPY2D

2D memory copy parameters

5.29.2.11 typedef struct CUDA_MEMCPY3D_st CUDA_MEMCPY3D

3D memory copy parameters

5.29.2.12 typedef struct CUDA_MEMCPY3D_PEER_st CUDA_MEMCPY3D_PEER

3D memory cross-context copy parameters

5.29.2.13 typedef struct CUDA_POINTER_ATTRIBUTE_P2P_TOKENS_st CUDA_POINTER_ATTRIBUTE_P2P_TOKENS

GPU Direct v3 tokens

5.29.2.14 typedef struct CUDA_RESOURCE_DESC_st CUDA_RESOURCE_DESC

CUDA Resource descriptor

5.29.2.15 typedef struct CUDA_RESOURCE_VIEW_DESC_st CUDA_RESOURCE_VIEW_DESC

Resource view descriptor

5.29.2.16 typedef struct CUDA_TEXTURE_DESC_st CUDA_TEXTURE_DESC

Texture descriptor

5.29.2.17 typedef int CUdevice

CUDA device

5.29.2.18 typedef enum CUdevice_attribute_enum CUdevice_attribute

Device properties

5.29.2.19 typedef unsigned int CUdeviceptr

CUDA device pointer

5.29.2.20 typedef struct CUdevprop_st CUdevprop

Legacy device properties

5.29.2.21 typedef struct CUevent_st* CUevent

CUDA event

5.29.2.22 typedef enum CUevent_flags_enum CUevent_flags

Event creation flags

5.29.2.23 typedef enum CUfilter_mode_enum CUfilter_mode

Texture reference filtering modes

5.29.2.24 typedef enum CUfunc_cache_enum CUfunc_cache

Function cache configurations

5.29.2.25 typedef struct CUfunc_st* CUfunction

CUDA function

5.29.2.26 typedef enum CUfunction_attribute_enum CUfunction_attribute

Function properties

5.29.2.27 typedef enum CUgraphicsMapResourceFlags_enum CUgraphicsMapResourceFlags

Flags for mapping and unmapping interop resources

5.29.2.28 typedef enum CUgraphicsRegisterFlags_enum CUgraphicsRegisterFlags

Flags to register a graphics resource

5.29.2.29 typedef struct CUgraphicsResource_st* CUgraphicsResource

CUDA graphics interop resource

5.29.2.30 typedef struct CUipcEventHandle_st CUipcEventHandle

CUDA IPC event handle

5.29.2.31 typedef enum CUipcMem_flags_enum CUipcMem_flags

CUDA Ipc Mem Flags

5.29.2.32 typedef struct CUipcMemHandle_st CUipcMemHandle

CUDA IPC mem handle

5.29.2.33 typedef enum CUjit_fallback_enum CUjit_fallback

Cubin matching fallback strategies

5.29.2.34 typedef enum CUjit_option_enum CUjit_option

Online compiler options

5.29.2.35 typedef enum CUjit_target_enum CUjit_target

Online compilation targets

5.29.2.36 typedef enum CUlimit_enum CUlimit

Limits

5.29.2.37 typedef enum CUmemorytype_enum CUmemorytype

Memory types

5.29.2.38 typedef struct CUmipmappedArray_st* CUmipmappedArray

CUDA mipmapped array

5.29.2.39 typedef struct CUmod_st* CUmodule

CUDA module

5.29.2.40 typedef enum CUpointer_attribute_enum CUpointer_attribute

Pointer information

5.29.2.41 typedef enum CUresourcetype_enum CUresourcetype

Resource types

5.29.2.42 typedef enum CUresourceViewFormat_enum CUresourceViewFormat

Resource view format

5.29.2.43 typedef enum cudaError_enum CUresult

Error codes

5.29.2.44 typedef enum CUsharedconfig_enum CUsharedconfig

Shared memory configurations

5.29.2.45 typedef struct CUstream_st* CUstream

CUDA stream

5.29.2.46 typedef enum CUstream_flags_enum CUstream_flags

Stream creation flags

5.29.2.47 typedef void(CUDA_CB * CUstreamCallback)(CUstream hStream, CUresult status, void *userData)

CUDA stream callback

Parameters:

hStream The stream the callback was added to, as passed to cuStreamAddCallback. May be NULL.

status CUDA_SUCCESS or any persistent error on the stream.

userData User parameter provided at registration.

5.29.2.48 typedef unsigned long long CUsurfObject

CUDA surface object

5.29.2.49 typedef struct CUsurfref_st* CUsurfref

CUDA surface reference

5.29.2.50 typedef unsigned long long CUtexObject

CUDA texture object

5.29.2.51 typedef struct CUtexref_st* CUtexref

CUDA texture reference

5.29.3 Enumeration Type Documentation

5.29.3.1 enum CUaddress_mode_enum

Texture reference addressing modes

Enumerator:

CU_TR_ADDRESS_MODE_WRAP Wrapping address mode

CU_TR_ADDRESS_MODE_CLAMP Clamp to edge address mode

CU_TR_ADDRESS_MODE_MIRROR Mirror address mode

CU_TR_ADDRESS_MODE_BORDER Border address mode

5.29.3.2 enum CUarray_cubemap_face_enum

Array indices for cube faces

Enumerator:

CU_CUBEMAP_FACE_POSITIVE_X Positive X face of cubemap

CU_CUBEMAP_FACE_NEGATIVE_X Negative X face of cubemap

CU_CUBEMAP_FACE_POSITIVE_Y Positive Y face of cubemap

CU_CUBEMAP_FACE_NEGATIVE_Y Negative Y face of cubemap

CU_CUBEMAP_FACE_POSITIVE_Z Positive Z face of cubemap

CU_CUBEMAP_FACE_NEGATIVE_Z Negative Z face of cubemap

5.29.3.3 enum CUarray_format_enum

Array formats

Enumerator:

CU_AD_FORMAT_UNSIGNED_INT8 Unsigned 8-bit integers

CU_AD_FORMAT_UNSIGNED_INT16 Unsigned 16-bit integers

CU_AD_FORMAT_UNSIGNED_INT32 Unsigned 32-bit integers

CU_AD_FORMAT_SIGNED_INT8 Signed 8-bit integers

CU_AD_FORMAT_SIGNED_INT16 Signed 16-bit integers

CU_AD_FORMAT_SIGNED_INT32 Signed 32-bit integers

CU_AD_FORMAT_HALF 16-bit floating point

CU_AD_FORMAT_FLOAT 32-bit floating point

5.29.3.4 enum CUcomputemode_enum

Compute Modes

Enumerator:

CU_COMPUTEMODE_DEFAULT Default compute mode (Multiple contexts allowed per device)

CU_COMPUTEMODE_EXCLUSIVE Compute-exclusive-thread mode (Only one context used by a single thread can be present on this device at a time)

CU_COMPUTEMODE_PROHIBITED Compute-prohibited mode (No contexts can be created on this device at this time)

CU_COMPUTEMODE_EXCLUSIVE_PROCESS Compute-exclusive-process mode (Only one context used by a single process can be present on this device at a time)

5.29.3.5 enum CUctx_flags_enum

Context creation flags

Enumerator:

CU_CTX_SCHED_AUTO Automatic scheduling

CU_CTX_SCHED_SPIN Set spin as default scheduling

CU_CTX_SCHED_YIELD Set yield as default scheduling

CU_CTX_SCHED_BLOCKING_SYNC Set blocking synchronization as default scheduling

CU_CTX_BLOCKING_SYNC Set blocking synchronization as default scheduling

Deprecated

This flag was deprecated as of CUDA 4.0 and was replaced with CU_CTX_SCHED_BLOCKING_SYNC.

CU_CTX_MAP_HOST Support mapped pinned allocations

CU_CTX_LMEM_RESIZE_TO_MAX Keep local memory allocation after launch

5.29.3.6 enum cudaError_enum

Error codes

Enumerator:

CUDA_SUCCESS The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see cuEventQuery() and cuStreamQuery()).

CUDA_ERROR_INVALID_VALUE This indicates that one or more of the parameters passed to the API call is not within an acceptable range of values.

CUDA_ERROR_OUT_OF_MEMORY The API call failed because it was unable to allocate enough memory to perform the requested operation.

CUDA_ERROR_NOT_INITIALIZED This indicates that the CUDA driver has not been initialized with cuInit() or that initialization has failed.

CUDA_ERROR_DEINITIALIZED This indicates that the CUDA driver is in the process of shutting down.

CUDA_ERROR_PROFILER_DISABLED This indicates profiler is not initialized for this run. This can happen when the application is running with external profiling tools like visual profiler.

Deprecated

CUDA_ERROR_PROFILER_NOT_INITIALIZED This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via cuProfilerStart or cuProfilerStop without initialization.

Deprecated

CUDA_ERROR_PROFILER_ALREADY_STARTED This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStart() when profiling is already enabled.

Deprecated

CUDA_ERROR_PROFILER_ALREADY_STOPPED This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStop() when profiling is already disabled.

CUDA_ERROR_NO_DEVICE This indicates that no CUDA-capable devices were detected by the installed CUDA driver.

CUDA_ERROR_INVALID_DEVICE This indicates that the device ordinal supplied by the user does not correspond to a valid CUDA device.

CUDA_ERROR_INVALID_IMAGE This indicates that the device kernel image is invalid. This can also indicate an invalid CUDA module.

CUDA_ERROR_INVALID_CONTEXT This most frequently indicates that there is no context bound to the current thread. This can also be returned if the context passed to an API call is not a valid handle (such as a context that has had cuCtxDestroy() invoked on it). This can also be returned if a user mixes different API versions (i.e. 3010 context with 3020 API calls). See cuCtxGetApiVersion() for more details.

CUDA_ERROR_CONTEXT_ALREADY_CURRENT This indicated that the context being supplied as a parameter to the API call was already the active context.

Deprecated

This error return is deprecated as of CUDA 3.2. It is no longer an error to attempt to push the active context via cuCtxPushCurrent().

CUDA_ERROR_MAP_FAILED This indicates that a map or register operation has failed.

CUDA_ERROR_UNMAP_FAILED This indicates that an unmap or unregister operation has failed.

CUDA_ERROR_ARRAY_IS_MAPPED This indicates that the specified array is currently mapped and thus cannot be destroyed.

CUDA_ERROR_ALREADY_MAPPED This indicates that the resource is already mapped.

CUDA_ERROR_NO_BINARY_FOR_GPU This indicates that there is no kernel image available that is suitable for the device. This can occur when a user specifies code generation options for a particular CUDA source file that do not include the corresponding device configuration.

CUDA_ERROR_ALREADY_ACQUIRED This indicates that a resource has already been acquired.

CUDA_ERROR_NOT_MAPPED This indicates that a resource is not mapped.

CUDA_ERROR_NOT_MAPPED_AS_ARRAY This indicates that a mapped resource is not available for access as an array.

CUDA_ERROR_NOT_MAPPED_AS_POINTER This indicates that a mapped resource is not available for access as a pointer.

CUDA_ERROR_ECC_UNCORRECTABLE This indicates that an uncorrectable ECC error was detected during execution.

CUDA_ERROR_UNSUPPORTED_LIMIT This indicates that the CUlimit passed to the API call is not supported by the active device.

CUDA_ERROR_CONTEXT_ALREADY_IN_USE This indicates that the CUcontext passed to the API call can only be bound to a single CPU thread at a time but is already bound to a CPU thread.

CUDA_ERROR_PEER_ACCESS_UNSUPPORTED This indicates that peer access is not supported across the given devices.

CUDA_ERROR_INVALID_SOURCE This indicates that the device kernel source is invalid.

CUDA_ERROR_FILE_NOT_FOUND This indicates that the file specified was not found.

CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND This indicates that a link to a shared object failed to resolve.

CUDA_ERROR_SHARED_OBJECT_INIT_FAILED This indicates that initialization of a shared object failed.

CUDA_ERROR_OPERATING_SYSTEM This indicates that an OS call failed.

CUDA_ERROR_INVALID_HANDLE This indicates that a resource handle passed to the API call was not valid. Resource handles are opaque types like CUstream and CUevent.

CUDA_ERROR_NOT_FOUND This indicates that a named symbol was not found. Examples of symbols are global/constant variable names, texture names, and surface names.

CUDA_ERROR_NOT_READY This indicates that asynchronous operations issued previously have not completed yet. This result is not actually an error, but must be indicated differently than CUDA_SUCCESS (which indicates completion). Calls that may return this value include cuEventQuery() and cuStreamQuery().

CUDA_ERROR_LAUNCH_FAILED An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. The context cannot be used, so it must be destroyed (and a new one should be created). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES This indicates that a launch did not occur because it did not have appropriate resources. This error usually indicates that the user has attempted to pass too many arguments to the device kernel, or the kernel launch specifies too many threads for the kernel's register count. Passing arguments of the wrong size (i.e. a 64-bit pointer when a 32-bit int is expected) is equivalent to passing too many arguments and can also result in this error.

CUDA_ERROR_LAUNCH_TIMEOUT This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device attribute CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT for more information. The context cannot be used (and must be destroyed similar to CUDA_ERROR_LAUNCH_FAILED). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING This error indicates a kernel launch that uses an incompatible texturing mode.

CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED This error indicates that a call to cuCtxEnablePeerAccess() is trying to re-enable peer access to a context which has already had peer access to it enabled.

CUDA_ERROR_PEER_ACCESS_NOT_ENABLED This error indicates that cuCtxDisablePeerAccess() is trying to disable peer access which has not been enabled yet via cuCtxEnablePeerAccess().

CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE This error indicates that the primary context for the specified device has already been initialized.

CUDA_ERROR_CONTEXT_IS_DESTROYED This error indicates that the context current to the calling thread has been destroyed using cuCtxDestroy, or is a primary context which has not yet been initialized.

CUDA_ERROR_ASSERT A device-side assert triggered during kernel execution. The context cannot be used anymore, and must be destroyed. All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_TOO_MANY_PEERS This error indicates that the hardware resources required to enable peer access have been exhausted for one or more of the devices passed to cuCtxEnablePeerAccess().

CUDA_ERROR_HOST_MEMORY_ALREADY_REGISTERED This error indicates that the memory range passed to cuMemHostRegister() has already been registered.

CUDA_ERROR_HOST_MEMORY_NOT_REGISTERED This error indicates that the pointer passed to cuMemHostUnregister() does not correspond to any currently registered memory region.

CUDA_ERROR_NOT_PERMITTED This error indicates that the attempted operation is not permitted.

CUDA_ERROR_NOT_SUPPORTED This error indicates that the attempted operation is not supported on the current system or device.

CUDA_ERROR_UNKNOWN This indicates that an unknown internal error has occurred.

5.29.3.7 enum CUdevice_attribute_enum

Device properties

Enumerator:

CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK Maximum number of threads per block

CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X Maximum block dimension X

CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y Maximum block dimension Y

CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z Maximum block dimension Z

CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X Maximum grid dimension X

CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y Maximum grid dimension Y

CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z Maximum grid dimension Z

CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK Maximum shared memory available per block in bytes

CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK Deprecated, use CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK

CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY Memory available on device for __constant__ variables in a CUDA C kernel in bytes

CU_DEVICE_ATTRIBUTE_WARP_SIZE Warp size in threads

CU_DEVICE_ATTRIBUTE_MAX_PITCH Maximum pitch in bytes allowed by memory copies

CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK Maximum number of 32-bit registers available per block

CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK Deprecated, use CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK

CU_DEVICE_ATTRIBUTE_CLOCK_RATE Peak clock frequency in kilohertz

CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT Alignment requirement for textures

CU_DEVICE_ATTRIBUTE_GPU_OVERLAP Device can possibly copy memory and execute a kernel concurrently. Deprecated. Use instead CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT.

CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT Number of multiprocessors on device

CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT Specifies whether there is a run time limit on kernels

CU_DEVICE_ATTRIBUTE_INTEGRATED Device is integrated with host memory

CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY Device can map host memory into CUDA address space

CU_DEVICE_ATTRIBUTE_COMPUTE_MODE Compute mode (See CUcomputemode for details)

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH Maximum 1D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH Maximum 2D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT Maximum 2D texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH Maximum 3D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT Maximum 3D texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH Maximum 3D texture depth

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH Maximum 2D layered texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT Maximum 2D layered texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS Maximum layers in a 2D layered texture

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH Deprecated, use CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT Deprecated, use CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES Deprecated, use CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS

CU_DEVICE_ATTRIBUTE_SURFACE_ALIGNMENT Alignment requirement for surfaces

CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS Device can possibly execute multiple kernels concurrently

CU_DEVICE_ATTRIBUTE_ECC_ENABLED Device has ECC support enabled

CU_DEVICE_ATTRIBUTE_PCI_BUS_ID PCI bus ID of the device

CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID PCI device ID of the device

CU_DEVICE_ATTRIBUTE_TCC_DRIVER Device is using TCC driver model

CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE Peak memory clock frequency in kilohertz

CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH Global memory bus width in bits

CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE Size of L2 cache in bytes

CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR Maximum resident threads per multiprocessor

CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT Number of asynchronous engines

CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING Device shares a unified address space with the host

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH Maximum 1D layered texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS Maximum layers in a 1D layered texture

CU_DEVICE_ATTRIBUTE_CAN_TEX2D_GATHER Deprecated, do not use.

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_WIDTH Maximum 2D texture width if CUDA_ARRAY3D_TEXTURE_GATHER is set

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_HEIGHT Maximum 2D texture height if CUDA_ARRAY3D_TEXTURE_GATHER is set

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH_ALTERNATE Alternate maximum 3D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE Alternate maximum 3D texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE Alternate maximum 3D texture depth

CU_DEVICE_ATTRIBUTE_PCI_DOMAIN_ID PCI domain ID of the device

CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT Pitch alignment requirement for textures

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_WIDTH Maximum cubemap texture width/height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH Maximum cubemap layered texture width/height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_LAYERS Maximum layers in a cubemap layered texture

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_WIDTH Maximum 1D surface width

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_WIDTH Maximum 2D surface width

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_HEIGHT Maximum 2D surface height

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_WIDTH Maximum 3D surface width

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_HEIGHT Maximum 3D surface height

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_DEPTH Maximum 3D surface depth

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_WIDTH Maximum 1D layered surface width

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_LAYERS Maximum layers in a 1D layered surface

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_WIDTH Maximum 2D layered surface width

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_HEIGHT Maximum 2D layered surface height

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_LAYERS Maximum layers in a 2D layered surface

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_WIDTH Maximum cubemap surface width

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH Maximum cubemap layered surface width

CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS Maximum layers in a cubemap layered surface

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH Maximum 1D linear texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH Maximum 2D linear texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT Maximum 2D linear texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH Maximum 2D linear texture pitch in bytes

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH Maximum mipmapped 2D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT Maximum mipmapped 2D texture height

CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR Major compute capability version number

CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR Minor compute capability version number

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH Maximum mipmapped 1D texture width

5.29.3.8 enum CUevent_flags_enum

Event creation flags

Enumerator:

CU_EVENT_DEFAULT Default event flag

CU_EVENT_BLOCKING_SYNC Event uses blocking synchronization

CU_EVENT_DISABLE_TIMING Event will not record timing data

CU_EVENT_INTERPROCESS Event is suitable for interprocess use. CU_EVENT_DISABLE_TIMING must be set

5.29.3.9 enum CUfilter_mode_enum

Texture reference filtering modes

Enumerator:

CU_TR_FILTER_MODE_POINT Point filter mode

CU_TR_FILTER_MODE_LINEAR Linear filter mode

5.29.3.10 enum CUfunc_cache_enum

Function cache configurations

Enumerator:

CU_FUNC_CACHE_PREFER_NONE no preference for shared memory or L1 (default)

CU_FUNC_CACHE_PREFER_SHARED prefer larger shared memory and smaller L1 cache

CU_FUNC_CACHE_PREFER_L1 prefer larger L1 cache and smaller shared memory

CU_FUNC_CACHE_PREFER_EQUAL prefer equal sized L1 cache and shared memory

5.29.3.11 enum CUfunction_attribute_enum

Function properties

Enumerator:

CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK The maximum number of threads per block, beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.

CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES The size in bytes of statically-allocated shared memory required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.

CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES The size in bytes of user-allocated constant memory required by this function.

CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES The size in bytes of local memory used by each thread of this function.

CU_FUNC_ATTRIBUTE_NUM_REGS The number of registers used by each thread of this function.

CU_FUNC_ATTRIBUTE_PTX_VERSION The PTX virtual architecture version for which the function was compiled. This value is the major PTX version $*$ 10 + the minor PTX version, so a PTX version 1.3 function would return the value 13. Note that this may return the undefined value of 0 for cubins compiled prior to CUDA 3.0.

CU_FUNC_ATTRIBUTE_BINARY_VERSION The binary architecture version for which the function was compiled. This value is the major binary version $*$ 10 + the minor binary version, so a binary version 1.3 function would return the value 13. Note that this will return a value of 10 for legacy cubins that do not have a properly-encoded binary architecture version.

5.29.3.12 enum CUgraphicsMapResourceFlags_enum

Flags for mapping and unmapping interop resources

5.29.3.13 enum CUgraphicsRegisterFlags_enum

Flags to register a graphics resource

5.29.3.14 enum CUipcMem_flags_enum

CUDA Ipc Mem Flags

Enumerator:

CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS Automatically enable peer access between remote devices as needed

5.29.3.15 enum CUjit_fallback_enum

Cubin matching fallback strategies

Enumerator:

CU_PREFER_PTX Prefer to compile ptx

CU_PREFER_BINARY Prefer to fall back to compatible binary code

5.29.3.16 enum CUjit_option_enum

Online compiler options

Enumerator:

CU_JIT_MAX_REGISTERS Max number of registers that a thread may use.

Option type: unsigned int

CU_JIT_THREADS_PER_BLOCK IN: Specifies minimum number of threads per block to target compilation for

OUT: Returns the number of threads the compiler actually targeted. This restricts the resource utilization fo the compiler (e.g. max registers) such that a block with the given number of threads should be able to launch based on register limitations. Note, this option does not currently take into account any other resource limitations, such as shared memory utilization.

Option type: unsigned int

CU_JIT_WALL_TIME Returns a float value in the option of the wall clock time, in milliseconds, spent creating the cubin

Option type: float

CU_JIT_INFO_LOG_BUFFER Pointer to a buffer in which to print any log messsages from PTXAS that are informational in nature (the buffer size is specified via option CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES)

Option type: char*

CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES IN: Log buffer size in bytes. Log messages will be capped at this size (including null terminator)

OUT: Amount of log buffer filled with messages

Option type: unsigned int

CU_JIT_ERROR_LOG_BUFFER Pointer to a buffer in which to print any log messages from PTXAS that reflect errors (the buffer size is specified via option CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES)

Option type: char*

CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES IN: Log buffer size in bytes. Log messages will be capped at this size (including null terminator)

OUT: Amount of log buffer filled with messages

Option type: unsigned int

CU_JIT_OPTIMIZATION_LEVEL Level of optimizations to apply to generated code (0 - 4), with 4 being the default and highest level of optimizations.

Option type: unsigned int

CU_JIT_TARGET_FROM_CUCONTEXT No option value required. Determines the target based on the current attached context (default)

Option type: No option value needed

CU_JIT_TARGET Target is chosen based on supplied CUjit_target_enum.
Option type: unsigned int for enumerated type CUjit_target_enum

CU_JIT_FALLBACK_STRATEGY Specifies choice of fallback strategy if matching cubin is not found. Choice is based on supplied CUjit_fallback_enum.
Option type: unsigned int for enumerated type CUjit_fallback_enum

5.29.3.17 enum CUjit_target_enum

Online compilation targets

Enumerator:

CU_TARGET_COMPUTE_10 Compute device class 1.0

CU_TARGET_COMPUTE_11 Compute device class 1.1

CU_TARGET_COMPUTE_12 Compute device class 1.2

CU_TARGET_COMPUTE_13 Compute device class 1.3

CU_TARGET_COMPUTE_20 Compute device class 2.0

CU_TARGET_COMPUTE_21 Compute device class 2.1

CU_TARGET_COMPUTE_30 Compute device class 3.0

CU_TARGET_COMPUTE_35 Compute device class 3.5

5.29.3.18 enum CUlimit_enum

Limits

Enumerator:

CU_LIMIT_STACK_SIZE GPU thread stack size

CU_LIMIT_PRINTF_FIFO_SIZE GPU printf FIFO size

CU_LIMIT_MALLOC_HEAP_SIZE GPU malloc heap size

CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH GPU device runtime launch synchronize depth

CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT GPU device runtime pending launch count

5.29.3.19 enum CUmemorytype_enum

Memory types

Enumerator:

CU_MEMORYTYPE_HOST Host memory

CU_MEMORYTYPE_DEVICE Device memory

CU_MEMORYTYPE_ARRAY Array memory

CU_MEMORYTYPE_UNIFIED Unified device or host memory

5.29.3.20 enum CUpointer_attribute_enum

Pointer information

Enumerator:

CU_POINTER_ATTRIBUTE_CONTEXT The CUcontext on which a pointer was allocated or registered

CU_POINTER_ATTRIBUTE_MEMORY_TYPE The CUmemorytype describing the physical location of a pointer

CU_POINTER_ATTRIBUTE_DEVICE_POINTER The address at which a pointer's memory may be accessed on the device

CU_POINTER_ATTRIBUTE_HOST_POINTER The address at which a pointer's memory may be accessed on the host

CU_POINTER_ATTRIBUTE_P2P_TOKENS A pair of tokens for use with the nv-p2p.h Linux kernel interface

5.29.3.21 enum CUresourcetype_enum

Resource types

Enumerator:

CU_RESOURCE_TYPE_ARRAY Array resoure

CU_RESOURCE_TYPE_MIPMAPPED_ARRAY Mipmapped array resource

CU_RESOURCE_TYPE_LINEAR Linear resource

CU_RESOURCE_TYPE_PITCH2D Pitch 2D resource

5.29.3.22 enum CUresourceViewFormat_enum

Resource view format

Enumerator:

CU_RES_VIEW_FORMAT_NONE No resource view format (use underlying resource format)

CU_RES_VIEW_FORMAT_UINT_1X8 1 channel unsigned 8-bit integers

CU_RES_VIEW_FORMAT_UINT_2X8 2 channel unsigned 8-bit integers

CU_RES_VIEW_FORMAT_UINT_4X8 4 channel unsigned 8-bit integers

CU_RES_VIEW_FORMAT_SINT_1X8 1 channel signed 8-bit integers

CU_RES_VIEW_FORMAT_SINT_2X8 2 channel signed 8-bit integers

CU_RES_VIEW_FORMAT_SINT_4X8 4 channel signed 8-bit integers

CU_RES_VIEW_FORMAT_UINT_1X16 1 channel unsigned 16-bit integers

CU_RES_VIEW_FORMAT_UINT_2X16 2 channel unsigned 16-bit integers

CU_RES_VIEW_FORMAT_UINT_4X16 4 channel unsigned 16-bit integers

CU_RES_VIEW_FORMAT_SINT_1X16 1 channel signed 16-bit integers

CU_RES_VIEW_FORMAT_SINT_2X16 2 channel signed 16-bit integers

CU_RES_VIEW_FORMAT_SINT_4X16 4 channel signed 16-bit integers

CU_RES_VIEW_FORMAT_UINT_1X32 1 channel unsigned 32-bit integers

CU_RES_VIEW_FORMAT_UINT_2X32 2 channel unsigned 32-bit integers

CU_RES_VIEW_FORMAT_UINT_4X32 4 channel unsigned 32-bit integers
CU_RES_VIEW_FORMAT_SINT_1X32 1 channel signed 32-bit integers
CU_RES_VIEW_FORMAT_SINT_2X32 2 channel signed 32-bit integers
CU_RES_VIEW_FORMAT_SINT_4X32 4 channel signed 32-bit integers
CU_RES_VIEW_FORMAT_FLOAT_1X16 1 channel 16-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_2X16 2 channel 16-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_4X16 4 channel 16-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_1X32 1 channel 32-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_2X32 2 channel 32-bit floating point
CU_RES_VIEW_FORMAT_FLOAT_4X32 4 channel 32-bit floating point
CU_RES_VIEW_FORMAT_UNSIGNED_BC1 Block compressed 1
CU_RES_VIEW_FORMAT_UNSIGNED_BC2 Block compressed 2
CU_RES_VIEW_FORMAT_UNSIGNED_BC3 Block compressed 3
CU_RES_VIEW_FORMAT_UNSIGNED_BC4 Block compressed 4 unsigned
CU_RES_VIEW_FORMAT_SIGNED_BC4 Block compressed 4 signed
CU_RES_VIEW_FORMAT_UNSIGNED_BC5 Block compressed 5 unsigned
CU_RES_VIEW_FORMAT_SIGNED_BC5 Block compressed 5 signed
CU_RES_VIEW_FORMAT_UNSIGNED_BC6H Block compressed 6 unsigned half-float
CU_RES_VIEW_FORMAT_SIGNED_BC6H Block compressed 6 signed half-float
CU_RES_VIEW_FORMAT_UNSIGNED_BC7 Block compressed 7

5.29.3.23 enum CUsharedconfig_enum

Shared memory configurations

Enumerator:

CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE set default shared memory bank size
CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE set shared memory bank width to four bytes
CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE set shared memory bank width to eight bytes

5.29.3.24 enum CUstream_flags_enum

Stream creation flags

Enumerator:

CU_STREAM_DEFAULT Default stream flag
CU_STREAM_NON_BLOCKING Stream does not synchronize with stream 0 (the NULL stream)

5.30 Initialization

Functions

- CUresult cuInit (unsigned int Flags)
  Initialize the CUDA driver API.

5.30.1 Detailed Description

This section describes the initialization functions of the low-level CUDA driver application programming interface.

5.30.2 Function Documentation

5.30.2.1 CUresult cuInit (unsigned int *Flags*)

Initializes the driver API and must be called before any other function from the driver API. Currently, the `Flags` parameter must be 0. If cuInit() has not been called, any function from the driver API will return CUDA_ERROR_NOT_INITIALIZED.

Parameters:

Flags - Initialization flag for CUDA.

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

5.31 Version Management

Functions

- CUresult cuDriverGetVersion (int ∗driverVersion)

  Returns the CUDA driver version.

5.31.1 Detailed Description

This section describes the version management functions of the low-level CUDA driver application programming interface.

5.31.2 Function Documentation

5.31.2.1 CUresult cuDriverGetVersion (int ∗ *driverVersion*)

Returns in `∗driverVersion` the version number of the installed CUDA driver. This function automatically returns CUDA_ERROR_INVALID_VALUE if the `driverVersion` argument is NULL.

Parameters:

driverVersion - Returns the CUDA driver version

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

5.32 Device Management

Functions

- CUresult cuDeviceGet (CUdevice *device, int ordinal)

  Returns a handle to a compute device.

- CUresult cuDeviceGetAttribute (int *pi, CUdevice_attribute attrib, CUdevice dev)

  Returns information about the device.

- CUresult cuDeviceGetCount (int *count)

  Returns the number of compute-capable devices.

- CUresult cuDeviceGetName (char *name, int len, CUdevice dev)

  Returns an identifer string for the device.

- CUresult cuDeviceTotalMem (size_t *bytes, CUdevice dev)

  Returns the total amount of memory on the device.

5.32.1 Detailed Description

This section describes the device management functions of the low-level CUDA driver application programming interface.

5.32.2 Function Documentation

5.32.2.1 CUresult cuDeviceGet (CUdevice * *device*, int *ordinal*)

Returns in `*device` a device handle given an ordinal in the range **[0, cuDeviceGetCount()-1]**.

Parameters:

device - Returned device handle

ordinal - Device number to get handle for

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceTotalMem

5.32.2.2 CUresult cuDeviceGetAttribute (int * *pi*, CUdevice_attribute *attrib*, CUdevice *dev*)

Returns in `*pi` the integer value of the attribute `attrib` on device `dev`. The supported attributes are:

- CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK: Maximum number of threads per block;
- CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X: Maximum x-dimension of a block;
- CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y: Maximum y-dimension of a block;
- CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z: Maximum z-dimension of a block;
- CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X: Maximum x-dimension of a grid;
- CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y: Maximum y-dimension of a grid;
- CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z: Maximum z-dimension of a grid;
- CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK: Maximum amount of shared memory available to a thread block in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;
- CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY: Memory available on device for __constant__ variables in a CUDA C kernel in bytes;
- CU_DEVICE_ATTRIBUTE_WARP_SIZE: Warp size in threads;
- CU_DEVICE_ATTRIBUTE_MAX_PITCH: Maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through cuMemAllocPitch();
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH: Maximum 1D texture width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH: Maximum width for a 1D texture bound to linear memory;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH: Maximum mipmapped 1D texture width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH: Maximum 2D texture width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT: Maximum 2D texture height;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH: Maximum width for a 2D texture bound to linear memory;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT: Maximum height for a 2D texture bound to linear memory;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH: Maximum pitch in bytes for a 2D texture bound to linear memory;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH: Maximum mipmapped 2D texture width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT: Maximum mipmapped 2D texture height;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH: Maximum 3D texture width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT: Maximum 3D texture height;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH: Maximum 3D texture depth;

- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH_ALTERNATE: Alternate maximum 3D texture width, 0 if no alternate maximum 3D texture size is supported;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE: Alternate maximum 3D texture height, 0 if no alternate maximum 3D texture size is supported;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE: Alternate maximum 3D texture depth, 0 if no alternate maximum 3D texture size is supported;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_WIDTH: Maximum cubemap texture width or height;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH: Maximum 1D layered texture width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS: Maximum layers in a 1D layered texture;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH: Maximum 2D layered texture width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT: Maximum 2D layered texture height;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS: Maximum layers in a 2D layered texture;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH: Maximum cubemap layered texture width or height;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_LAYERS: Maximum layers in a cubemap layered texture;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_WIDTH: Maximum 1D surface width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_WIDTH: Maximum 2D surface width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_HEIGHT: Maximum 2D surface height;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_WIDTH: Maximum 3D surface width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_HEIGHT: Maximum 3D surface height;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_DEPTH: Maximum 3D surface depth;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_WIDTH: Maximum 1D layered surface width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_LAYERS: Maximum layers in a 1D layered surface;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_WIDTH: Maximum 2D layered surface width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_HEIGHT: Maximum 2D layered surface height;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_LAYERS: Maximum layers in a 2D layered surface;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_WIDTH: Maximum cubemap surface width;

- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH: Maximum cubemap layered surface width;
- CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS: Maximum layers in a cubemap layered surface;
- CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK: Maximum number of 32-bit registers available to a thread block; this number is shared by all thread blocks simultaneously resident on a multiprocessor;
- CU_DEVICE_ATTRIBUTE_CLOCK_RATE: Peak clock frequency in kilohertz;
- CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT: Alignment requirement; texture base addresses aligned to textureAlign bytes do not need an offset applied to texture fetches;
- CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT: Pitch alignment requirement for 2D texture references bound to pitched memory;
- CU_DEVICE_ATTRIBUTE_GPU_OVERLAP: 1 if the device can concurrently copy memory between host and device while executing a kernel, or 0 if not;
- CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT: Number of multiprocessors on the device;
- CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT: 1 if there is a run time limit for kernels executed on the device, or 0 if not;
- CU_DEVICE_ATTRIBUTE_INTEGRATED: 1 if the device is integrated with the memory subsystem, or 0 if not;
- CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY: 1 if the device can map host memory into the CUDA address space, or 0 if not;
- CU_DEVICE_ATTRIBUTE_COMPUTE_MODE: Compute mode that device is currently in. Available modes are as follows:
  - CU_COMPUTEMODE_DEFAULT: Default mode - Device is not restricted and can have multiple CUDA contexts present at a single time.
  - CU_COMPUTEMODE_EXCLUSIVE: Compute-exclusive mode - Device can have only one CUDA context present on it at a time.
  - CU_COMPUTEMODE_PROHIBITED: Compute-prohibited mode - Device is prohibited from creating new CUDA contexts.
  - CU_COMPUTEMODE_EXCLUSIVE_PROCESS: Compute-exclusive-process mode - Device can have only one context used by a single process at a time.
- CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS: 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;
- CU_DEVICE_ATTRIBUTE_ECC_ENABLED: 1 if error correction is enabled on the device, 0 if error correction is disabled or not supported by the device;
- CU_DEVICE_ATTRIBUTE_PCI_BUS_ID: PCI bus identifier of the device;
- CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID: PCI device (also known as slot) identifier of the device;
- CU_DEVICE_ATTRIBUTE_TCC_DRIVER: 1 if the device is using a TCC driver. TCC is only available on Tesla hardware running Windows Vista or later;
- CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE: Peak memory clock frequency in kilohertz;
- CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH: Global memory bus width in bits;

- CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE: Size of L2 cache in bytes. 0 if the device doesn't have L2 cache;
- CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR: Maximum resident threads per multiprocessor;
- CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING: 1 if the device shares a unified address space with the host, or 0 if not;
- CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR: Major compute capability version number;
- CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR: Minor compute capability version number;

Parameters:

pi - Returned device attribute value

attrib - Device attribute to query

dev - Device handle

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuDeviceGetCount, cuDeviceGetName, cuDeviceGet, cuDeviceTotalMem

5.32.2.3 CUresult cuDeviceGetCount (int * *count*)

Returns in `*count` the number of devices with compute capability greater than or equal to 1.0 that are available for execution. If there is no such device, cuDeviceGetCount() returns 0.

Parameters:

count - Returned number of compute-capable devices

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuDeviceGetAttribute, cuDeviceGetName, cuDeviceGet, cuDeviceTotalMem

5.32.2.4 CUresult cuDeviceGetName (char * *name*, int *len*, CUdevice *dev*)

Returns an ASCII string identifying the device `dev` in the NULL-terminated string pointed to by `name`. `len` specifies the maximum length of the string that may be returned.

Parameters:

name - Returned identifier string for the device

len - Maximum length of string to store in `name`

dev - Device to get identifier string for

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGet, cuDeviceTotalMem

5.32.2.5 CUresult cuDeviceTotalMem (size_t * *bytes*, CUdevice *dev*)

Returns in `*bytes` the total amount of memory available on the device `dev` in bytes.

Parameters:

bytes - Returned memory available on device in bytes

dev - Device handle

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGet,

5.33 Device Management [DEPRECATED]

Functions

- CUresult cuDeviceComputeCapability (int *major, int *minor, CUdevice dev)

  Returns the compute capability of the device.

- CUresult cuDeviceGetProperties (CUdevprop *prop, CUdevice dev)

  Returns properties for a selected device.

5.33.1 Detailed Description

This section describes the device management functions of the low-level CUDA driver application programming interface.

5.33.2 Function Documentation

5.33.2.1 CUresult cuDeviceComputeCapability (int * *major*, int * *minor*, CUdevice *dev*)

Deprecated

This function was deprecated as of CUDA 5.0 and its functionality superceded by cuDeviceGetAttribute().

Returns in `*major` and `*minor` the major and minor revision numbers that define the compute capability of the device `dev`.

Parameters:

major - Major revision number

minor - Minor revision number

dev - Device handle

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGet, cuDeviceTotalMem

5.33.2.2 CUresult cuDeviceGetProperties (CUdevprop * *prop*, CUdevice *dev*)

Deprecated

This function was deprecated as of CUDA 5.0 and replaced by cuDeviceGetAttribute().

Returns in `*prop` the properties of device `dev`. The CUdevprop structure is defined as:

```
typedef struct CUdevprop_st {
int maxThreadsPerBlock;
int maxThreadsDim[3];
int maxGridSize[3];
int sharedMemPerBlock;
int totalConstantMemory;
int SIMDWidth;
int memPitch;
int regsPerBlock;
int clockRate;
int textureAlign
} CUdevprop;
```

where:

- maxThreadsPerBlock is the maximum number of threads per block;
- maxThreadsDim[3] is the maximum sizes of each dimension of a block;
- maxGridSize[3] is the maximum sizes of each dimension of a grid;
- sharedMemPerBlock is the total amount of shared memory available per block in bytes;
- totalConstantMemory is the total amount of constant memory available on the device in bytes;
- SIMDWidth is the warp size;
- memPitch is the maximum pitch allowed by the memory copy functions that involve memory regions allocated through cuMemAllocPitch();
- regsPerBlock is the total number of registers available per block;
- clockRate is the clock frequency in kilohertz;
- textureAlign is the alignment requirement; texture base addresses that are aligned to textureAlign bytes do not need an offset applied to texture fetches.

Parameters:

prop - Returned properties of device

dev - Device to get properties for

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGet, cuDeviceTotalMem

5.34 Context Management

Functions

- CUresult cuCtxCreate (CUcontext *pctx, unsigned int flags, CUdevice dev)

  Create a CUDA context.

- CUresult cuCtxDestroy (CUcontext ctx)

  Destroy a CUDA context.

- CUresult cuCtxGetApiVersion (CUcontext ctx, unsigned int *version)

  Gets the context's API version.

- CUresult cuCtxGetCacheConfig (CUfunc_cache *pconfig)

  Returns the preferred cache configuration for the current context.

- CUresult cuCtxGetCurrent (CUcontext *pctx)

  Returns the CUDA context bound to the calling CPU thread.

- CUresult cuCtxGetDevice (CUdevice *device)

  Returns the device ID for the current context.

- CUresult cuCtxGetLimit (size_t *pvalue, CUlimit limit)

  Returns resource limits.

- CUresult cuCtxGetSharedMemConfig (CUsharedconfig *pConfig)

  Returns the current shared memory configuration for the current context.

- CUresult cuCtxPopCurrent (CUcontext *pctx)

  Pops the current CUDA context from the current CPU thread.

- CUresult cuCtxPushCurrent (CUcontext ctx)

  Pushes a context on the current CPU thread.

- CUresult cuCtxSetCacheConfig (CUfunc_cache config)

  Sets the preferred cache configuration for the current context.

- CUresult cuCtxSetCurrent (CUcontext ctx)

  Binds the specified CUDA context to the calling CPU thread.

- CUresult cuCtxSetLimit (CUlimit limit, size_t value)

  Set resource limits.

- CUresult cuCtxSetSharedMemConfig (CUsharedconfig config)

  Sets the shared memory configuration for the current context.

- CUresult cuCtxSynchronize (void)

  Block for a context's tasks to complete.

5.34.1 Detailed Description

This section describes the context management functions of the low-level CUDA driver application programming interface.

5.34.2 Function Documentation

5.34.2.1 CUresult cuCtxCreate (CUcontext * *pctx*, unsigned int *flags*, CUdevice *dev*)

Creates a new CUDA context and associates it with the calling thread. The `flags` parameter is described below. The context is created with a usage count of 1 and the caller of cuCtxCreate() must call cuCtxDestroy() or when done using the context. If a context is already current to the thread, it is supplanted by the newly created context and may be restored by a subsequent call to cuCtxPopCurrent().

The three LSBs of the `flags` parameter can be used to control how the OS thread, which owns the CUDA context at the time of an API call, interacts with the OS scheduler when waiting for results from the GPU. Only one of the scheduling flags can be set when creating a context.

- CU_CTX_SCHED_AUTO: The default value if the `flags` parameter is zero, uses a heuristic based on the number of active CUDA contexts in the process C and the number of logical processors in the system P. If $C > P$, then CUDA will yield to other OS threads when waiting for the GPU, otherwise CUDA will not yield while waiting for results and actively spin on the processor.

- CU_CTX_SCHED_SPIN: Instruct CUDA to actively spin when waiting for results from the GPU. This can decrease latency when waiting for the GPU, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.

- CU_CTX_SCHED_YIELD: Instruct CUDA to yield its thread when waiting for results from the GPU. This can increase latency when waiting for the GPU, but can increase the performance of CPU threads performing work in parallel with the GPU.

- CU_CTX_SCHED_BLOCKING_SYNC: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.

- CU_CTX_BLOCKING_SYNC: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.

  Deprecated: This flag was deprecated as of CUDA 4.0 and was replaced with CU_CTX_SCHED_BLOCKING_SYNC.

- CU_CTX_MAP_HOST: Instruct CUDA to support mapped pinned allocations. This flag must be set in order to allocate pinned host memory that is accessible to the GPU.

- CU_CTX_LMEM_RESIZE_TO_MAX: Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage.

Context creation will fail with CUDA_ERROR_UNKNOWN if the compute mode of the device is CU_COMPUTEMODE_PROHIBITED. Similarly, context creation will also fail with CUDA_ERROR_UNKNOWN if the compute mode for the device is set to CU_COMPUTEMODE_EXCLUSIVE and there is already an active context on the device. The function cuDeviceGetAttribute() can be used with CU_DEVICE_ATTRIBUTE_COMPUTE_MODE to determine the compute mode of the device. The *nvidia-smi* tool can be used to set the compute mode for devices. Documentation for *nvidia-smi* can be obtained by passing a -h option to it.

Parameters:

pctx - Returned context handle of the new context

flags - Context creation flags

dev - Device to create context on

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

5.34.2.2 CUresult cuCtxDestroy (CUcontext *ctx*)

Destroys the CUDA context specified by `ctx`. The context `ctx` will be destroyed regardless of how many threads it is current to. It is the responsibility of the calling function to ensure that no API call issues using `ctx` while cuCtxDestroy() is executing.

If `ctx` is current to the calling thread then `ctx` will also be popped from the current thread's context stack (as though cuCtxPopCurrent() were called). If `ctx` is current to other threads, then `ctx` will remain current to those threads, and attempting to access `ctx` from those threads will result in the error CUDA_ERROR_CONTEXT_IS_DESTROYED.

Parameters:

ctx - Context to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

5.34.2.3 CUresult cuCtxGetApiVersion (CUcontext *ctx*, unsigned int * *version*)

Returns a version number in `version` corresponding to the capabilities of the context (e.g. 3010 or 3020), which library developers can use to direct callers to a specific API version. If `ctx` is NULL, returns the API version used to create the currently bound context.

Note that new API versions are only introduced when context capabilities are changed that break binary compatibility, so the API version and driver version may be different. For example, it is valid for the API version to be 3020 while the driver version is 4020.

Parameters:

ctx - Context to check

version - Pointer to version

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSet-CacheConfig, cuCtxSetLimit, cuCtxSynchronize

5.34.2.4 CUresult cuCtxGetCacheConfig (CUfunc_cache * *pconfig*)

On devices where the L1 cache and shared memory use the same hardware resources, this function returns through `pconfig` the preferred cache configuration for the current context. This is only a preference. The driver will use the requested configuration if possible, but it is free to choose a different configuration if required to execute functions.

This will return a `pconfig` of CU_FUNC_CACHE_PREFER_NONE on devices where the size of the L1 cache and shared memory are fixed.

The supported cache configurations are:

- CU_FUNC_CACHE_PREFER_NONE: no preference for shared memory or L1 (default)
- CU_FUNC_CACHE_PREFER_SHARED: prefer larger shared memory and smaller L1 cache
- CU_FUNC_CACHE_PREFER_L1: prefer larger L1 cache and smaller shared memory
- CU_FUNC_CACHE_PREFER_EQUAL: prefer equal sized L1 cache and shared memory

Parameters:

pconfig - Returned cache configuration

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtx-PushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize, cuFuncSetCacheConfig

5.34.2.5 CUresult cuCtxGetCurrent (CUcontext * *pctx*)

Returns in `*pctx` the CUDA context bound to the calling CPU thread. If no context is bound to the calling CPU thread then `*pctx` is set to NULL and CUDA_SUCCESS is returned.

Parameters:

pctx - Returned context handle

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxSetCurrent, cuCtxCreate, cuCtxDestroy

5.34.2.6 CUresult cuCtxGetDevice (CUdevice * *device*)

Returns in `*device` the ordinal of the current context's device.

Parameters:

device - Returned device ID for the current context

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

5.34.2.7 CUresult cuCtxGetLimit (size_t * *pvalue*, CUlimit *limit*)

Returns in `*pvalue` the current size of `limit`. The supported CUlimit values are:

- CU_LIMIT_STACK_SIZE: stack size in bytes of each GPU thread.
- CU_LIMIT_PRINTF_FIFO_SIZE: size in bytes of the FIFO used by the printf() device system call.
- CU_LIMIT_MALLOC_HEAP_SIZE: size in bytes of the heap used by the malloc() and free() device system calls.
- CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH: maximum grid depth at which a thread can issue the device runtime call cudaDeviceSynchronize() to wait on child grid launches to complete.

- CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT: maximum number of outstanding device runtime launches that can be made from this context.

Parameters:

limit - Limit to query

pvalue - Returned size of limit

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_UNSUPPORTED_LIMIT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

5.34.2.8 CUresult cuCtxGetSharedMemConfig (CUsharedconfig * *pConfig*)

This function will return in `pConfig` the current size of shared memory banks in the current context. On devices with configurable shared memory banks, cuCtxSetSharedMemConfig can be used to change this setting, so that all subsequent kernel launches will by default use the new bank size. When cuCtxGetSharedMemConfig is called on devices without configurable shared memory, it will return the fixed bank size of the hardware.

The returned bank configurations can be either:

- CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE: shared memory bank width is four bytes.
- CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE: shared memory bank width will eight bytes.

Parameters:

pConfig - returned shared memory configuration

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetLimit, cuCtxSynchronize, cuCtxGetSharedMemConfig, cuFuncSetCacheConfig,

5.34.2.9 CUresult cuCtxPopCurrent (CUcontext * *pctx*)

Pops the current CUDA context from the CPU thread and passes back the old context handle in `*pctx`. That context may then be made current to a different CPU thread by calling cuCtxPushCurrent().

If a context was current to the CPU thread before cuCtxCreate() or cuCtxPushCurrent() was called, this function makes that context current to the CPU thread again.

Parameters:

pctx - Returned new context handle

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

5.34.2.10 CUresult cuCtxPushCurrent (CUcontext *ctx*)

Pushes the given context `ctx` onto the CPU thread's stack of current contexts. The specified context becomes the CPU thread's current context, so all CUDA functions that operate on the current context are affected.

The previous current context may be made current again by calling cuCtxDestroy() or cuCtxPopCurrent().

Parameters:

ctx - Context to push

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

5.34.2.11 CUresult cuCtxSetCacheConfig (CUfunc_cache *config*)

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `config` the preferred cache configuration for the current context. This is only a preference. The driver will use the requested

configuration if possible, but it is free to choose a different configuration if required to execute the function. Any function preference set via cuFuncSetCacheConfig() will be preferred over this context-wide setting. Setting the context-wide cache configuration to CU_FUNC_CACHE_PREFER_NONE will cause subsequent kernel launches to prefer to not change the cache configuration unless required to launch the kernel.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- CU_FUNC_CACHE_PREFER_NONE: no preference for shared memory or L1 (default)
- CU_FUNC_CACHE_PREFER_SHARED: prefer larger shared memory and smaller L1 cache
- CU_FUNC_CACHE_PREFER_L1: prefer larger L1 cache and smaller shared memory
- CU_FUNC_CACHE_PREFER_EQUAL: prefer equal sized L1 cache and shared memory

Parameters:

config - Requested cache configuration

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetLimit, cuCtxSynchronize, cuFuncSetCacheConfig

5.34.2.12 CUresult cuCtxSetCurrent (CUcontext *ctx*)

Binds the specified CUDA context to the calling CPU thread. If `ctx` is NULL then the CUDA context previously bound to the calling CPU thread is unbound and CUDA_SUCCESS is returned.

If there exists a CUDA context stack on the calling CPU thread, this will replace the top of that stack with `ctx`. If `ctx` is NULL then this will be equivalent to popping the top of the calling CPU thread's CUDA context stack (or a no-op if the calling CPU thread's CUDA context stack is empty).

Parameters:

ctx - Context to bind to the calling CPU thread

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxGetCurrent, cuCtxCreate, cuCtxDestroy

5.34.2.13 CUresult cuCtxSetLimit (CUlimit *limit*, size_t *value*)

Setting `limit` to `value` is a request by the application to update the current limit maintained by the context. The driver is free to modify the requested value to meet h/w requirements (this could be clamping to minimum or maximum values, rounding up to nearest element size, etc). The application can use cuCtxGetLimit() to find out exactly what the limit has been set to.

Setting each CUlimit has its own specific restrictions, so each is discussed here.

- CU_LIMIT_STACK_SIZE controls the stack size in bytes of each GPU thread. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

- CU_LIMIT_PRINTF_FIFO_SIZE controls the size in bytes of the FIFO used by the printf() device system call. Setting CU_LIMIT_PRINTF_FIFO_SIZE must be performed before launching any kernel that uses the printf() device system call, otherwise CUDA_ERROR_INVALID_VALUE will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

- CU_LIMIT_MALLOC_HEAP_SIZE controls the size in bytes of the heap used by the malloc() and free() device system calls. Setting CU_LIMIT_MALLOC_HEAP_SIZE must be performed before launching any kernel that uses the malloc() or free() device system calls, otherwise CUDA_ERROR_INVALID_VALUE will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

- CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH controls the maximum nesting depth of a grid at which a thread can safely call cudaDeviceSynchronize(). Setting this limit must be performed before any launch of a kernel that uses the device runtime and calls cudaDeviceSynchronize() above the default sync depth, two levels of grids. Calls to cudaDeviceSynchronize() will fail with error code cudaErrorSyncDepthExceeded if the limitation is violated. This limit can be set smaller than the default or up the maximum launch depth of 24. When setting this limit, keep in mind that additional levels of sync depth require the driver to reserve large amounts of device memory which can no longer be used for user allocations. If these reservations of device memory fail, cuCtxSetLimit will return CUDA_ERROR_OUT_OF_MEMORY, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

- CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT controls the maximum number of outstanding device runtime launches that can be made from the current context. A grid is outstanding from the point of launch up until the grid is known to have been completed. Device runtime launches which violate this limitation fail and return cudaErrorLaunchPendingCountExceeded when cudaGetLastError() is called after launch. If more pending launches than the default (2048 launches) are needed for a module using the device runtime, this limit can be increased. Keep in mind that being able to sustain additional pending launches will require the driver to reserve larger amounts of device memory upfront which can no longer be used for allocations. If these reservations fail, cuCtxSetLimit will return CUDA_ERROR_OUT_OF_MEMORY, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

Parameters:

limit - Limit to set

value - Size of limit

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_UNSUPPORTED_LIMIT, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSynchronize

5.34.2.14 CUresult cuCtxSetSharedMemConfig (CUsharedconfig *config*)

On devices with configurable shared memory banks, this function will set the context's shared memory bank size which is used for subsequent kernel launches.

Changed the shared memory configuration between launches may insert a device side synchronization point between those launches.

Changing the shared memory bank size will not increase shared memory usage or affect occupancy of kernels, but may have major effects on performance. Larger bank sizes will allow for greater potential bandwidth to shared memory, but will change what kinds of accesses to shared memory will result in bank conflicts.

This function will do nothing on devices with fixed shared memory bank size.

The supported bank configurations are:

- CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE: set bank width to the default initial setting (currently, four bytes).
- CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE: set shared memory bank width to be natively four bytes.
- CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE: set shared memory bank width to be natively eight bytes.

Parameters:

config - requested shared memory configuration

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetLimit, cuCtxSynchronize, cuCtxGetSharedMemConfig, cuFuncSetCacheConfig,

5.34.2.15 CUresult cuCtxSynchronize (void)

Blocks until the device has completed all preceding requested tasks. cuCtxSynchronize() returns an error if one of the preceding tasks failed. If the context was created with the CU_CTX_SCHED_BLOCKING_SYNC flag, the CPU thread will block until the GPU context has finished its work.

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent cuCtxSetCacheConfig, cuCtxSetLimit

5.35 Context Management [DEPRECATED]

Functions

- CUresult cuCtxAttach (CUcontext *pctx, unsigned int flags)

  Increment a context's usage-count.

- CUresult cuCtxDetach (CUcontext ctx)

  Decrement a context's usage-count.

5.35.1 Detailed Description

This section describes the deprecated context management functions of the low-level CUDA driver application programming interface.

5.35.2 Function Documentation

5.35.2.1 CUresult cuCtxAttach (CUcontext * *pctx*, unsigned int *flags*)

Deprecated

Note that this function is deprecated and should not be used.

Increments the usage count of the context and passes back a context handle in `*pctx` that must be passed to cuCtxDetach() when the application is done with the context. cuCtxAttach() fails if there is no context current to the thread.

Currently, the `flags` parameter must be 0.

Parameters:

pctx - Returned context handle of the current context

flags - Context attach flags (must be 0)

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxDetach, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

5.35.2.2 CUresult cuCtxDetach (CUcontext *ctx*)

Deprecated

Note that this function is deprecated and should not be used.

Decrements the usage count of the context `ctx`, and destroys the context if the usage count goes to 0. The context must be a handle that was passed back by cuCtxCreate() or cuCtxAttach(), and must be current to the calling thread.

Parameters:

ctx - Context to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

5.36 Module Management

Functions

- CUresult cuModuleGetFunction (CUfunction *hfunc, CUmodule hmod, const char *name)
  Returns a function handle.
- CUresult cuModuleGetGlobal (CUdeviceptr *dptr, size_t *bytes, CUmodule hmod, const char *name)
  Returns a global pointer from a module.
- CUresult cuModuleGetSurfRef (CUsurfref *pSurfRef, CUmodule hmod, const char *name)
  Returns a handle to a surface reference.
- CUresult cuModuleGetTexRef (CUtexref *pTexRef, CUmodule hmod, const char *name)
  Returns a handle to a texture reference.
- CUresult cuModuleLoad (CUmodule *module, const char *fname)
  Loads a compute module.
- CUresult cuModuleLoadData (CUmodule *module, const void *image)
  Load a module's data.
- CUresult cuModuleLoadDataEx (CUmodule *module, const void *image, unsigned int numOptions, CUjit_option *options, void **optionValues)
  Load a module's data with options.
- CUresult cuModuleLoadFatBinary (CUmodule *module, const void *fatCubin)
  Load a module's data.
- CUresult cuModuleUnload (CUmodule hmod)
  Unloads a module.

5.36.1 Detailed Description

This section describes the module management functions of the low-level CUDA driver application programming interface.

5.36.2 Function Documentation

5.36.2.1 CUresult cuModuleGetFunction (CUfunction * *hfunc*, CUmodule *hmod*, const char * *name*)

Returns in `*hfunc` the handle of the function of name `name` located in module `hmod`. If no function of that name exists, cuModuleGetFunction() returns CUDA_ERROR_NOT_FOUND.

Parameters:

hfunc - Returned function handle

hmod - Module to retrieve function from

name - Name of function to retrieve

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cu-ModuleLoadFatBinary, cuModuleUnload

5.36.2.2 CUresult cuModuleGetGlobal (CUdeviceptr * *dptr*, size_t * *bytes*, CUmodule *hmod*, const char * *name*)

Returns in `*dptr` and `*bytes` the base pointer and size of the global of name `name` located in module `hmod`. If no variable of that name exists, cuModuleGetGlobal() returns CUDA_ERROR_NOT_FOUND. Both parameters `dptr` and `bytes` are optional. If one of them is NULL, it is ignored.

Parameters:

dptr - Returned global device pointer

bytes - Returned global size in bytes

hmod - Module to retrieve global from

name - Name of global to retrieve

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuModuleGetFunction, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cu-ModuleLoadFatBinary, cuModuleUnload

5.36.2.3 CUresult cuModuleGetSurfRef (CUsurfref * *pSurfRef*, CUmodule *hmod*, const char * *name*)

Returns in `*pSurfRef` the handle of the surface reference of name `name` in the module `hmod`. If no surface reference of that name exists, cuModuleGetSurfRef() returns CUDA_ERROR_NOT_FOUND.

Parameters:

pSurfRef - Returned surface reference

hmod - Module to retrieve surface reference from

name - Name of surface reference to retrieve

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cu-ModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

5.36.2.4 CUresult cuModuleGetTexRef (CUtexref * *pTexRef*, CUmodule *hmod*, const char * *name*)

Returns in `*pTexRef` the handle of the texture reference of name `name` in the module `hmod`. If no texture reference of that name exists, cuModuleGetTexRef() returns CUDA_ERROR_NOT_FOUND. This texture reference handle should not be destroyed, since it will be destroyed when the module is unloaded.

Parameters:

pTexRef - Returned texture reference

hmod - Module to retrieve texture reference from

name - Name of texture reference to retrieve

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetSurfRef, cuModuleLoad, cuModuleLoadData, cu-ModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

5.36.2.5 CUresult cuModuleLoad (CUmodule * *module*, const char * *fname*)

Takes a filename `fname` and loads the corresponding module `module` into the current context. The CUDA driver API does not attempt to lazily allocate the resources needed by a module; if the memory for functions and data (constant and global) needed by the module cannot be allocated, cuModuleLoad() fails. The file should be a *cubin* file as output by **nvcc**, or a *PTX* file either as output by **nvcc** or handwritten, or a *fatbin* file as output by **nvcc** from toolchain 4.0 or later.

Parameters:

module - Returned module

fname - Filename of module to load

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_FILE_NOT_FOUND, CUDA_ERROR_SHARED_-OBJECT_SYMBOL_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

5.36.2.6 CUresult cuModuleLoadData (CUmodule * *module*, const void * *image*)

Takes a pointer `image` and loads the corresponding module `module` into the current context. The pointer may be obtained by mapping a *cubin* or *PTX* or *fatbin* file, passing a *cubin* or *PTX* or *fatbin* file as a NULL-terminated text string, or incorporating a *cubin* or *fatbin* object into the executable resources and using operating system calls such as Windows `FindResource()` to obtain the pointer.

Parameters:

module - Returned module

image - Module data to load

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_-INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadDataEx, cu-ModuleLoadFatBinary, cuModuleUnload

5.36.2.7 CUresult cuModuleLoadDataEx (CUmodule * *module*, const void * *image*, unsigned int *numOptions*, CUjit_option * *options*, void ** *optionValues*)

Takes a pointer `image` and loads the corresponding module `module` into the current context. The pointer may be obtained by mapping a *cubin* or *PTX* or *fatbin* file, passing a *cubin* or *PTX* or *fatbin* file as a NULL-terminated text string, or incorporating a *cubin* or *fatbin* object into the executable resources and using operating system calls such as Windows `FindResource()` to obtain the pointer. Options are passed as an array via `options` and any corresponding parameters are passed in `optionValues`. The number of total options is supplied via `numOptions`. Any outputs will be returned via `optionValues`. Supported options are (types for the option values are specified in parentheses after the option name):

- CU_JIT_MAX_REGISTERS: (unsigned int) input specifies the maximum number of registers per thread;
- CU_JIT_THREADS_PER_BLOCK: (unsigned int) input specifies number of threads per block to target compilation for; output returns the number of threads the compiler actually targeted;
- CU_JIT_WALL_TIME: (float) output returns the float value of wall clock time, in milliseconds, spent compiling the *PTX* code;
- CU_JIT_INFO_LOG_BUFFER: (char∗) input is a pointer to a buffer in which to print any informational log messages from *PTX* assembly (the buffer size is specified via option CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES);
- CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES: (unsigned int) input is the size in bytes of the buffer; output is the number of bytes filled with messages;
- CU_JIT_ERROR_LOG_BUFFER: (char∗) input is a pointer to a buffer in which to print any error log messages from *PTX* assembly (the buffer size is specified via option CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES);
- CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES: (unsigned int) input is the size in bytes of the buffer; output is the number of bytes filled with messages;
- CU_JIT_OPTIMIZATION_LEVEL: (unsigned int) input is the level of optimization to apply to generated code (0 - 4), with 4 being the default and highest level;
- CU_JIT_TARGET_FROM_CUCONTEXT: (No option value) causes compilation target to be determined based on current attached context (default);
- CU_JIT_TARGET: (unsigned int for enumerated type CUjit_target_enum) input is the compilation target based on supplied CUjit_target_enum; possible values are:
  - CU_TARGET_COMPUTE_10
  - CU_TARGET_COMPUTE_11
  - CU_TARGET_COMPUTE_12
  - CU_TARGET_COMPUTE_13
  - CU_TARGET_COMPUTE_20
- CU_JIT_FALLBACK_STRATEGY: (unsigned int for enumerated type CUjit_fallback_enum) chooses fallback strategy if matching cubin is not found; possible values are:
  - CU_PREFER_PTX
  - CU_PREFER_BINARY

Parameters:

module - Returned module

image - Module data to load

numOptions - Number of options

options - Options for JIT

optionValues - Option values for JIT

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NO_BINARY_FOR_GPU, CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadFatBinary, cuModuleUnload

5.36.2.8 CUresult cuModuleLoadFatBinary (CUmodule * *module*, const void * *fatCubin*)

Takes a pointer `fatCubin` and loads the corresponding module `module` into the current context. The pointer represents a *fat binary* object, which is a collection of different *cubin* and/or *PTX* files, all representing the same device code, but compiled and optimized for different architectures.

Prior to CUDA 4.0, there was no documented API for constructing and using fat binary objects by programmers. Starting with CUDA 4.0, fat binary objects can be constructed by providing the *-fatbin option* to **nvcc**. More information can be found in the **nvcc** document.

Parameters:

module - Returned module

fatCubin - Fat binary to load

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NO_BINARY_FOR_GPU, CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleUnload

5.36.2.9 CUresult cuModuleUnload (CUmodule *hmod*)

Unloads a module `hmod` from the current context.

Parameters:

hmod - Module to unload

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary

5.37 Memory Management

Functions

- CUresult cuArray3DCreate (CUarray *pHandle, const CUDA_ARRAY3D_DESCRIPTOR *pAllocateArray)

  Creates a 3D CUDA array.

- CUresult cuArray3DGetDescriptor (CUDA_ARRAY3D_DESCRIPTOR *pArrayDescriptor, CUarray hArray)

  Get a 3D CUDA array descriptor.

- CUresult cuArrayCreate (CUarray *pHandle, const CUDA_ARRAY_DESCRIPTOR *pAllocateArray)

  Creates a 1D or 2D CUDA array.

- CUresult cuArrayDestroy (CUarray hArray)

  Destroys a CUDA array.

- CUresult cuArrayGetDescriptor (CUDA_ARRAY_DESCRIPTOR *pArrayDescriptor, CUarray hArray)

  Get a 1D or 2D CUDA array descriptor.

- CUresult cuDeviceGetByPCIBusId (CUdevice *dev, char *pciBusId)

  Returns a handle to a compute device.

- CUresult cuDeviceGetPCIBusId (char *pciBusId, int len, CUdevice dev)

  Returns a PCI Bus Id string for the device.

- CUresult cuIpcCloseMemHandle (CUdeviceptr dptr)

  Close memory mapped with cuIpcOpenMemHandle.

- CUresult cuIpcGetEventHandle (CUipcEventHandle *pHandle, CUevent event)

  Gets an interprocess handle for a previously allocated event.

- CUresult cuIpcGetMemHandle (CUipcMemHandle *pHandle, CUdeviceptr dptr)
- CUresult cuIpcOpenEventHandle (CUevent *phEvent, CUipcEventHandle handle)

  Opens an interprocess event handle for use in the current process.

- CUresult cuIpcOpenMemHandle (CUdeviceptr *pdptr, CUipcMemHandle handle, unsigned int Flags)
- CUresult cuMemAlloc (CUdeviceptr *dptr, size_t bytesize)

  Allocates device memory.

- CUresult cuMemAllocHost (void **pp, size_t bytesize)

  Allocates page-locked host memory.

- CUresult cuMemAllocPitch (CUdeviceptr *dptr, size_t *pPitch, size_t WidthInBytes, size_t Height, unsigned int ElementSizeBytes)

  Allocates pitched device memory.

- CUresult cuMemcpy (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount)

  Copies memory.

- CUresult cuMemcpy2D (const CUDA_MEMCPY2D *pCopy)

  Copies memory for 2D arrays.

- CUresult cuMemcpy2DAsync (const CUDA_MEMCPY2D *pCopy, CUstream hStream)

  Copies memory for 2D arrays.

- CUresult cuMemcpy2DUnaligned (const CUDA_MEMCPY2D *pCopy)

  Copies memory for 2D arrays.

- CUresult cuMemcpy3D (const CUDA_MEMCPY3D *pCopy)

  Copies memory for 3D arrays.

- CUresult cuMemcpy3DAsync (const CUDA_MEMCPY3D *pCopy, CUstream hStream)

  Copies memory for 3D arrays.

- CUresult cuMemcpy3DPeer (const CUDA_MEMCPY3D_PEER *pCopy)

  Copies memory between contexts.

- CUresult cuMemcpy3DPeerAsync (const CUDA_MEMCPY3D_PEER *pCopy, CUstream hStream)

  Copies memory between contexts asynchronously.

- CUresult cuMemcpyAsync (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount, CUstream hStream)

  Copies memory asynchronously.

- CUresult cuMemcpyAtoA (CUarray dstArray, size_t dstOffset, CUarray srcArray, size_t srcOffset, size_t ByteCount)

  Copies memory from Array to Array.

- CUresult cuMemcpyAtoD (CUdeviceptr dstDevice, CUarray srcArray, size_t srcOffset, size_t ByteCount)

  Copies memory from Array to Device.

- CUresult cuMemcpyAtoH (void *dstHost, CUarray srcArray, size_t srcOffset, size_t ByteCount)

  Copies memory from Array to Host.

- CUresult cuMemcpyAtoHAsync (void *dstHost, CUarray srcArray, size_t srcOffset, size_t ByteCount, CUstream hStream)

  Copies memory from Array to Host.

- CUresult cuMemcpyDtoA (CUarray dstArray, size_t dstOffset, CUdeviceptr srcDevice, size_t ByteCount)

  Copies memory from Device to Array.

- CUresult cuMemcpyDtoD (CUdeviceptr dstDevice, CUdeviceptr srcDevice, size_t ByteCount)

  Copies memory from Device to Device.

- CUresult cuMemcpyDtoDAsync (CUdeviceptr dstDevice, CUdeviceptr srcDevice, size_t ByteCount, CUstream hStream)

  Copies memory from Device to Device.

- CUresult cuMemcpyDtoH (void *dstHost, CUdeviceptr srcDevice, size_t ByteCount)

  Copies memory from Device to Host.

- CUresult cuMemcpyDtoHAsync (void *dstHost, CUdeviceptr srcDevice, size_t ByteCount, CUstream hStream)

  Copies memory from Device to Host.

- CUresult cuMemcpyHtoA (CUarray dstArray, size_t dstOffset, const void *srcHost, size_t ByteCount)

  Copies memory from Host to Array.

- CUresult cuMemcpyHtoAAsync (CUarray dstArray, size_t dstOffset, const void *srcHost, size_t ByteCount, CUstream hStream)

  Copies memory from Host to Array.

- CUresult cuMemcpyHtoD (CUdeviceptr dstDevice, const void *srcHost, size_t ByteCount)

  Copies memory from Host to Device.

- CUresult cuMemcpyHtoDAsync (CUdeviceptr dstDevice, const void *srcHost, size_t ByteCount, CUstream hStream)

  Copies memory from Host to Device.

- CUresult cuMemcpyPeer (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount)

  Copies device memory between two contexts.

- CUresult cuMemcpyPeerAsync (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount, CUstream hStream)

  Copies device memory between two contexts asynchronously.

- CUresult cuMemFree (CUdeviceptr dptr)

  Frees device memory.

- CUresult cuMemFreeHost (void *p)

  Frees page-locked host memory.

- CUresult cuMemGetAddressRange (CUdeviceptr *pbase, size_t *psize, CUdeviceptr dptr)

  Get information on memory allocations.

- CUresult cuMemGetInfo (size_t *free, size_t *total)

  Gets free and total memory.

- CUresult cuMemHostAlloc (void **pp, size_t bytesize, unsigned int Flags)

  Allocates page-locked host memory.

- CUresult cuMemHostGetDevicePointer (CUdeviceptr *pdptr, void *p, unsigned int Flags)

  Passes back device pointer of mapped pinned memory.

- CUresult cuMemHostGetFlags (unsigned int *pFlags, void *p)

  Passes back flags that were used for a pinned allocation.

- CUresult cuMemHostRegister (void *p, size_t bytesize, unsigned int Flags)

  Registers an existing host memory range for use by CUDA.

- CUresult cuMemHostUnregister (void *p)

  Unregisters a memory range that was registered with cuMemHostRegister.

- CUresult cuMemsetD16 (CUdeviceptr dstDevice, unsigned short us, size_t N)

  Initializes device memory.

- CUresult cuMemsetD16Async (CUdeviceptr dstDevice, unsigned short us, size_t N, CUstream hStream)

  Sets device memory.

- CUresult cuMemsetD2D16 (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height)

  Initializes device memory.

- CUresult cuMemsetD2D16Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height, CUstream hStream)

  Sets device memory.

- CUresult cuMemsetD2D32 (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height)

  Initializes device memory.

- CUresult cuMemsetD2D32Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height, CUstream hStream)

  Sets device memory.

- CUresult cuMemsetD2D8 (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height)

  Initializes device memory.

- CUresult cuMemsetD2D8Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height, CUstream hStream)

  Sets device memory.

- CUresult cuMemsetD32 (CUdeviceptr dstDevice, unsigned int ui, size_t N)

  Initializes device memory.

- CUresult cuMemsetD32Async (CUdeviceptr dstDevice, unsigned int ui, size_t N, CUstream hStream)

  Sets device memory.

- CUresult cuMemsetD8 (CUdeviceptr dstDevice, unsigned char uc, size_t N)

  Initializes device memory.

- CUresult cuMemsetD8Async (CUdeviceptr dstDevice, unsigned char uc, size_t N, CUstream hStream)

  Sets device memory.

- CUresult cuMipmappedArrayCreate (CUmipmappedArray *pHandle, const CUDA_ARRAY3D_DESCRIPTOR *pMipmappedArrayDesc, unsigned int numMipmapLevels)

  Creates a CUDA mipmapped array.

- CUresult cuMipmappedArrayDestroy (CUmipmappedArray hMipmappedArray)

  Destroys a CUDA mipmapped array.

- CUresult cuMipmappedArrayGetLevel (CUarray *pLevelArray, CUmipmappedArray hMipmappedArray, unsigned int level)

  Gets a mipmap level of a CUDA mipmapped array.

5.37.1 Detailed Description

This section describes the memory management functions of the low-level CUDA driver application programming interface.

5.37.2 Function Documentation

5.37.2.1 CUresult cuArray3DCreate (CUarray * *pHandle*, const CUDA_ARRAY3D_DESCRIPTOR * *pAllocateArray*)

Creates a CUDA array according to the CUDA_ARRAY3D_DESCRIPTOR structure `pAllocateArray` and returns a handle to the new CUDA array in `*pHandle`. The CUDA_ARRAY3D_DESCRIPTOR is defined as:

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    unsigned int Depth;
    CUarray_format Format;
    unsigned int NumChannels;
    unsigned int Flags;
} CUDA_ARRAY3D_DESCRIPTOR;
```

where:

- `Width`, `Height`, and `Depth` are the width, height, and depth of the CUDA array (in elements); the following types of CUDA arrays can be allocated:
  - A 1D array is allocated if `Height` and `Depth` extents are both zero.
  - A 2D array is allocated if only `Depth` extent is zero.
  - A 3D array is allocated if all three extents are non-zero.
  - A 1D layered CUDA array is allocated if only `Height` is zero and the CUDA_ARRAY3D_LAYERED flag is set. Each layer is a 1D array. The number of layers is determined by the depth extent.
  - A 2D layered CUDA array is allocated if all three extents are non-zero and the CUDA_ARRAY3D_LAYERED flag is set. Each layer is a 2D array. The number of layers is determined by the depth extent.
  - A cubemap CUDA array is allocated if all three extents are non-zero and the CUDA_ARRAY3D_CUBEMAP flag is set. `Width` must be equal to `Height`, and `Depth` must be six. A cubemap is a special type of 2D layered CUDA array, where the six layers represent the six faces of a cube. The order of the six layers in memory is the same as that listed in CUarray_cubemap_face.
  - A cubemap layered CUDA array is allocated if all three extents are non-zero, and both, CUDA_ARRAY3D_CUBEMAP and CUDA_ARRAY3D_LAYERED flags are set. `Width` must be equal to `Height`, and `Depth` must be a multiple of six. A cubemap layered CUDA array is a special type of 2D layered CUDA array that consists of a collection of cubemaps. The first six layers represent the first cubemap, the next six layers form the second cubemap, and so on.
- Format specifies the format of the elements; CUarray_format is defined as:

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20
} CUarray_format;
```

- `NumChannels` specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;

- Flags may be set to

  - CUDA_ARRAY3D_LAYERED to enable creation of layered CUDA arrays. If this flag is set, `Depth` specifies the number of layers, not the depth of a 3D array.

  - CUDA_ARRAY3D_SURFACE_LDST to enable surface references to be bound to the CUDA array. If this flag is not set, cuSurfRefSetArray will fail when attempting to bind the CUDA array to a surface reference.

  - CUDA_ARRAY3D_CUBEMAP to enable creation of cubemaps. If this flag is set, `Width` must be equal to `Height`, and `Depth` must be six. If the CUDA_ARRAY3D_LAYERED flag is also set, then `Depth` must be a multiple of six.

  - CUDA_ARRAY3D_TEXTURE_GATHER to indicate that the CUDA array will be used for texture gather. Texture gather can only be performed on 2D CUDA arrays.

`Width`, `Height` and `Depth` must meet certain size requirements as listed in the following table. All values are specified in elements. Note that for brevity's sake, the full name of the device attribute is not specified. For ex., TEXTURE1D_WIDTH refers to the device attribute CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_-WIDTH.

Note that 2D CUDA arrays have different size requirements if the CUDA_ARRAY3D_TEXTURE_GATHER flag is set. `Width` and `Height` must not be greater than CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_-GATHER_WIDTH and CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_HEIGHT respectively, in that case.

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}	Valid extents with CUDA_ARRAY3D_SURFACE_LDST set {(width range in elements), (height range), (depth range)}
1D	{ (1,TEXTURE1D_WIDTH), 0, 0 }	{ (1,SURFACE1D_WIDTH), 0, 0 }
2D	{ (1,TEXTURE2D_WIDTH), (1,TEXTURE2D_HEIGHT), 0 }	{ (1,SURFACE2D_WIDTH), (1,SURFACE2D_HEIGHT), 0 }
3D	{ (1,TEXTURE3D_WIDTH), (1,TEXTURE3D_HEIGHT), (1,TEXTURE3D_DEPTH) } OR { (1,TEXTURE3D_WIDTH_ALTERNATE), (1,TEXTURE3D_HEIGHT_ALTERNATE), (1,TEXTURE3D_DEPTH_ALTERNATE) }	{ (1,SURFACE3D_WIDTH), (1,SURFACE3D_HEIGHT), (1,SURFACE3D_DEPTH) }
1D Layered	{ (1,TEXTURE1D_LAYERED_WIDTH), 0, (1,TEXTURE1D_LAYERED_LAYERS) }	{ (1,SURFACE1D_LAYERED_WIDTH), 0, (1,SURFACE1D_LAYERED_LAYERS) }
2D Layered	{ (1,TEXTURE2D_LAYERED_WIDTH), (1,TEXTURE2D_LAYERED_HEIGHT), (1,TEXTURE2D_LAYERED_LAYERS) }	{ (1,SURFACE2D_LAYERED_WIDTH), (1,SURFACE2D_LAYERED_HEIGHT), (1,SURFACE2D_LAYERED_LAYERS) }
Cubemap	{ (1,TEXTURECUBEMAP_WIDTH), (1,TEXTURECUBEMAP_WIDTH), 6 }	{ (1,SURFACECUBEMAP_WIDTH), (1,SURFACECUBEMAP_WIDTH), 6 }
Cubemap Layered	{ (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_LAYERS) }	{ (1,SURFACECUBEMAP_LAYERED_WIDTH), (1,SURFACECUBEMAP_LAYERED_WIDTH), (1,SURFACECUBEMAP_LAYERED_LAYERS) }

Here are examples of CUDA array descriptions:

Description for a CUDA array of 2048 floats:

```
CUDA_ARRAY3D_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 2048;
desc.Height = 0;
desc.Depth = 0;
```

Description for a 64 x 64 CUDA array of floats:

```
CUDA_ARRAY3D_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 64;
desc.Height = 64;
desc.Depth = 0;
```

Description for a `width` x `height` x `depth` CUDA array of 64-bit, 4x16-bit float16's:

```
CUDA_ARRAY3D_DESCRIPTOR desc;
desc.FormatFlags = CU_AD_FORMAT_HALF;
desc.NumChannels = 4;
desc.Width = width;
desc.Height = height;
desc.Depth = depth;
```

Parameters:

pHandle - Returned array

pAllocateArray - 3D array descriptor

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.2 CUresult cuArray3DGetDescriptor (CUDA_ARRAY3D_DESCRIPTOR * *pArrayDescriptor*, CUarray *hArray*)

Returns in `*pArrayDescriptor` a descriptor containing information on the format and dimensions of the CUDA array `hArray`. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

This function may be called on 1D and 2D arrays, in which case the `Height` and/or `Depth` members of the descriptor struct will be set to 0.

Parameters:

pArrayDescriptor - Returned 3D array descriptor

hArray - 3D array to get descriptor of

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.3 CUresult cuArrayCreate (CUarray * *pHandle*, const CUDA_ARRAY_DESCRIPTOR * *pAllocateArray*)

Creates a CUDA array according to the CUDA_ARRAY_DESCRIPTOR structure `pAllocateArray` and returns a handle to the new CUDA array in `*pHandle`. The CUDA_ARRAY_DESCRIPTOR is defined as:

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    CUarray_format Format;
    unsigned int NumChannels;
} CUDA_ARRAY_DESCRIPTOR;
```

where:

- `Width`, and `Height` are the width, and height of the CUDA array (in elements); the CUDA array is one-dimensional if height is 0, two-dimensional otherwise;
- Format specifies the format of the elements; CUarray_format is defined as:

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20
} CUarray_format;
```

- `NumChannels` specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;

Here are examples of CUDA array descriptions:

Description for a CUDA array of 2048 floats:

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 2048;
desc.Height = 1;
```

Description for a 64 x 64 CUDA array of floats:

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 64;
desc.Height = 64;
```

Description for a `width` x `height` CUDA array of 64-bit, 4x16-bit float16's:

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.FormatFlags = CU_AD_FORMAT_HALF;
desc.NumChannels = 4;
desc.Width = width;
desc.Height = height;
```

Description for a `width` x `height` CUDA array of 16-bit elements, each of which is two 8-bit unsigned chars:

```
CUDA_ARRAY_DESCRIPTOR arrayDesc;
desc.FormatFlags = CU_AD_FORMAT_UNSIGNED_INT8;
desc.NumChannels = 2;
desc.Width = width;
desc.Height = height;
```

Parameters:

pHandle - Returned array

pAllocateArray - Array descriptor

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.4 CUresult cuArrayDestroy (CUarray *hArray*)

Destroys the CUDA array `hArray`.

Parameters:

hArray - Array to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ARRAY_IS_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.5 CUresult cuArrayGetDescriptor (CUDA_ARRAY_DESCRIPTOR * *pArrayDescriptor*, CUarray *hArray*)

Returns in `*pArrayDescriptor` a descriptor containing information on the format and dimensions of the CUDA array `hArray`. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

Parameters:

pArrayDescriptor - Returned array descriptor

hArray - Array to get descriptor of

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.6 CUresult cuDeviceGetByPCIBusId (CUdevice * *dev*, char * *pciBusId*)

Returns in `*device` a device handle given a PCI bus ID string.

Parameters:

dev - Returned device handle

pciBusId - String in one of the following forms: [domain]:[bus]:[device].[function] [domain]:[bus]:[device] [bus]:[device].[function] where `domain`, `bus`, `device`, and `function` are all hexadecimal values

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuDeviceGet, cuDeviceGetAttribute, cuDeviceGetPCIBusId

5.37.2.7 CUresult cuDeviceGetPCIBusId (char * *pciBusId*, int *len*, CUdevice *dev*)

Returns an ASCII string identifying the device `dev` in the NULL-terminated string pointed to by `pciBusId`. `len` specifies the maximum length of the string that may be returned.

Parameters:

pciBusId - Returned identifier string for the device in the following format [domain]:[bus]:[device].[function] where `domain`, `bus`, `device`, and `function` are all hexadecimal values. pciBusId should be large enough to store 13 characters including the NULL-terminator.

len - Maximum length of string to store in `name`

dev - Device to get identifier string for

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuDeviceGet, cuDeviceGetAttribute, cuDeviceGetByPCIBusId

5.37.2.8 CUresult cuIpcCloseMemHandle (CUdeviceptr *dptr*)

Unmaps memory returnd by cuIpcOpenMemHandle. The original allocation in the exporting process as well as imported mappings in other processes will be unaffected.

Any resources used to enable peer access will be freed if this is the last mapping using them.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

Parameters:

dptr - Device pointer returned by cuIpcOpenMemHandle

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_MAP_FAILED, CUDA_-ERROR_INVALID_HANDLE,

See also:

cuMemAlloc, cuMemFree, cuIpcGetEventHandle, cuIpcOpenEventHandle, cuIpcGetMemHandle, cuIpcOpen-MemHandle,

5.37.2.9 CUresult cuIpcGetEventHandle (CUipcEventHandle * *pHandle*, CUevent *event*)

Takes as input a previously allocated event. This event must have been created with the CU_EVENT_-INTERPROCESS and CU_EVENT_DISABLE_TIMING flags set. This opaque handle may be copied into other processes and opened with cuIpcOpenEventHandle to allow efficient hardware synchronization between GPU work in different processes.

After the event has been been opened in the importing process, cuEventRecord, cuEventSynchronize, cuStreamWaitEvent and cuEventQuery may be used in either process. Performing operations on the imported event after the exported event has been freed with cuEventDestroy will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

Parameters:

pHandle - Pointer to a user allocated CUipcEventHandle in which to return the opaque event handle

event - Event allocated with CU_EVENT_INTERPROCESS and CU_EVENT_DISABLE_TIMING flags.

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_MAP_FAILED

See also:

cuEventCreate, cuEventDestroy, cuEventSynchronize, cuEventQuery, cuStreamWaitEvent, cuIpcOpenEventHandle, cuIpcGetMemHandle, cuIpcOpenMemHandle, cuIpcCloseMemHandle

5.37.2.10 CUresult cuIpcGetMemHandle (CUipcMemHandle * *pHandle*, CUdeviceptr *dptr*)

/brief Gets an interprocess memory handle for an existing device memory allocation

Takes a pointer to the base of an existing device memory allocation created with cuMemAlloc and exports it for use in another process. This is a lightweight operation and may be called multiple times on an allocation without adverse effects.

If a region of memory is freed with cuMemFree and a subsequent call to cuMemAlloc returns memory with the same device address, cuIpcGetMemHandle will return a unique handle for the new memory.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

Parameters:

pHandle - Pointer to user allocated CUipcMemHandle to return the handle in.

dptr - Base pointer to previously allocated device memory

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_MAP_FAILED,

See also:

cuMemAlloc, cuMemFree, cuIpcGetEventHandle, cuIpcOpenEventHandle, cuIpcOpenMemHandle, cuIpcCloseMemHandle

5.37.2.11 CUresult cuIpcOpenEventHandle (CUevent * *phEvent*, CUipcEventHandle *handle*)

Opens an interprocess event handle exported from another process with cuIpcGetEventHandle. This function returns a CUevent that behaves like a locally created event with the CU_EVENT_DISABLE_TIMING flag specified. This event must be freed with cuEventDestroy.

Performing operations on the imported event after the exported event has been freed with cuEventDestroy will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

Parameters:

phEvent - Returned the imported event

handle - Interprocess handle to open

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_MAP_FAILED, CUDA_ERROR_PEER_ACCESS_UNSUPPORTED, CUDA_ERROR_INVALID_HANDLE

See also:

cuEventCreate, cuEventDestroy, cuEventSynchronize, cuEventQuery, cuStreamWaitEvent, cuIpcGetEventHandle, cuIpcGetMemHandle, cuIpcOpenMemHandle, cuIpcCloseMemHandle

5.37.2.12 CUresult cuIpcOpenMemHandle (CUdeviceptr * *pdptr*, CUipcMemHandle *handle*, unsigned int *Flags*)

/brief Opens an interprocess memory handle exported from another process and returns a device pointer usable in the local process.

Maps memory exported from another process with cuIpcGetMemHandle into the current device address space. For contexts on different devices cuIpcOpenMemHandle can attempt to enable peer access between the devices as if the user called cuCtxEnablePeerAccess. This behavior is controlled by the CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS flag. cuDeviceCanAccessPeer can determine if a mapping is possible.

Contexts that may open CUipcMemHandles are restricted in the following way. CUipcMemHandles from each CUdevice in a given process may only be opened by one CUcontext per CUdevice per other process.

Memory returned from cuIpcOpenMemHandle must be freed with cuIpcCloseMemHandle.

Calling cuMemFree on an exported memory region before calling cuIpcCloseMemHandle in the importing context will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

Parameters:

pdptr - Returned device pointer

handle - CUipcMemHandle to open

Flags - Flags for this operation. Must be specified as CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_MAP_FAILED, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_TOO_MANY_PEERS

See also:

cuMemAlloc, cuMemFree, cuIpcGetEventHandle, cuIpcOpenEventHandle, cuIpcGetMemHandle, cuIpcCloseMemHandle, cuCtxEnablePeerAccess, cuDeviceCanAccessPeer,

5.37.2.13 CUresult cuMemAlloc (CUdeviceptr * *dptr*, size_t *bytesize*)

Allocates `bytesize` bytes of linear memory on the device and returns in `*dptr` a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If `bytesize` is 0, cuMemAlloc() returns CUDA_ERROR_INVALID_VALUE.

Parameters:

dptr - Returned device pointer

bytesize - Requested allocation size in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAl-locHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpy-DtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGe-tAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMem-setD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.14 CUresult cuMemAllocHost (void ** *pp*, size_t *bytesize*)

Allocates `bytesize` bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as cuMemcpy(). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc(). Allocating excessive amounts of memory with cuMemAl-locHost() may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

Note all host memory allocated using cuMemHostAlloc() will automatically be immediately accessible to all contexts on all devices which support unified addressing (as may be queried using CU_DEVICE_ATTRIBUTE_UNIFIED_-ADDRESSING). The device pointer that may be used to access this host memory from those contexts is always equal to the returned host pointer `*pp`. See Unified Addressing for additional details.

Parameters:

pp - Returned host pointer to page-locked memory

bytesize - Requested allocation size in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAl-loc, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMem-cpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA,

cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.15 CUresult cuMemAllocPitch (CUdeviceptr * *dptr*, size_t * *pPitch*, size_t *WidthInBytes*, size_t *Height*, unsigned int *ElementSizeBytes*)

Allocates at least `WidthInBytes * Height` bytes of linear memory on the device and returns in `*dptr` a pointer to the allocated memory. The function may pad the allocation to ensure that corresponding pointers in any given row will continue to meet the alignment requirements for coalescing as the address is updated from row to row. `ElementSizeBytes` specifies the size of the largest reads and writes that will be performed on the memory range. `ElementSizeBytes` may be 4, 8 or 16 (since coalesced memory transactions are not possible on other data sizes). If `ElementSizeBytes` is smaller than the actual read/write size of a kernel, the kernel will run correctly, but possibly at reduced speed. The pitch returned in `*pPitch` by cuMemAllocPitch() is the width in bytes of the allocation. The intended usage of pitch is as a separate parameter of the allocation, used to compute addresses within the 2D array. Given the row and column of an array element of type **T**, the address is computed as:

```
T* pElement = (T*)((char*)BaseAddress + Row * Pitch) + Column;
```

The pitch returned by cuMemAllocPitch() is guaranteed to work with cuMemcpy2D() under all circumstances. For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using cuMemAllocPitch(). Due to alignment restrictions in the hardware, this is especially true if the application will be performing 2D memory copies between different regions of device memory (whether linear memory or CUDA arrays).

The byte alignment of the pitch returned by cuMemAllocPitch() is guaranteed to match or exceed the alignment requirement for texture binding with cuTexRefSetAddress2D().

Parameters:

dptr - Returned device pointer

pPitch - Returned pitch of allocation in bytes

WidthInBytes - Requested allocation width in bytes

Height - Requested allocation height in rows

ElementSizeBytes - Size of largest reads/writes for range

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.16 CUresult cuMemcpy (CUdeviceptr *dst*, CUdeviceptr *src*, size_t *ByteCount*)

Copies data between two pointers. `dst` and `src` are base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy. Note that this function infers the type of the transfer (host to host, host to device, device to device, or device to host) from the pointer values. This function is only allowed in contexts which support unified addressing. Note that this function is synchronous.

Parameters:

dst - Destination unified virtual address space pointer

src - Source unified virtual address space pointer

ByteCount - Size of memory copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAl-loc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAto-HAsync, cuMemcpyDtoA, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGet-Info, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.17 CUresult cuMemcpy2D (const CUDA_MEMCPY2D * *pCopy*)

Perform a 2D memory copy according to the parameters specified in `pCopy`. The CUDA_MEMCPY2D structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
   unsigned int srcXInBytes, srcY;
   CUmemorytype srcMemoryType;
       const void *srcHost;
       CUdeviceptr srcDevice;
       CUarray srcArray;
       unsigned int srcPitch;

   unsigned int dstXInBytes, dstY;
   CUmemorytype dstMemoryType;
       void *dstHost;
       CUdeviceptr dstDevice;
       CUarray dstArray;
       unsigned int dstPitch;

   unsigned int WidthInBytes;
   unsigned int Height;
} CUDA_MEMCPY2D;
```

where:

- srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
typedef enum CUmemorytype_enum {
   CU_MEMORYTYPE_HOST = 0x01,
   CU_MEMORYTYPE_DEVICE = 0x02,
   CU_MEMORYTYPE_ARRAY = 0x03,
   CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

If srcMemoryType is CU_MEMORYTYPE_UNIFIED, srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is CU_MEMORYTYPE_HOST, srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_DEVICE, srcDevice and srcPitch specify the (device) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_ARRAY, srcArray specifies the handle of the source data. srcHost, srcDevice and srcPitch are ignored.

If dstMemoryType is CU_MEMORYTYPE_HOST, dstHost and dstPitch specify the (host) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_UNIFIED, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is CU_MEMORYTYPE_DEVICE, dstDevice and dstPitch specify the (device) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_ARRAY, dstArray specifies the handle of the destination data. dstHost, dstDevice and dstPitch are ignored.

- srcXInBytes and srcY specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes and dstY specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

- WidthInBytes and Height specify the width (in bytes) and height of the 2D copy being performed.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.

cuMemcpy2D() returns an error if any pitch is greater than the maximum allowed (CU_DEVICE_ATTRIBUTE_MAX_PITCH). cuMemAllocPitch() passes back pitches that always work with cuMemcpy2D(). On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), cuMemcpy2D() may fail for pitches not computed by cuMemAllocPitch(). cuMemcpy2DUnaligned() does not have this restriction, but may run significantly slower in the cases where cuMemcpy2D() would have returned an error code.

Parameters:

pCopy - Parameters for the memory copy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.18 CUresult cuMemcpy2DAsync (const CUDA_MEMCPY2D * *pCopy*, CUstream *hStream*)

Perform a 2D memory copy according to the parameters specified in `pCopy`. The CUDA_MEMCPY2D structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
   unsigned int srcXInBytes, srcY;
   CUmemorytype srcMemoryType;
   const void *srcHost;
   CUdeviceptr srcDevice;
   CUarray srcArray;
   unsigned int srcPitch;
   unsigned int dstXInBytes, dstY;
   CUmemorytype dstMemoryType;
   void *dstHost;
   CUdeviceptr dstDevice;
   CUarray dstArray;
   unsigned int dstPitch;
   unsigned int WidthInBytes;
   unsigned int Height;
} CUDA_MEMCPY2D;
```

where:

- srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
typedef enum CUmemorytype_enum {
   CU_MEMORYTYPE_HOST = 0x01,
   CU_MEMORYTYPE_DEVICE = 0x02,
   CU_MEMORYTYPE_ARRAY = 0x03,
   CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

  If srcMemoryType is CU_MEMORYTYPE_HOST, srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

  If srcMemoryType is CU_MEMORYTYPE_UNIFIED, srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

  If srcMemoryType is CU_MEMORYTYPE_DEVICE, srcDevice and srcPitch specify the (device) base address of the source data and the bytes per row to apply. srcArray is ignored.

  If srcMemoryType is CU_MEMORYTYPE_ARRAY, srcArray specifies the handle of the source data. srcHost, srcDevice and srcPitch are ignored.

  If dstMemoryType is CU_MEMORYTYPE_UNIFIED, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

  If dstMemoryType is CU_MEMORYTYPE_HOST, dstHost and dstPitch specify the (host) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_DEVICE, dstDevice and dstPitch specify the (device) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_ARRAY, dstArray specifies the handle of the destination data. dstHost, dstDevice and dstPitch are ignored.

- srcXInBytes and srcY specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes and dstY specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

- WidthInBytes and Height specify the width (in bytes) and height of the 2D copy being performed.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
- If specified, srcHeight must be greater than or equal to Height + srcY, and dstHeight must be greater than or equal to Height + dstY.

cuMemcpy2D() returns an error if any pitch is greater than the maximum allowed (CU_DEVICE_ATTRIBUTE_-MAX_PITCH). cuMemAllocPitch() passes back pitches that always work with cuMemcpy2D(). On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), cuMemcpy2D() may fail for pitches not computed by cuMemAllocPitch(). cuMemcpy2DUnaligned() does not have this restriction, but may run significantly slower in the cases where cuMemcpy2D() would have returned an error code.

cuMemcpy2DAsync() is asynchronous and can optionally be associated to a stream by passing a non-zero `hStream` argument. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input.

Parameters:

pCopy - Parameters for the memory copy

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.19 CUresult cuMemcpy2DUnaligned (const CUDA_MEMCPY2D * *pCopy*)

Perform a 2D memory copy according to the parameters specified in `pCopy`. The CUDA_MEMCPY2D structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
   unsigned int srcXInBytes, srcY;
   CUmemorytype srcMemoryType;
   const void *srcHost;
   CUdeviceptr srcDevice;
   CUarray srcArray;
   unsigned int srcPitch;
   unsigned int dstXInBytes, dstY;
   CUmemorytype dstMemoryType;
   void *dstHost;
   CUdeviceptr dstDevice;
   CUarray dstArray;
   unsigned int dstPitch;
   unsigned int WidthInBytes;
   unsigned int Height;
} CUDA_MEMCPY2D;
```

where:

- srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
typedef enum CUmemorytype_enum {
   CU_MEMORYTYPE_HOST = 0x01,
   CU_MEMORYTYPE_DEVICE = 0x02,
   CU_MEMORYTYPE_ARRAY = 0x03,
   CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

If srcMemoryType is CU_MEMORYTYPE_UNIFIED, srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is CU_MEMORYTYPE_HOST, srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_DEVICE, srcDevice and srcPitch specify the (device) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_ARRAY, srcArray specifies the handle of the source data. srcHost, srcDevice and srcPitch are ignored.

If dstMemoryType is CU_MEMORYTYPE_UNIFIED, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is CU_MEMORYTYPE_HOST, dstHost and dstPitch specify the (host) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_DEVICE, dstDevice and dstPitch specify the (device) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_ARRAY, dstArray specifies the handle of the destination data. dstHost, dstDevice and dstPitch are ignored.

- srcXInBytes and srcY specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes and dstY specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

- WidthInBytes and Height specify the width (in bytes) and height of the 2D copy being performed.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.

cuMemcpy2D() returns an error if any pitch is greater than the maximum allowed (CU_DEVICE_ATTRIBUTE_MAX_PITCH). cuMemAllocPitch() passes back pitches that always work with cuMemcpy2D(). On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), cuMemcpy2D() may fail for pitches not computed by cuMemAllocPitch(). cuMemcpy2DUnaligned() does not have this restriction, but may run significantly slower in the cases where cuMemcpy2D() would have returned an error code.

Parameters:

pCopy - Parameters for the memory copy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.20 CUresult cuMemcpy3D (const CUDA_MEMCPY3D * *pCopy*)

Perform a 3D memory copy according to the parameters specified in `pCopy`. The CUDA_MEMCPY3D structure is defined as:

```
typedef struct CUDA_MEMCPY3D_st {

    unsigned int srcXInBytes, srcY, srcZ;
    unsigned int srcLOD;
    CUmemorytype srcMemoryType;
        const void *srcHost;
        CUdeviceptr srcDevice;
        CUarray srcArray;
        unsigned int srcPitch;  // ignored when src is array
        unsigned int srcHeight; // ignored when src is array; may be 0 if Depth==1

    unsigned int dstXInBytes, dstY, dstZ;
    unsigned int dstLOD;
    CUmemorytype dstMemoryType;
        void *dstHost;
        CUdeviceptr dstDevice;
        CUarray dstArray;
        unsigned int dstPitch;  // ignored when dst is array
        unsigned int dstHeight; // ignored when dst is array; may be 0 if Depth==1

    unsigned int WidthInBytes;
    unsigned int Height;
    unsigned int Depth;
} CUDA_MEMCPY3D;
```

where:

- srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
typedef enum CUmemorytype_enum {
   CU_MEMORYTYPE_HOST = 0x01,
   CU_MEMORYTYPE_DEVICE = 0x02,
   CU_MEMORYTYPE_ARRAY = 0x03,
   CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

If srcMemoryType is CU_MEMORYTYPE_UNIFIED, srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is CU_MEMORYTYPE_HOST, srcHost, srcPitch and srcHeight specify the (host) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_DEVICE, srcDevice, srcPitch and srcHeight specify the (device) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_ARRAY, srcArray specifies the handle of the source data. srcHost, srcDevice, srcPitch and srcHeight are ignored.

If dstMemoryType is CU_MEMORYTYPE_UNIFIED, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is CU_MEMORYTYPE_HOST, dstHost and dstPitch specify the (host) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_DEVICE, dstDevice and dstPitch specify the (device) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_ARRAY, dstArray specifies the handle of the destination data. dstHost, dstDevice, dstPitch and dstHeight are ignored.

- srcXInBytes, srcY and srcZ specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+(srcZ*srcHeight+srcY)*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+(srcZ*srcHeight+srcY)*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes, dstY and dstZ specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+(dstZ*dstHeight+dstY)*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+(dstZ*dstHeight+dstY)*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

- WidthInBytes, Height and Depth specify the width (in bytes), height and depth of the 3D copy being performed.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
- If specified, srcHeight must be greater than or equal to Height + srcY, and dstHeight must be greater than or equal to Height + dstY.

cuMemcpy3D() returns an error if any pitch is greater than the maximum allowed (CU_DEVICE_ATTRIBUTE_MAX_PITCH).

The srcLOD and dstLOD members of the CUDA_MEMCPY3D structure must be set to 0.

Parameters:

pCopy - Parameters for the memory copy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.21 CUresult cuMemcpy3DAsync (const CUDA_MEMCPY3D * *pCopy*, CUstream *hStream*)

Perform a 3D memory copy according to the parameters specified in `pCopy`. The CUDA_MEMCPY3D structure is defined as:

```
typedef struct CUDA_MEMCPY3D_st {

    unsigned int srcXInBytes, srcY, srcZ;
    unsigned int srcLOD;
    CUmemorytype srcMemoryType;
        const void *srcHost;
        CUdeviceptr srcDevice;
        CUarray srcArray;
        unsigned int srcPitch;  // ignored when src is array
        unsigned int srcHeight; // ignored when src is array; may be 0 if Depth==1

    unsigned int dstXInBytes, dstY, dstZ;
    unsigned int dstLOD;
    CUmemorytype dstMemoryType;
        void *dstHost;
        CUdeviceptr dstDevice;
        CUarray dstArray;
        unsigned int dstPitch;  // ignored when dst is array
        unsigned int dstHeight; // ignored when dst is array; may be 0 if Depth==1

    unsigned int WidthInBytes;
    unsigned int Height;
    unsigned int Depth;
} CUDA_MEMCPY3D;
```

where:

- srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
typedef enum CUmemorytype_enum {
```

```
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
    CU_MEMORYTYPE_ARRAY = 0x03,
    CU_MEMORYTYPE_UNIFIED = 0x04
} CUmemorytype;
```

If srcMemoryType is CU_MEMORYTYPE_UNIFIED, srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is CU_MEMORYTYPE_HOST, srcHost, srcPitch and srcHeight specify the (host) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_DEVICE, srcDevice, srcPitch and srcHeight specify the (device) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_ARRAY, srcArray specifies the handle of the source data. srcHost, srcDevice, srcPitch and srcHeight are ignored.

If dstMemoryType is CU_MEMORYTYPE_UNIFIED, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is CU_MEMORYTYPE_HOST, dstHost and dstPitch specify the (host) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_DEVICE, dstDevice and dstPitch specify the (device) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_ARRAY, dstArray specifies the handle of the destination data. dstHost, dstDevice, dstPitch and dstHeight are ignored.

- srcXInBytes, srcY and srcZ specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+(srcZ*srcHeight+srcY)*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+(srcZ*srcHeight+srcY)*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

- dstXInBytes, dstY and dstZ specify the base address of the destination data for the copy.

  For host pointers, the base address is

  ```
  void* dstStart = (void*)((char*)dstHost+(dstZ*dstHeight+dstY)*dstPitch + dstXInBytes);
  ```

  For device pointers, the starting address is

  ```
  CUdeviceptr dstStart = dstDevice+(dstZ*dstHeight+dstY)*dstPitch+dstXInBytes;
  ```

  For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

- WidthInBytes, Height and Depth specify the width (in bytes), height and depth of the 3D copy being performed.
- If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
- If specified, srcHeight must be greater than or equal to Height + srcY, and dstHeight must be greater than or equal to Height + dstY.

  cuMemcpy3D() returns an error if any pitch is greater than the maximum allowed (CU_DEVICE_ATTRIBUTE_MAX_PITCH).

cuMemcpy3DAsync() is asynchronous and can optionally be associated to a stream by passing a non-zero `hStream` argument. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input.

The srcLOD and dstLOD members of the CUDA_MEMCPY3D structure must be set to 0.

Parameters:

pCopy - Parameters for the memory copy

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.22 CUresult cuMemcpy3DPeer (const CUDA_MEMCPY3D_PEER * *pCopy*)

Perform a 3D memory copy according to the parameters specified in `pCopy`. See the definition of the CUDA_MEMCPY3D_PEER structure for documentation of its parameters.

Note that this function is synchronous with respect to the host only if the source or destination memory is of type CU_MEMORYTYPE_HOST. Note also that this copy is serialized with respect all pending and future asynchronous work in to the current context, the copy's source context, and the copy's destination context (use cuMemcpy3DPeerAsync to avoid this synchronization).

Parameters:

pCopy - Parameters for the memory copy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemcpyDtoD, cuMemcpyPeer, cuMemcpyDtoDAsync, cuMemcpyPeerAsync, cuMemcpy3DPeerAsync

5.37.2.23 CUresult cuMemcpy3DPeerAsync (const CUDA_MEMCPY3D_PEER * *pCopy*, CUstream *hStream*)

Perform a 3D memory copy according to the parameters specified in `pCopy`. See the definition of the CUDA_MEMCPY3D_PEER structure for documentation of its parameters.

Parameters:

pCopy - Parameters for the memory copy

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemcpyDtoD, cuMemcpyPeer, cuMemcpyDtoDAsync, cuMemcpyPeerAsync, cuMemcpy3DPeerAsync

5.37.2.24 CUresult cuMemcpyAsync (CUdeviceptr *dst*, CUdeviceptr *src*, size_t *ByteCount*, CUstream *hStream*)

Copies data between two pointers. `dst` and `src` are base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy. Note that this function infers the type of the transfer (host to

host, host to device, device to device, or device to host) from the pointer values. This function is only allowed in contexts which support unified addressing. Note that this function is asynchronous and can optionally be associated to a stream by passing a non-zero `hStream` argument

Parameters:

dst - Destination unified virtual address space pointer

src - Source unified virtual address space pointer

ByteCount - Size of memory copy in bytes

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.25 CUresult cuMemcpyAtoA (CUarray *dstArray*, size_t *dstOffset*, CUarray *srcArray*, size_t *srcOffset*, size_t *ByteCount*)

Copies from one 1D CUDA array to another. `dstArray` and `srcArray` specify the handles of the destination and source CUDA arrays for the copy, respectively. `dstOffset` and `srcOffset` specify the destination and source offsets in bytes into the CUDA arrays. `ByteCount` is the number of bytes to be copied. The size of the elements in the CUDA arrays need not be the same format, but the elements must be the same size; and count must be evenly divisible by that size.

Parameters:

dstArray - Destination array

dstOffset - Offset in bytes of destination array

srcArray - Source array

srcOffset - Offset in bytes of source array

ByteCount - Size of memory copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.26 CUresult cuMemcpyAtoD (CUdeviceptr *dstDevice*, CUarray *srcArray*, size_t *srcOffset*, size_t *ByteCount*)

Copies from one 1D CUDA array to device memory. `dstDevice` specifies the base pointer of the destination and must be naturally aligned with the CUDA array elements. `srcArray` and `srcOffset` specify the CUDA array handle and the offset in bytes into the array where the copy is to begin. `ByteCount` specifies the number of bytes to copy and must be evenly divisible by the array element size.

Parameters:

dstDevice - Destination device pointer

srcArray - Source array

srcOffset - Offset in bytes of source array

ByteCount - Size of memory copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.27 CUresult cuMemcpyAtoH (void * *dstHost*, CUarray *srcArray*, size_t *srcOffset*, size_t *ByteCount*)

Copies from one 1D CUDA array to host memory. `dstHost` specifies the base pointer of the destination. `srcArray` and `srcOffset` specify the CUDA array handle and starting offset in bytes of the source data. `ByteCount` specifies the number of bytes to copy.

Parameters:

dstHost - Destination device pointer

srcArray - Source array

srcOffset - Offset in bytes of source array

ByteCount - Size of memory copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.28 CUresult cuMemcpyAtoHAsync (void * *dstHost*, CUarray *srcArray*, size_t *srcOffset*, size_t *ByteCount*, CUstream *hStream*)

Copies from one 1D CUDA array to host memory. `dstHost` specifies the base pointer of the destination. `srcArray` and `srcOffset` specify the CUDA array handle and starting offset in bytes of the source data. `ByteCount` specifies the number of bytes to copy.

cuMemcpyAtoHAsync() is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input.

Parameters:

dstHost - Destination pointer

srcArray - Source array

srcOffset - Offset in bytes of source array

ByteCount - Size of memory copy in bytes

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.29 CUresult cuMemcpyDtoA (CUarray *dstArray*, size_t *dstOffset*, CUdeviceptr *srcDevice*, size_t *ByteCount*)

Copies from device memory to a 1D CUDA array. `dstArray` and `dstOffset` specify the CUDA array handle and starting index of the destination data. `srcDevice` specifies the base pointer of the source. `ByteCount` specifies the number of bytes to copy.

Parameters:

dstArray - Destination array

dstOffset - Offset in bytes of destination array

srcDevice - Source device pointer

ByteCount - Size of memory copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.30 CUresult cuMemcpyDtoD (CUdeviceptr *dstDevice*, CUdeviceptr *srcDevice*, size_t *ByteCount*)

Copies from device memory to device memory. `dstDevice` and `srcDevice` are the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy. Note that this function is asynchronous.

Parameters:

dstDevice - Destination device pointer

srcDevice - Source device pointer

ByteCount - Size of memory copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.31 CUresult cuMemcpyDtoDAsync (CUdeviceptr *dstDevice*, CUdeviceptr *srcDevice*, size_t *ByteCount*, CUstream *hStream*)

Copies from device memory to device memory. `dstDevice` and `srcDevice` are the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy. Note that this function is asynchronous and can optionally be associated to a stream by passing a non-zero `hStream` argument

Parameters:

dstDevice - Destination device pointer

srcDevice - Source device pointer

ByteCount - Size of memory copy in bytes

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.32 CUresult cuMemcpyDtoH (void * *dstHost*, CUdeviceptr *srcDevice*, size_t *ByteCount*)

Copies from device to host memory. `dstHost` and `srcDevice` specify the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy. Note that this function is synchronous.

Parameters:

dstHost - Destination host pointer

srcDevice - Source device pointer

ByteCount - Size of memory copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.33 CUresult cuMemcpyDtoHAsync (void * *dstHost*, CUdeviceptr *srcDevice*, size_t *ByteCount*, CUstream *hStream*)

Copies from device to host memory. `dstHost` and `srcDevice` specify the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.

cuMemcpyDtoHAsync() is asynchronous and can optionally be associated to a stream by passing a non-zero `hStream` argument. It only works on page-locked memory and returns an error if a pointer to pageable memory is passed as input.

Parameters:

dstHost - Destination host pointer

srcDevice - Source device pointer

ByteCount - Size of memory copy in bytes

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.34 CUresult cuMemcpyHtoA (CUarray *dstArray*, size_t *dstOffset*, const void * *srcHost*, size_t *ByteCount*)

Copies from host memory to a 1D CUDA array. `dstArray` and `dstOffset` specify the CUDA array handle and starting offset in bytes of the destination data. `pSrc` specifies the base address of the source. `ByteCount` specifies the number of bytes to copy.

Parameters:

dstArray - Destination array

dstOffset - Offset in bytes of destination array

srcHost - Source host pointer

ByteCount - Size of memory copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.35 CUresult cuMemcpyHtoAAsync (CUarray *dstArray*, size_t *dstOffset*, const void * *srcHost*, size_t *ByteCount*, CUstream *hStream*)

Copies from host memory to a 1D CUDA array. `dstArray` and `dstOffset` specify the CUDA array handle and starting offset in bytes of the destination data. `srcHost` specifies the base address of the source. `ByteCount` specifies the number of bytes to copy.

cuMemcpyHtoAAsync() is asynchronous and can optionally be associated to a stream by passing a non-zero `hStream` argument. It only works on page-locked memory and returns an error if a pointer to pageable memory is passed as input.

Parameters:

dstArray - Destination array

dstOffset - Offset in bytes of destination array

srcHost - Source host pointer

ByteCount - Size of memory copy in bytes

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.36 CUresult cuMemcpyHtoD (CUdeviceptr *dstDevice*, const void * *srcHost*, size_t *ByteCount*)

Copies from host memory to device memory. `dstDevice` and `srcHost` are the base addresses of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy. Note that this function is synchronous.

Parameters:

dstDevice - Destination device pointer

srcHost - Source host pointer

ByteCount - Size of memory copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.37 CUresult cuMemcpyHtoDAsync (CUdeviceptr *dstDevice*, const void * *srcHost*, size_t *ByteCount*, CUstream *hStream*)

Copies from host memory to device memory. `dstDevice` and `srcHost` are the base addresses of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.

cuMemcpyHtoDAsync() is asynchronous and can optionally be associated to a stream by passing a non-zero `hStream` argument. It only works on page-locked memory and returns an error if a pointer to pageable memory is passed as input.

Parameters:

dstDevice - Destination device pointer

srcHost - Source host pointer

ByteCount - Size of memory copy in bytes

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.38 CUresult cuMemcpyPeer (CUdeviceptr *dstDevice*, CUcontext *dstContext*, CUdeviceptr *srcDevice*, CUcontext *srcContext*, size_t *ByteCount*)

Copies from device memory in one context to device memory in another context. `dstDevice` is the base device pointer of the destination memory and `dstContext` is the destination context. `srcDevice` is the base device pointer of the source memory and `srcContext` is the source pointer. `ByteCount` specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host, but serialized with respect all pending and future asynchronous work in to the current context, `srcContext`, and `dstContext` (use cuMemcpyPeerAsync to avoid this synchronization).

Parameters:

dstDevice - Destination device pointer

dstContext - Destination context

srcDevice - Source device pointer

srcContext - Source context

ByteCount - Size of memory copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemcpyDtoD, cuMemcpy3DPeer, cuMemcpyDtoDAsync, cuMemcpyPeerAsync, cuMemcpy3DPeerAsync

5.37.2.39 CUresult cuMemcpyPeerAsync (CUdeviceptr *dstDevice*, CUcontext *dstContext*, CUdeviceptr *srcDevice*, CUcontext *srcContext*, size_t *ByteCount*, CUstream *hStream*)

Copies from device memory in one context to device memory in another context. `dstDevice` is the base device pointer of the destination memory and `dstContext` is the destination context. `srcDevice` is the base device pointer of the source memory and `srcContext` is the source pointer. `ByteCount` specifies the number of bytes to copy. Note that this function is asynchronous with respect to the host and all work in other streams in other devices.

Parameters:

dstDevice - Destination device pointer

dstContext - Destination context

srcDevice - Source device pointer

srcContext - Source context

ByteCount - Size of memory copy in bytes

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemcpyDtoD, cuMemcpyPeer, cuMemcpy3DPeer, cuMemcpyDtoDAsync, cuMemcpy3DPeerAsync

5.37.2.40 CUresult cuMemFree (CUdeviceptr *dptr*)

Frees the memory space pointed to by `dptr`, which must have been returned by a previous call to cuMemAlloc() or cuMemAllocPitch().

Parameters:

dptr - Pointer to memory to free

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAl-loc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyA-toHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDto-HAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.41 CUresult cuMemFreeHost (void * *p*)

Frees the memory space pointed to by `p`, which must have been returned by a previous call to cuMemAllocHost().

Parameters:

p - Pointer to memory to free

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAl-loc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAto-HAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemGe-tAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMem-setD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.42 CUresult cuMemGetAddressRange (CUdeviceptr * *pbase*, size_t * *psize*, CUdeviceptr *dptr*)

Returns the base address in `*pbase` and size in `*psize` of the allocation by cuMemAlloc() or cuMemAllocPitch() that contains the input pointer `dptr`. Both parameters `pbase` and `psize` are optional. If one of them is NULL, it is ignored.

Parameters:

pbase - Returned base address

psize - Returned size of device memory allocation

dptr - Device pointer to query

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.43 CUresult cuMemGetInfo (size_t * *free*, size_t * *total*)

Returns in `*free` and `*total` respectively, the free and total amount of memory available for allocation by the CUDA context, in bytes.

Parameters:

free - Returned free memory in bytes

total - Returned total memory in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.44 CUresult cuMemHostAlloc (void ** *pp*, size_t *bytesize*, unsigned int *Flags*)

Allocates `bytesize` bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as cuMemcpyHtoD(). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc(). Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

The `Flags` parameter enables different options to be specified that affect the allocation, as follows.

- CU_MEMHOSTALLOC_PORTABLE: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- CU_MEMHOSTALLOC_DEVICEMAP: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling cuMemHostGetDevicePointer(). This feature is available only on GPUs with compute capability greater than or equal to 1.1.
- CU_MEMHOSTALLOC_WRITECOMBINED: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the GPU via mapped pinned memory or host->device transfers.

All of these flags are orthogonal to one another: a developer may allocate memory that is portable, mapped and/or write-combined with no restrictions.

The CUDA context must have been created with the CU_CTX_MAP_HOST flag in order for the CU_MEMHOSTALLOC_DEVICEMAP flag to have any effect.

The CU_MEMHOSTALLOC_DEVICEMAP flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to cuMemHostGetDevicePointer() because the memory may be mapped into other CUDA contexts via the CU_MEMHOSTALLOC_PORTABLE flag.

The memory allocated by this function must be freed with cuMemFreeHost().

Note all host memory allocated using cuMemHostAlloc() will automatically be immediately accessible to all contexts on all devices which support unified addressing (as may be queried using CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING). Unless the flag CU_MEMHOSTALLOC_WRITECOMBINED is specified, the device pointer that may be used to access this host memory from those contexts is always equal to the returned host pointer `*pp`. If the flag CU_MEMHOSTALLOC_WRITECOMBINED is specified, then the function cuMemHostGetDevicePointer() must be used to query the device pointer, even if the context supports unified addressing. See Unified Addressing for additional details.

Parameters:

pp - Returned host pointer to page-locked memory

bytesize - Requested allocation size in bytes

Flags - Flags for allocation request

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.45 CUresult cuMemHostGetDevicePointer (CUdeviceptr * *pdptr*, void * *p*, unsigned int *Flags*)

Passes back the device pointer `pdptr` corresponding to the mapped, pinned host buffer `p` allocated by cuMemHostAlloc.

cuMemHostGetDevicePointer() will fail if the CU_MEMHOSTALLOC_DEVICEMAP flag was not specified at the time the memory was allocated, or if the function is called on a GPU that does not support mapped pinned memory.

`Flags` provides for future releases. For now, it must be set to 0.

Parameters:

pdptr - Returned device pointer

p - Host pointer

Flags - Options (must be 0)

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

5.37.2.46 CUresult cuMemHostGetFlags (unsigned int * *pFlags*, void * *p*)

Passes back the flags `pFlags` that were specified when allocating the pinned host buffer `p` allocated by cuMemHostAlloc.

cuMemHostGetFlags() will fail if the pointer does not reside in an allocation performed by cuMemAllocHost() or cuMemHostAlloc().

Parameters:

pFlags - Returned flags word

p - Host pointer

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemAllocHost, cuMemHostAlloc

5.37.2.47 CUresult cuMemHostRegister (void * *p*, size_t *bytesize*, unsigned int *Flags*)

Page-locks the memory range specified by `p` and `bytesize` and maps it for the device(s) as specified by `Flags`. This memory range also is added to the same tracking mechanism as cuMemHostAlloc to automatically accelerate calls to functions such as cuMemcpyHtoD(). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory that has not been registered. Page-locking excessive amounts of memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to register staging areas for data exchange between host and device.

This function has limited support on Mac OS X. OS 10.7 or higher is required.

The `Flags` parameter enables different options to be specified that affect the allocation, as follows.

- CU_MEMHOSTREGISTER_PORTABLE: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.

- CU_MEMHOSTREGISTER_DEVICEMAP: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling cuMemHostGetDevicePointer(). This feature is available only on GPUs with compute capability greater than or equal to 1.1.

All of these flags are orthogonal to one another: a developer may page-lock memory that is portable or mapped with no restrictions.

The CUDA context must have been created with the CU_CTX_MAP_HOST flag in order for the CU_MEMHOSTREGISTER_DEVICEMAP flag to have any effect.

The CU_MEMHOSTREGISTER_DEVICEMAP flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to cuMemHostGetDevicePointer() because the memory may be mapped into other CUDA contexts via the CU_MEMHOSTREGISTER_PORTABLE flag.

The memory page-locked by this function must be unregistered with cuMemHostUnregister().

Parameters:

p - Host pointer to memory to page-lock

bytesize - Size in bytes of the address range to page-lock

Flags - Flags for allocation request

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_HOST_MEMORY_ALREADY_REGISTERED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemHostUnregister, cuMemHostGetFlags, cuMemHostGetDevicePointer

5.37.2.48 CUresult cuMemHostUnregister (void * *p*)

Unmaps the memory range whose base address is specified by `p`, and makes it pageable again.

The base address must be the same one specified to cuMemHostRegister().

Parameters:

p - Host pointer to memory to unregister

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_HOST_MEMORY_NOT_REGISTERED,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemHostRegister

5.37.2.49 CUresult cuMemsetD16 (CUdeviceptr *dstDevice*, unsigned short *us*, size_t *N*)

Sets the memory range of `N` 16-bit values to the specified value `us`. The `dstDevice` pointer must be two byte aligned.

Note that this function is asynchronous with respect to the host unless `dstDevice` refers to pinned host memory.

Parameters:

dstDevice - Destination device pointer

us - Value to set

N - Number of elements

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.50 CUresult cuMemsetD16Async (CUdeviceptr *dstDevice*, unsigned short *us*, size_t *N*, CUstream *hStream*)

Sets the memory range of `N` 16-bit values to the specified value `us`. The `dstDevice` pointer must be two byte aligned.

cuMemsetD16Async() is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

Parameters:

dstDevice - Destination device pointer

us - Value to set

N - Number of elements

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD32, cuMemsetD32Async

5.37.2.51 CUresult cuMemsetD2D16 (CUdeviceptr *dstDevice*, size_t *dstPitch*, unsigned short *us*, size_t *Width*, size_t *Height*)

Sets the 2D memory range of `Width` 16-bit values to the specified value `us`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. The `dstDevice` pointer and `dstPitch` offset must be two byte aligned. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

Note that this function is asynchronous with respect to the host unless `dstDevice` refers to pinned host memory.

Parameters:

dstDevice - Destination device pointer

dstPitch - Pitch of destination device pointer

us - Value to set

Width - Width of row

Height - Number of rows

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.52 CUresult cuMemsetD2D16Async (CUdeviceptr *dstDevice*, size_t *dstPitch*, unsigned short *us*, size_t *Width*, size_t *Height*, CUstream *hStream*)

Sets the 2D memory range of `Width` 16-bit values to the specified value `us`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. The `dstDevice` pointer and `dstPitch` offset must be two byte aligned. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

cuMemsetD2D16Async() is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

Parameters:

dstDevice - Destination device pointer

dstPitch - Pitch of destination device pointer

us - Value to set

Width - Width of row

Height - Number of rows

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAl-loc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAto-HAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFree-Host, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMem-setD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.53 CUresult cuMemsetD2D32 (CUdeviceptr *dstDevice*, size_t *dstPitch*, unsigned int *ui*, size_t *Width*, size_t *Height*)

Sets the 2D memory range of `Width` 32-bit values to the specified value `ui`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. The `dstDevice` pointer and `dstPitch` offset must be four byte aligned. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

Note that this function is asynchronous with respect to the host unless `dstDevice` refers to pinned host memory.

Parameters:

dstDevice - Destination device pointer

dstPitch - Pitch of destination device pointer

ui - Value to set

Width - Width of row

Height - Number of rows

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAl-loc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned,

cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.54 CUresult cuMemsetD2D32Async (CUdeviceptr *dstDevice*, size_t *dstPitch*, unsigned int *ui*, size_t *Width*, size_t *Height*, CUstream *hStream*)

Sets the 2D memory range of `Width` 32-bit values to the specified value `ui`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. The `dstDevice` pointer and `dstPitch` offset must be four byte aligned. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

cuMemsetD2D32Async() is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

Parameters:

dstDevice - Destination device pointer

dstPitch - Pitch of destination device pointer

ui - Value to set

Width - Width of row

Height - Number of rows

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.55 CUresult cuMemsetD2D8 (CUdeviceptr *dstDevice*, size_t *dstPitch*, unsigned char *uc*, size_t *Width*, size_t *Height*)

Sets the 2D memory range of `Width` 8-bit values to the specified value `uc`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

Note that this function is asynchronous with respect to the host unless `dstDevice` refers to pinned host memory.

Parameters:

dstDevice - Destination device pointer

dstPitch - Pitch of destination device pointer

uc - Value to set

Width - Width of row

Height - Number of rows

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.56 CUresult cuMemsetD2D8Async (CUdeviceptr *dstDevice*, size_t *dstPitch*, unsigned char *uc*, size_t *Width*, size_t *Height*, CUstream *hStream*)

Sets the 2D memory range of `Width` 8-bit values to the specified value `uc`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

cuMemsetD2D8Async() is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

Parameters:

dstDevice - Destination device pointer

dstPitch - Pitch of destination device pointer

uc - Value to set

Width - Width of row

Height - Number of rows

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.57 CUresult cuMemsetD32 (CUdeviceptr *dstDevice*, unsigned int *ui*, size_t *N*)

Sets the memory range of `N` 32-bit values to the specified value `ui`. The `dstDevice` pointer must be four byte aligned.

Note that this function is asynchronous with respect to the host unless `dstDevice` refers to pinned host memory.

Parameters:

dstDevice - Destination device pointer

ui - Value to set

N - Number of elements

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32Async

5.37.2.58 CUresult cuMemsetD32Async (CUdeviceptr *dstDevice*, unsigned int *ui*, size_t *N*, CUstream *hStream*)

Sets the memory range of `N` 32-bit values to the specified value `ui`. The `dstDevice` pointer must be four byte aligned.

cuMemsetD32Async() is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

Parameters:

dstDevice - Destination device pointer

ui - Value to set

N - Number of elements

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32

5.37.2.59 CUresult cuMemsetD8 (CUdeviceptr *dstDevice*, unsigned char *uc*, size_t *N*)

Sets the memory range of `N` 8-bit values to the specified value `uc`.

Note that this function is asynchronous with respect to the host unless `dstDevice` refers to pinned host memory.

Parameters:

dstDevice - Destination device pointer

uc - Value to set

N - Number of elements

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.60 CUresult cuMemsetD8Async (CUdeviceptr *dstDevice*, unsigned char *uc*, size_t *N*, CUstream *hStream*)

Sets the memory range of `N` 8-bit values to the specified value `uc`.

cuMemsetD8Async() is asynchronous and can optionally be associated to a stream by passing a non-zero `stream` argument.

Parameters:

dstDevice - Destination device pointer

uc - Value to set

N - Number of elements

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

5.37.2.61 CUresult cuMipmappedArrayCreate (CUmipmappedArray * *pHandle*, const CUDA_ARRAY3D_DESCRIPTOR * *pMipmappedArrayDesc*, unsigned int *numMipmapLevels*)

Creates a CUDA mipmapped array according to the CUDA_ARRAY3D_DESCRIPTOR structure `pMipmappedArrayDesc` and returns a handle to the new CUDA mipmapped array in `*pHandle`. `numMipmapLevels` specifies the number of mipmap levels to be allocated. This value is clamped to the range [1, 1 + floor(log2(max(width, height, depth)))].

The CUDA_ARRAY3D_DESCRIPTOR is defined as:

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    unsigned int Depth;
    CUarray_format Format;
    unsigned int NumChannels;
    unsigned int Flags;
} CUDA_ARRAY3D_DESCRIPTOR;
```

where:

- Width, Height, and Depth are the width, height, and depth of the CUDA array (in elements); the following types of CUDA arrays can be allocated:
  - A 1D mipmapped array is allocated if Height and Depth extents are both zero.
  - A 2D mipmapped array is allocated if only Depth extent is zero.
  - A 3D mipmapped array is allocated if all three extents are non-zero.
  - A 1D layered CUDA mipmapped array is allocated if only Height is zero and the CUDA_ARRAY3D_LAYERED flag is set. Each layer is a 1D array. The number of layers is determined by the depth extent.
  - A 2D layered CUDA mipmapped array is allocated if all three extents are non-zero and the CUDA_ARRAY3D_LAYERED flag is set. Each layer is a 2D array. The number of layers is determined by the depth extent.
  - A cubemap CUDA mipmapped array is allocated if all three extents are non-zero and the CUDA_ARRAY3D_CUBEMAP flag is set. Width must be equal to Height, and Depth must be six. A cubemap is a special type of 2D layered CUDA array, where the six layers represent the six faces of a cube. The order of the six layers in memory is the same as that listed in CUarray_cubemap_face.
  - A cubemap layered CUDA mipmapped array is allocated if all three extents are non-zero, and both, CUDA_ARRAY3D_CUBEMAP and CUDA_ARRAY3D_LAYERED flags are set. Width must be equal to Height, and Depth must be a multiple of six. A cubemap layered CUDA array is a special type of 2D layered CUDA array that consists of a collection of cubemaps. The first six layers represent the first cubemap, the next six layers form the second cubemap, and so on.

- Format specifies the format of the elements; CUarray_format is defined as:

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20
} CUarray_format;
```

- NumChannels specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;

- Flags may be set to
  - CUDA_ARRAY3D_LAYERED to enable creation of layered CUDA mipmapped arrays. If this flag is set, Depth specifies the number of layers, not the depth of a 3D array.
  - CUDA_ARRAY3D_SURFACE_LDST to enable surface references to be bound to individual mipmap levels of the CUDA mipmapped array. If this flag is not set, cuSurfRefSetArray will fail when attempting to bind a mipmap level of the CUDA mipmapped array to a surface reference.
  - CUDA_ARRAY3D_CUBEMAP to enable creation of mipmapped cubemaps. If this flag is set, Width must be equal to Height, and Depth must be six. If the CUDA_ARRAY3D_LAYERED flag is also set, then Depth must be a multiple of six.
  - CUDA_ARRAY3D_TEXTURE_GATHER to indicate that the CUDA mipmapped array will be used for texture gather. Texture gather can only be performed on 2D CUDA mipmapped arrays.

Width, Height and Depth must meet certain size requirements as listed in the following table. All values are specified in elements. Note that for brevity's sake, the full name of the device attribute is not specified. For ex., TEXTURE1D_MIPMAPPED_WIDTH refers to the device attribute CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH.

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}
1D	{ (1,TEXTURE1D_MIPMAPPED_WIDTH), 0, 0 }
2D	{ (1,TEXTURE2D_MIPMAPPED_WIDTH), (1,TEXTURE2D_MIPMAPPED_HEIGHT), 0 }
3D	{ (1,TEXTURE3D_WIDTH), (1,TEXTURE3D_HEIGHT), (1,TEXTURE3D_DEPTH) } OR { (1,TEXTURE3D_WIDTH_ALTERNATE), (1,TEXTURE3D_HEIGHT_ALTERNATE), (1,TEXTURE3D_DEPTH_ALTERNATE) }
1D Layered	{ (1,TEXTURE1D_LAYERED_WIDTH), 0, (1,TEXTURE1D_LAYERED_LAYERS) }
2D Layered	{ (1,TEXTURE2D_LAYERED_WIDTH), (1,TEXTURE2D_LAYERED_HEIGHT), (1,TEXTURE2D_LAYERED_LAYERS) }
Cubemap	{ (1,TEXTURECUBEMAP_WIDTH), (1,TEXTURECUBEMAP_WIDTH), 6 }
Cubemap Layered	{ (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_LAYERS) }

Parameters:

pHandle - Returned mipmapped array

pMipmappedArrayDesc - mipmapped array descriptor

numMipmapLevels - Number of mipmap levels

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMipmappedArrayDestroy, cuMipmappedArrayGetLevel, cuArrayCreate,

5.37.2.62 CUresult cuMipmappedArrayDestroy (CUmipmappedArray *hMipmappedArray*)

Destroys the CUDA mipmapped array `hMipmappedArray`.

Parameters:

hMipmappedArray - Mipmapped array to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ARRAY_IS_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMipmappedArrayCreate, cuMipmappedArrayGetLevel, cuArrayCreate,

5.37.2.63 CUresult cuMipmappedArrayGetLevel (CUarray * *pLevelArray*, CUmipmappedArray *hMipmappedArray*, unsigned int *level*)

Returns in `*pLevelArray` a CUDA array that represents a single mipmap level of the CUDA mipmapped array `hMipmappedArray`.

If `level` is greater than the maximum number of levels in this mipmapped array, CUDA_ERROR_INVALID_VALUE is returned.

Parameters:

pLevelArray - Returned mipmap level CUDA array

hMipmappedArray - CUDA mipmapped array

level - Mipmap level

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMipmappedArrayCreate, cuMipmappedArrayDestroy, cuArrayCreate,

5.38 Unified Addressing

Functions

- CUresult cuPointerGetAttribute (void *data, CUpointer_attribute attribute, CUdeviceptr ptr)

  Returns information about a pointer.

5.38.1 Detailed Description

This section describes the unified addressing functions of the low-level CUDA driver application programming interface.

5.38.2 Overview

CUDA devices can share a unified address space with the host. For these devices there is no distinction between a device pointer and a host pointer – the same pointer value may be used to access memory from the host program and from a kernel running on the device (with exceptions enumerated below).

5.38.3 Supported Platforms

Whether or not a device supports unified addressing may be queried by calling cuDeviceGetAttribute() with the device attribute CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING.

Unified addressing is automatically enabled in 64-bit processes on devices with compute capability greater than or equal to 2.0.

5.38.4 Looking Up Information from Pointer Values

It is possible to look up information about the memory which backs a pointer value. For instance, one may want to know if a pointer points to host or device memory. As another example, in the case of device memory, one may want to know on which CUDA device the memory resides. These properties may be queried using the function cuPointerGetAttribute()

Since pointers are unique, it is not necessary to specify information about the pointers specified to the various copy functions in the CUDA API. The function cuMemcpy() may be used to perform a copy between two pointers, ignoring whether they point to host or device memory (making cuMemcpyHtoD(), cuMemcpyDtoD(), and cuMemcpyDtoH() unnecessary for devices supporting unified addressing). For multidimensional copies, the memory type CU_MEMORYTYPE_UNIFIED may be used to specify that the CUDA driver should infer the location of the pointer from its value.

5.38.5 Automatic Mapping of Host Allocated Host Memory

All host memory allocated in all contexts using cuMemAllocHost() and cuMemHostAlloc() is always directly accessible from all contexts on all devices that support unified addressing. This is the case regardless of whether or not the flags CU_MEMHOSTALLOC_PORTABLE and CU_MEMHOSTALLOC_DEVICEMAP are specified.

The pointer value through which allocated host memory may be accessed in kernels on all devices that support unified addressing is the same as the pointer value through which that memory is accessed on the host, so it is not necessary to call cuMemHostGetDevicePointer() to get the device pointer for these allocations.

Note that this is not the case for memory allocated using the flag CU_MEMHOSTALLOC_WRITECOMBINED, as discussed below.

5.38.6 Automatic Registration of Peer Memory

Upon enabling direct access from a context that supports unified addressing to another peer context that supports unified addressing using cuCtxEnablePeerAccess() all memory allocated in the peer context using cuMemAlloc() and cuMemAllocPitch() will immediately be accessible by the current context. The device pointer value through which any peer memory may be accessed in the current context is the same pointer value through which that memory may be accessed in the peer context.

5.38.7 Exceptions, Disjoint Addressing

Not all memory may be accessed on devices through the same pointer value through which they are accessed on the host. These exceptions are host memory registered using cuMemHostRegister() and host memory allocated using the flag CU_MEMHOSTALLOC_WRITECOMBINED. For these exceptions, there exists a distinct host and device address for the memory. The device address is guaranteed to not overlap any valid host pointer range and is guaranteed to have the same value across all contexts that support unified addressing.

This device address may be queried using cuMemHostGetDevicePointer() when a context using unified addressing is current. Either the host or the unified device pointer value may be used to refer to this memory through cuMemcpy() and similar functions using the CU_MEMORYTYPE_UNIFIED memory type.

5.38.8 Function Documentation

5.38.8.1 CUresult cuPointerGetAttribute (void * *data*, CUpointer_attribute *attribute*, CUdeviceptr *ptr*)

The supported attributes are:

- CU_POINTER_ATTRIBUTE_CONTEXT:

Returns in `*data` the CUcontext in which `ptr` was allocated or registered. The type of `data` must be CUcontext *.

If `ptr` was not allocated by, mapped by, or registered with a CUcontext which uses unified virtual addressing then CUDA_ERROR_INVALID_VALUE is returned.

- CU_POINTER_ATTRIBUTE_MEMORY_TYPE:

Returns in `*data` the physical memory type of the memory that `ptr` addresses as a CUmemorytype enumerated value. The type of `data` must be unsigned int.

If `ptr` addresses device memory then `*data` is set to CU_MEMORYTYPE_DEVICE. The particular CUdevice on which the memory resides is the CUdevice of the CUcontext returned by the CU_POINTER_ATTRIBUTE_-CONTEXT attribute of `ptr`.

If `ptr` addresses host memory then `*data` is set to CU_MEMORYTYPE_HOST.

If `ptr` was not allocated by, mapped by, or registered with a CUcontext which uses unified virtual addressing then CUDA_ERROR_INVALID_VALUE is returned.

If the current CUcontext does not support unified virtual addressing then CUDA_ERROR_INVALID_CONTEXT is returned.

- CU_POINTER_ATTRIBUTE_DEVICE_POINTER:

Returns in *data the device pointer value through which ptr may be accessed by kernels running in the current CUcontext. The type of data must be CUdeviceptr *.

If there exists no device pointer value through which kernels running in the current CUcontext may access ptr then CUDA_ERROR_INVALID_VALUE is returned.

If there is no current CUcontext then CUDA_ERROR_INVALID_CONTEXT is returned.

Except in the exceptional disjoint addressing cases discussed below, the value returned in *data will equal the input value ptr.

- CU_POINTER_ATTRIBUTE_HOST_POINTER:

Returns in *data the host pointer value through which ptr may be accessed by by the host program. The type of data must be void **. If there exists no host pointer value through which the host program may directly access ptr then CUDA_ERROR_INVALID_VALUE is returned.

Except in the exceptional disjoint addressing cases discussed below, the value returned in *data will equal the input value ptr.

- CU_POINTER_ATTRIBUTE_P2P_TOKENS:

Returns in *data two tokens for use with the nv-p2p.h Linux kernel interface. data must be a struct of type CUDA_POINTER_ATTRIBUTE_P2P_TOKENS.

ptr must be a pointer to memory obtained from :cuMemAlloc(). Note that p2pToken and vaSpaceToken are only valid for the lifetime of the source allocation. A subsequent allocation at the same address may return completely different tokens.

Note that for most allocations in the unified virtual address space the host and device pointer for accessing the allocation will be the same. The exceptions to this are

- user memory registered using cuMemHostRegister
- host memory allocated using cuMemHostAlloc with the CU_MEMHOSTALLOC_WRITECOMBINED flag For these types of allocation there will exist separate, disjoint host and device addresses for accessing the allocation. In particular
- The host address will correspond to an invalid unmapped device address (which will result in an exception if accessed from the device)
- The device address will correspond to an invalid unmapped host address (which will result in an exception if accessed from the host). For these types of allocations, querying CU_POINTER_ATTRIBUTE_HOST_POINTER and CU_POINTER_ATTRIBUTE_DEVICE_POINTER may be used to retrieve the host and device addresses from either address.

Parameters:

data - Returned pointer attribute value

attribute - Pointer attribute to query

ptr - Pointer

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuMemAlloc, cuMemFree, cuMemAllocHost, cuMemFreeHost, cuMemHostAlloc, cuMemHostRegister, cuMemHostUnregister

5.39 Stream Management

Functions

- CUresult cuStreamAddCallback (CUstream hStream, CUstreamCallback callback, void ∗userData, unsigned int flags)

  Add a callback to a compute stream.

- CUresult cuStreamCreate (CUstream ∗phStream, unsigned int Flags)

  Create a stream.

- CUresult cuStreamDestroy (CUstream hStream)

  Destroys a stream.

- CUresult cuStreamQuery (CUstream hStream)

  Determine status of a compute stream.

- CUresult cuStreamSynchronize (CUstream hStream)

  Wait until a stream's tasks are completed.

- CUresult cuStreamWaitEvent (CUstream hStream, CUevent hEvent, unsigned int Flags)

  Make a compute stream wait on an event.

5.39.1 Detailed Description

This section describes the stream management functions of the low-level CUDA driver application programming interface.

5.39.2 Function Documentation

5.39.2.1 CUresult cuStreamAddCallback (CUstream *hStream*, CUstreamCallback *callback*, void ∗ *userData*, unsigned int *flags*)

Adds a callback to be called on the host after all currently enqueued items in the stream have completed. For each cuStreamAddCallback call, the callback will be executed exactly once. The callback will block later work in the stream until it is finished.

The callback may be passed CUDA_SUCCESS or an error code. In the event of a device error, all subsequently executed callbacks will receive an appropriate CUresult.

Callbacks must not make any CUDA API calls. Attempting to use a CUDA API will result in CUDA_ERROR_NOT_PERMITTED. Callbacks must not perform any synchronization that may depend on outstanding device work or other callbacks that are not mandated to run earlier. Callbacks without a mandated order (in independent streams) execute in undefined order and may be serialized.

This API requires compute capability 1.1 or greater. See cuDeviceGetAttribute or cuDeviceGetProperties to query compute capability. Attempting to use this API with earlier compute versions will return CUDA_ERROR_NOT_SUPPORTED.

Parameters:

hStream - Stream to add callback to

callback - The function to call once preceding stream operations are complete

userData - User specified data to be passed to the callback function

flags - Reserved for future use, must be 0

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_SUPPORTED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuStreamQuery, cuStreamSynchronize, cuStreamWaitEvent, cuStreamDestroy

5.39.2.2 CUresult cuStreamCreate (CUstream * *phStream*, unsigned int *Flags*)

Creates a stream and returns a handle in `phStream`. The `Flags` argument determines behaviors of the stream. Valid values for `Flags` are:

- CU_STREAM_DEFAULT: Default stream creation flag.
- CU_STREAM_NON_BLOCKING: Specifies that work running in the created stream may run concurrently with work in stream 0 (the NULL stream), and that the created stream should perform no implicit synchronization with stream 0.

Parameters:

phStream - Returned newly created stream

Flags - Parameters for stream creation

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamDestroy, cuStreamWaitEvent, cuStreamQuery, cuStreamSynchronize, cuStreamAddCallback

5.39.2.3 CUresult cuStreamDestroy (CUstream *hStream*)

Destroys the stream specified by `hStream`.

In case the device is still doing work in the stream `hStream` when cuStreamDestroy() is called, the function will return immediately and the resources associated with `hStream` will be released automatically once the device has completed all work in `hStream`.

Parameters:

hStream - Stream to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuStreamWaitEvent, cuStreamQuery, cuStreamSynchronize, cuStreamAddCallback

5.39.2.4 CUresult cuStreamQuery (CUstream *hStream*)

Returns CUDA_SUCCESS if all operations in the stream specified by `hStream` have completed, or CUDA_-ERROR_NOT_READY if not.

Parameters:

hStream - Stream to query status of

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_READY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuStreamWaitEvent, cuStreamDestroy, cuStreamSynchronize, cuStreamAddCallback

5.39.2.5 CUresult cuStreamSynchronize (CUstream *hStream*)

Waits until the device has completed all operations in the stream specified by `hStream`. If the context was created with the CU_CTX_SCHED_BLOCKING_SYNC flag, the CPU thread will block until the stream is finished with all of its tasks.

Parameters:

hStream - Stream to wait for

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuStreamDestroy, cuStreamWaitEvent, cuStreamQuery, cuStreamAddCallback

5.39.2.6 CUresult cuStreamWaitEvent (CUstream *hStream*, CUevent *hEvent*, unsigned int *Flags*)

Makes all future work submitted to `hStream` wait until `hEvent` reports completion before beginning execution. This synchronization will be performed efficiently on the device. The event `hEvent` may be from a different context than `hStream`, in which case this function will perform cross-device synchronization.

The stream `hStream` will wait only for the completion of the most recent host call to cuEventRecord() on `hEvent`. Once this call has returned, any functions (including cuEventRecord() and cuEventDestroy()) may be called on `hEvent` again, and subsequent calls will not have any effect on `hStream`.

If `hStream` is 0 (the NULL stream) any future work submitted in any stream will wait for `hEvent` to complete before beginning execution. This effectively creates a barrier for all future work submitted to the context.

If cuEventRecord() has not been called on `hEvent`, this call acts as if the record has already completed, and so is a functional no-op.

Parameters:

hStream - Stream to wait

hEvent - Event to wait on (may not be NULL)

Flags - Parameters for the operation (must be 0)

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuEventRecord, cuStreamQuery, cuStreamSynchronize, cuStreamAddCallback, cuStreamDestroy

5.40 Event Management

Functions

- CUresult cuEventCreate (CUevent ∗phEvent, unsigned int Flags)

  Creates an event.

- CUresult cuEventDestroy (CUevent hEvent)

  Destroys an event.

- CUresult cuEventElapsedTime (float ∗pMilliseconds, CUevent hStart, CUevent hEnd)

  Computes the elapsed time between two events.

- CUresult cuEventQuery (CUevent hEvent)

  Queries an event's status.

- CUresult cuEventRecord (CUevent hEvent, CUstream hStream)

  Records an event.

- CUresult cuEventSynchronize (CUevent hEvent)

  Waits for an event to complete.

5.40.1 Detailed Description

This section describes the event management functions of the low-level CUDA driver application programming interface.

5.40.2 Function Documentation

5.40.2.1 CUresult cuEventCreate (CUevent ∗ *phEvent*, unsigned int *Flags*)

Creates an event ∗phEvent with the flags specified via `Flags`. Valid flags include:

- CU_EVENT_DEFAULT: Default event creation flag.
- CU_EVENT_BLOCKING_SYNC: Specifies that the created event should use blocking synchronization. A CPU thread that uses cuEventSynchronize() to wait on an event created with this flag will block until the event has actually been recorded.
- CU_EVENT_DISABLE_TIMING: Specifies that the created event does not need to record timing data. Events created with this flag specified and the CU_EVENT_BLOCKING_SYNC flag not specified will provide the best performance when used with cuStreamWaitEvent() and cuEventQuery().
- CU_EVENT_INTERPROCESS: Specifies that the created event may be used as an interprocess event by cuIpcGetEventHandle(). CU_EVENT_INTERPROCESS must be specified along with CU_EVENT_-DISABLE_TIMING.

Parameters:

phEvent - Returns newly created event

Flags - Event creation flags

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventDestroy, cuEventElapsedTime

5.40.2.2 CUresult cuEventDestroy (CUevent *hEvent*)

Destroys the event specified by `hEvent`.

In case `hEvent` has been recorded but has not yet been completed when cuEventDestroy() is called, the function will return immediately and the resources associated with `hEvent` will be released automatically once the device has completed `hEvent`.

Parameters:

hEvent - Event to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventCreate, cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventElapsedTime

5.40.2.3 CUresult cuEventElapsedTime (float * *pMilliseconds*, CUevent *hStart*, CUevent *hEnd*)

Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds).

If either event was last recorded in a non-NULL stream, the resulting time may be greater than expected (even if both used the same stream handle). This happens because the cuEventRecord() operation takes place asynchronously and there is no guarantee that the measured latency is actually just between the two events. Any number of other different stream operations could execute in between the two measured events, thus altering the timing in a significant way.

If cuEventRecord() has not been called on either event then CUDA_ERROR_INVALID_HANDLE is returned. If cuEventRecord() has been called on both events but one or both of them has not yet been completed (that is, cuEvent-Query() would return CUDA_ERROR_NOT_READY on at least one of the events), CUDA_ERROR_NOT_READY is returned. If either event was created with the CU_EVENT_DISABLE_TIMING flag, then this function will return CUDA_ERROR_INVALID_HANDLE.

Parameters:

pMilliseconds - Time between `hStart` and `hEnd` in ms

hStart - Starting event

hEnd - Ending event

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_READY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventCreate, cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventDestroy

5.40.2.4 CUresult cuEventQuery (CUevent *hEvent*)

Query the status of all device work preceding the most recent call to cuEventRecord() (in the appropriate compute streams, as specified by the arguments to cuEventRecord()).

If this work has successfully been completed by the device, or if cuEventRecord() has not been called on `hEvent`, then CUDA_SUCCESS is returned. If this work has not yet been completed by the device then CUDA_ERROR_-NOT_READY is returned.

Parameters:

hEvent - Event to query

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_READY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventCreate, cuEventRecord, cuEventSynchronize, cuEventDestroy, cuEventElapsedTime

5.40.2.5 CUresult cuEventRecord (CUevent *hEvent*, CUstream *hStream*)

Records an event. If `hStream` is non-zero, the event is recorded after all preceding operations in `hStream` have been completed; otherwise, it is recorded after all preceding operations in the CUDA context have been completed. Since operation is asynchronous, cuEventQuery and/or cuEventSynchronize() must be used to determine when the event has actually been recorded.

If cuEventRecord() has previously been called on `hEvent`, then this call will overwrite any existing state in `hEvent`. Any subsequent calls which examine the status of `hEvent` will only examine the completion of this most recent call to cuEventRecord().

It is necessary that `hEvent` and `hStream` be created on the same context.

Parameters:

hEvent - Event to record

hStream - Stream to record event for

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventCreate, cuEventQuery, cuEventSynchronize, cuStreamWaitEvent, cuEventDestroy, cuEventElapsedTime

5.40.2.6 CUresult cuEventSynchronize (CUevent *hEvent*)

Wait until the completion of all device work preceding the most recent call to cuEventRecord() (in the appropriate compute streams, as specified by the arguments to cuEventRecord()).

If cuEventRecord() has not been called on `hEvent`, CUDA_SUCCESS is returned immediately.

Waiting for an event that was created with the CU_EVENT_BLOCKING_SYNC flag will cause the calling CPU thread to block until the event has been completed by the device. If the CU_EVENT_BLOCKING_SYNC flag has not been set, then the CPU thread will busy-wait until the event has been completed by the device.

Parameters:

hEvent - Event to wait for

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventCreate, cuEventRecord, cuEventQuery, cuEventDestroy, cuEventElapsedTime

5.41 Execution Control

Functions

- CUresult cuFuncGetAttribute (int *pi, CUfunction_attribute attrib, CUfunction hfunc)

  Returns information about a function.

- CUresult cuFuncSetCacheConfig (CUfunction hfunc, CUfunc_cache config)

  Sets the preferred cache configuration for a device function.

- CUresult cuFuncSetSharedMemConfig (CUfunction hfunc, CUsharedconfig config)

  Sets the shared memory configuration for a device function.

- CUresult cuLaunchKernel (CUfunction f, unsigned int gridDimX, unsigned int gridDimY, unsigned int gridDimZ, unsigned int blockDimX, unsigned int blockDimY, unsigned int blockDimZ, unsigned int sharedMemBytes, CUstream hStream, void **kernelParams, void **extra)

  Launches a CUDA function.

5.41.1 Detailed Description

This section describes the execution control functions of the low-level CUDA driver application programming interface.

5.41.2 Function Documentation

5.41.2.1 CUresult cuFuncGetAttribute (int * *pi*, CUfunction_attribute *attrib*, CUfunction *hfunc*)

Returns in `*pi` the integer value of the attribute `attrib` on the kernel given by `hfunc`. The supported attributes are:

- CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK: The maximum number of threads per block, beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.

- CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES: The size in bytes of statically-allocated shared memory per block required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.

- CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES: The size in bytes of user-allocated constant memory required by this function.

- CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES: The size in bytes of local memory used by each thread of this function.

- CU_FUNC_ATTRIBUTE_NUM_REGS: The number of registers used by each thread of this function.

- CU_FUNC_ATTRIBUTE_PTX_VERSION: The PTX virtual architecture version for which the function was compiled. This value is the major PTX version * 10 + the minor PTX version, so a PTX version 1.3 function would return the value 13. Note that this may return the undefined value of 0 for cubins compiled prior to CUDA 3.0.

- CU_FUNC_ATTRIBUTE_BINARY_VERSION: The binary architecture version for which the function was compiled. This value is the major binary version * 10 + the minor binary version, so a binary version 1.3 function would return the value 13. Note that this will return a value of 10 for legacy cubins that do not have a properly-encoded binary architecture version.

Parameters:

pi - Returned attribute value

attrib - Attribute requested

hfunc - Function to query attribute of

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuFuncSetCacheConfig, cuLaunchKernel

5.41.2.2 CUresult cuFuncSetCacheConfig (CUfunction *hfunc*, CUfunc_cache *config*)

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `config` the preferred cache configuration for the device function `hfunc`. This is only a preference. The driver will use the requested configuration if possible, but it is free to choose a different configuration if required to execute `hfunc`. Any context-wide preference set via cuCtxSetCacheConfig() will be overridden by this per-function setting unless the per-function setting is CU_FUNC_CACHE_PREFER_NONE. In that case, the current context-wide setting will be used.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- CU_FUNC_CACHE_PREFER_NONE: no preference for shared memory or L1 (default)
- CU_FUNC_CACHE_PREFER_SHARED: prefer larger shared memory and smaller L1 cache
- CU_FUNC_CACHE_PREFER_L1: prefer larger L1 cache and smaller shared memory
- CU_FUNC_CACHE_PREFER_EQUAL: prefer equal sized L1 cache and shared memory

Parameters:

hfunc - Kernel to configure cache for

config - Requested cache configuration

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuFuncGetAttribute, cuLaunchKernel

5.41.2.3 CUresult cuFuncSetSharedMemConfig (CUfunction *hfunc*, CUsharedconfig *config*)

On devices with configurable shared memory banks, this function will force all subsequent launches of the specified device function to have the given shared memory bank size configuration. On any given launch of the function, the shared memory configuration of the device will be temporarily changed if needed to suit the function's preferred configuration. Changes in shared memory configuration between subsequent launches of functions, may introduce a device side synchronization point.

Any per-function setting of shared memory bank size set via cuFuncSetSharedMemConfig will override the context wide setting set with cuCtxSetSharedMemConfig.

Changing the shared memory bank size will not increase shared memory usage or affect occupancy of kernels, but may have major effects on performance. Larger bank sizes will allow for greater potential bandwidth to shared memory, but will change what kinds of accesses to shared memory will result in bank conflicts.

This function will do nothing on devices with fixed shared memory bank size.

The supported bank configurations are:

- CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE: use the context's shared memory configuration when launching this function.
- CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE: set shared memory bank width to be natively four bytes when launching this function.
- CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE: set shared memory bank width to be natively eight bytes when launching this function.

Parameters:

hfunc - kernel to be given a shared memory config

config - requested shared memory configuration

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuCtxGetSharedMemConfig cuCtxSetSharedMemConfig cuFuncGetAttribute, cuLaunchKernel

5.41.2.4 CUresult cuLaunchKernel (CUfunction *f*, unsigned int *gridDimX*, unsigned int *gridDimY*, unsigned int *gridDimZ*, unsigned int *blockDimX*, unsigned int *blockDimY*, unsigned int *blockDimZ*, unsigned int *sharedMemBytes*, CUstream *hStream*, void ** *kernelParams*, void ** *extra*)

Invokes the kernel `f` on a `gridDimX` x `gridDimY` x `gridDimZ` grid of blocks. Each block contains `blockDimX` x `blockDimY` x `blockDimZ` threads.

`sharedMemBytes` sets the amount of dynamic shared memory that will be available to each thread block.

cuLaunchKernel() can optionally be associated to a stream by passing a non-zero `hStream` argument.

Kernel parameters to `f` can be specified in one of two ways:

1) Kernel parameters can be specified via `kernelParams`. If `f` has N parameters, then `kernelParams` needs to be an array of N pointers. Each of `kernelParams`[0] through `kernelParams`[N-1] must point to a region of memory from which the actual kernel parameter will be copied. The number of kernel parameters and their offsets and sizes do not need to be specified as that information is retrieved directly from the kernel's image.

2) Kernel parameters can also be packaged by the application into a single buffer that is passed in via the `extra` parameter. This places the burden on the application of knowing each kernel parameter's size and alignment/padding within the buffer. Here is an example of using the `extra` parameter in this manner:

```
size_t argBufferSize;
char argBuffer[256];

// populate argBuffer and argBufferSize

void *config[] = {
    CU_LAUNCH_PARAM_BUFFER_POINTER, argBuffer,
    CU_LAUNCH_PARAM_BUFFER_SIZE,    &argBufferSize,
    CU_LAUNCH_PARAM_END
};
status = cuLaunchKernel(f, gx, gy, gz, bx, by, bz, sh, s, NULL, config);
```

The `extra` parameter exists to allow cuLaunchKernel to take additional less commonly used arguments. `extra` specifies a list of names of extra settings and their corresponding values. Each extra setting name is immediately followed by the corresponding value. The list must be terminated with either NULL or CU_LAUNCH_PARAM_END.

- CU_LAUNCH_PARAM_END, which indicates the end of the `extra` array;
- CU_LAUNCH_PARAM_BUFFER_POINTER, which specifies that the next value in `extra` will be a pointer to a buffer containing all the kernel parameters for launching kernel `f`;
- CU_LAUNCH_PARAM_BUFFER_SIZE, which specifies that the next value in `extra` will be a pointer to a size_t containing the size of the buffer specified with CU_LAUNCH_PARAM_BUFFER_POINTER;

The error CUDA_ERROR_INVALID_VALUE will be returned if kernel parameters are specified with both `kernelParams` and `extra` (i.e. both `kernelParams` and `extra` are non-NULL).

Calling cuLaunchKernel() sets persistent function state that is the same as function state set through the following deprecated APIs:

cuFuncSetBlockShape() cuFuncSetSharedSize() cuParamSetSize() cuParamSeti() cuParamSetf() cuParamSetv()

When the kernel `f` is launched via cuLaunchKernel(), the previous block shape, shared size and parameter info associated with `f` is overwritten.

Note that to use cuLaunchKernel(), the kernel `f` must either have been compiled with toolchain version 3.2 or later so that it will contain kernel parameter information, or have no kernel parameters. If either of these conditions is not met, then cuLaunchKernel() will return CUDA_ERROR_INVALID_IMAGE.

Parameters:

f - Kernel to launch

gridDimX - Width of grid in blocks

gridDimY - Height of grid in blocks

gridDimZ - Depth of grid in blocks

blockDimX - X dimension of each thread block

blockDimY - Y dimension of each thread block

blockDimZ - Z dimension of each thread block

sharedMemBytes - Dynamic shared-memory size per thread block in bytes

hStream - Stream identifier

kernelParams - Array of pointers to kernel parameters

extra - Extra options

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_-IMAGE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_-LAUNCH_OUT_OF_RESOURCES, CUDA_ERROR_LAUNCH_TIMEOUT, CUDA_ERROR_LAUNCH_-INCOMPATIBLE_TEXTURING, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuFuncSetCacheConfig, cuFuncGetAttribute,

5.42 Execution Control [DEPRECATED]

Functions

- CUresult cuFuncSetBlockShape (CUfunction hfunc, int x, int y, int z)
  Sets the block-dimensions for the function.
- CUresult cuFuncSetSharedSize (CUfunction hfunc, unsigned int bytes)
  Sets the dynamic shared-memory size for the function.
- CUresult cuLaunch (CUfunction f)
  Launches a CUDA function.
- CUresult cuLaunchGrid (CUfunction f, int grid_width, int grid_height)
  Launches a CUDA function.
- CUresult cuLaunchGridAsync (CUfunction f, int grid_width, int grid_height, CUstream hStream)
  Launches a CUDA function.
- CUresult cuParamSetf (CUfunction hfunc, int offset, float value)
  Adds a floating-point parameter to the function's argument list.
- CUresult cuParamSeti (CUfunction hfunc, int offset, unsigned int value)
  Adds an integer parameter to the function's argument list.
- CUresult cuParamSetSize (CUfunction hfunc, unsigned int numbytes)
  Sets the parameter size for the function.
- CUresult cuParamSetTexRef (CUfunction hfunc, int texunit, CUtexref hTexRef)
  Adds a texture-reference to the function's argument list.
- CUresult cuParamSetv (CUfunction hfunc, int offset, void *ptr, unsigned int numbytes)
  Adds arbitrary data to the function's argument list.

5.42.1 Detailed Description

This section describes the deprecated execution control functions of the low-level CUDA driver application programming interface.

5.42.2 Function Documentation

5.42.2.1 CUresult cuFuncSetBlockShape (CUfunction *hfunc*, int *x*, int *y*, int *z*)

Deprecated

Specifies the `x`, `y`, and `z` dimensions of the thread blocks that are created when the kernel given by `hfunc` is launched.

Parameters:

hfunc - Kernel to specify dimensions of

x - X dimension

y - Y dimension

z - Z dimension

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetSharedSize, cuFuncSetCacheConfig, cuFuncGetAttribute, cuParamSetSize, cuParamSeti, cuParamSetf, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

5.42.2.2 CUresult cuFuncSetSharedSize (CUfunction *hfunc*, unsigned int *bytes*)

Deprecated

Sets through `bytes` the amount of dynamic shared memory that will be available to each thread block when the kernel given by `hfunc` is launched.

Parameters:

hfunc - Kernel to specify dynamic shared-memory size for

bytes - Dynamic shared-memory size per thread in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetCacheConfig, cuFuncGetAttribute, cuParamSetSize, cuParamSeti, cuParamSetf, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

5.42.2.3 CUresult cuLaunch (CUfunction *f*)

Deprecated

Invokes the kernel `f` on a 1 x 1 x 1 grid of blocks. The block contains the number of threads specified by a previous call to cuFuncSetBlockShape().

Parameters:

f - Kernel to launch

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES, CUDA_ERROR_LAUNCH_TIMEOUT, CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

5.42.2.4 CUresult cuLaunchGrid (CUfunction *f*, int *grid_width*, int *grid_height*)

Deprecated

Invokes the kernel `f` on a `grid_width` x `grid_height` grid of blocks. Each block contains the number of threads specified by a previous call to cuFuncSetBlockShape().

Parameters:

f - Kernel to launch

grid_width - Width of grid in blocks

grid_height - Height of grid in blocks

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES, CUDA_ERROR_LAUNCH_TIMEOUT, CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGridAsync, cuLaunchKernel

5.42.2.5 CUresult cuLaunchGridAsync (CUfunction *f*, int *grid_width*, int *grid_height*, CUstream *hStream*)

Deprecated

Invokes the kernel `f` on a `grid_width` x `grid_height` grid of blocks. Each block contains the number of threads specified by a previous call to cuFuncSetBlockShape().

cuLaunchGridAsync() can optionally be associated to a stream by passing a non-zero `hStream` argument.

Parameters:

f - Kernel to launch

grid_width - Width of grid in blocks

grid_height - Height of grid in blocks

hStream - Stream identifier

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES, CUDA_ERROR_LAUNCH_TIMEOUT, CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchKernel

5.42.2.6 CUresult cuParamSetf (CUfunction *hfunc*, int *offset*, float *value*)

Deprecated

Sets a floating-point parameter that will be specified the next time the kernel corresponding to `hfunc` will be invoked. `offset` is a byte offset.

Parameters:

hfunc - Kernel to add parameter to

offset - Offset to add parameter to argument list

value - Value of parameter

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

5.42.2.7 CUresult cuParamSeti (CUfunction *hfunc*, int *offset*, unsigned int *value*)

Deprecated

Sets an integer parameter that will be specified the next time the kernel corresponding to `hfunc` will be invoked. `offset` is a byte offset.

Parameters:

hfunc - Kernel to add parameter to

offset - Offset to add parameter to argument list

value - Value of parameter

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

5.42.2.8 CUresult cuParamSetSize (CUfunction *hfunc*, unsigned int *numbytes*)

Deprecated

Sets through `numbytes` the total size in bytes needed by the function parameters of the kernel corresponding to `hfunc`.

Parameters:

hfunc - Kernel to set parameter size for

numbytes - Size of parameter list in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

5.42.2.9 CUresult cuParamSetTexRef (CUfunction *hfunc*, int *texunit*, CUtexref *hTexRef*)

Deprecated

Makes the CUDA array or linear memory bound to the texture reference `hTexRef` available to a device program as a texture. In this version of CUDA, the texture-reference must be obtained via cuModuleGetTexRef() and the `texunit` parameter must be set to CU_PARAM_TR_DEFAULT.

Parameters:

hfunc - Kernel to add texture-reference to

texunit - Texture unit (must be CU_PARAM_TR_DEFAULT)

hTexRef - Texture-reference to add to argument list

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

5.42.2.10 CUresult cuParamSetv (CUfunction *hfunc*, int *offset*, void * *ptr*, unsigned int *numbytes*)

Deprecated

Copies an arbitrary amount of data (specified in `numbytes`) from `ptr` into the parameter space of the kernel corresponding to `hfunc`. `offset` is a byte offset.

Parameters:

hfunc - Kernel to add data to

offset - Offset to add data to argument list

ptr - Pointer to arbitrary data

numbytes - Size of data to copy in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

5.43 Texture Reference Management

Functions

- CUresult cuTexRefGetAddress (CUdeviceptr *pdptr, CUtexref hTexRef)

  Gets the address associated with a texture reference.

- CUresult cuTexRefGetAddressMode (CUaddress_mode *pam, CUtexref hTexRef, int dim)

  Gets the addressing mode used by a texture reference.

- CUresult cuTexRefGetArray (CUarray *phArray, CUtexref hTexRef)

  Gets the array bound to a texture reference.

- CUresult cuTexRefGetFilterMode (CUfilter_mode *pfm, CUtexref hTexRef)

  Gets the filter-mode used by a texture reference.

- CUresult cuTexRefGetFlags (unsigned int *pFlags, CUtexref hTexRef)

  Gets the flags used by a texture reference.

- CUresult cuTexRefGetFormat (CUarray_format *pFormat, int *pNumChannels, CUtexref hTexRef)

  Gets the format used by a texture reference.

- CUresult cuTexRefGetMaxAnisotropy (int *pmaxAniso, CUtexref hTexRef)

  Gets the maximum anistropy for a texture reference.

- CUresult cuTexRefGetMipmapFilterMode (CUfilter_mode *pfm, CUtexref hTexRef)

  Gets the mipmap filtering mode for a texture reference.

- CUresult cuTexRefGetMipmapLevelBias (float *pbias, CUtexref hTexRef)

  Gets the mipmap level bias for a texture reference.

- CUresult cuTexRefGetMipmapLevelClamp (float *pminMipmapLevelClamp, float *pmaxMipmapLevelClamp, CUtexref hTexRef)

  Gets the min/max mipmap level clamps for a texture reference.

- CUresult cuTexRefGetMipmappedArray (CUmipmappedArray *phMipmappedArray, CUtexref hTexRef)

  Gets the mipmapped array bound to a texture reference.

- CUresult cuTexRefSetAddress (size_t *ByteOffset, CUtexref hTexRef, CUdeviceptr dptr, size_t bytes)

  Binds an address as a texture reference.

- CUresult cuTexRefSetAddress2D (CUtexref hTexRef, const CUDA_ARRAY_DESCRIPTOR *desc, CUdeviceptr dptr, size_t Pitch)

  Binds an address as a 2D texture reference.

- CUresult cuTexRefSetAddressMode (CUtexref hTexRef, int dim, CUaddress_mode am)

  Sets the addressing mode for a texture reference.

- CUresult cuTexRefSetArray (CUtexref hTexRef, CUarray hArray, unsigned int Flags)

  Binds an array as a texture reference.

- CUresult cuTexRefSetFilterMode (CUtexref hTexRef, CUfilter_mode fm)

  Sets the filtering mode for a texture reference.

- CUresult cuTexRefSetFlags (CUtexref hTexRef, unsigned int Flags)

  Sets the flags for a texture reference.

- CUresult cuTexRefSetFormat (CUtexref hTexRef, CUarray_format fmt, int NumPackedComponents)

  Sets the format for a texture reference.

- CUresult cuTexRefSetMaxAnisotropy (CUtexref hTexRef, unsigned int maxAniso)

  Sets the maximum anistropy for a texture reference.

- CUresult cuTexRefSetMipmapFilterMode (CUtexref hTexRef, CUfilter_mode fm)

  Sets the mipmap filtering mode for a texture reference.

- CUresult cuTexRefSetMipmapLevelBias (CUtexref hTexRef, float bias)

  Sets the mipmap level bias for a texture reference.

- CUresult cuTexRefSetMipmapLevelClamp (CUtexref hTexRef, float minMipmapLevelClamp, float maxMipmapLevelClamp)

  Sets the mipmap min/max mipmap level clamps for a texture reference.

- CUresult cuTexRefSetMipmappedArray (CUtexref hTexRef, CUmipmappedArray hMipmappedArray, unsigned int Flags)

  Binds a mipmapped array to a texture reference.

5.43.1 Detailed Description

This section describes the texture reference management functions of the low-level CUDA driver application programming interface.

5.43.2 Function Documentation

5.43.2.1 CUresult cuTexRefGetAddress (CUdeviceptr * *pdptr*, CUtexref *hTexRef*)

Returns in `*pdptr` the base address bound to the texture reference `hTexRef`, or returns CUDA_ERROR_INVALID_VALUE if the texture reference is not bound to any device memory range.

Parameters:

pdptr - Returned device address

hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.2 CUresult cuTexRefGetAddressMode (CUaddress_mode * *pam*, CUtexref *hTexRef*, int *dim*)

Returns in `*pam` the addressing mode corresponding to the dimension `dim` of the texture reference `hTexRef`. Currently, the only valid value for `dim` are 0 and 1.

Parameters:

pam - Returned addressing mode

hTexRef - Texture reference

dim - Dimension

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.3 CUresult cuTexRefGetArray (CUarray * *phArray*, CUtexref *hTexRef*)

Returns in `*phArray` the CUDA array bound to the texture reference `hTexRef`, or returns CUDA_ERROR_INVALID_VALUE if the texture reference is not bound to any CUDA array.

Parameters:

phArray - Returned array

hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.4 CUresult cuTexRefGetFilterMode (CUfilter_mode * *pfm*, CUtexref *hTexRef*)

Returns in `*pfm` the filtering mode of the texture reference `hTexRef`.

Parameters:

pfm - Returned filtering mode

hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.5 CUresult cuTexRefGetFlags (unsigned int * *pFlags*, CUtexref *hTexRef*)

Returns in `*pFlags` the flags of the texture reference `hTexRef`.

Parameters:

pFlags - Returned flags

hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFormat

5.43.2.6 CUresult cuTexRefGetFormat (CUarray_format * *pFormat*, int * *pNumChannels*, CUtexref *hTexRef*)

Returns in `*pFormat` and `*pNumChannels` the format and number of components of the CUDA array bound to the texture reference `hTexRef`. If `pFormat` or `pNumChannels` is NULL, it will be ignored.

Parameters:

pFormat - Returned format

pNumChannels - Returned number of components

hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags

5.43.2.7 CUresult cuTexRefGetMaxAnisotropy (int * *pmaxAniso*, CUtexref *hTexRef*)

Returns the maximum aniostropy in `pmaxAniso` that's used when reading memory through the texture reference `hTexRef`.

Parameters:

pmaxAniso - Returned maximum anisotropy

hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.8 CUresult cuTexRefGetMipmapFilterMode (CUfilter_mode * *pfm*, CUtexref *hTexRef*)

Returns the mipmap filtering mode in `pfm` that's used when reading memory through the texture reference `hTexRef`.

Parameters:

pfm - Returned mipmap filtering mode

hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.9 CUresult cuTexRefGetMipmapLevelBias (float * *pbias*, CUtexref *hTexRef*)

Returns the mipmap level bias in `pBias` that's added to the specified mipmap level when reading memory through the texture reference `hTexRef`.

Parameters:

pbias - Returned mipmap level bias

hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.10 CUresult cuTexRefGetMipmapLevelClamp (float * *pminMipmapLevelClamp*, float * *pmaxMipmapLevelClamp*, CUtexref *hTexRef*)

Returns the min/max mipmap level clamps in `pminMipmapLevelClamp` and `pmaxMipmapLevelClamp` that's used when reading memory through the texture reference `hTexRef`.

Parameters:

pminMipmapLevelClamp - Returned mipmap min level clamp

pmaxMipmapLevelClamp - Returned mipmap max level clamp

hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.11 CUresult cuTexRefGetMipmappedArray (CUmipmappedArray * *phMipmappedArray*, CUtexref *hTexRef*)

Returns in `*phMipmappedArray` the CUDA mipmapped array bound to the texture reference `hTexRef`, or returns CUDA_ERROR_INVALID_VALUE if the texture reference is not bound to any CUDA mipmapped array.

Parameters:

phMipmappedArray - Returned mipmapped array

hTexRef - Texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.12 CUresult cuTexRefSetAddress (size_t * *ByteOffset*, CUtexref *hTexRef*, CUdeviceptr *dptr*, size_t *bytes*)

Binds a linear address range to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to `hTexRef` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, cuTexRefSetAddress() passes back a byte offset in `*ByteOffset` that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the tex1Dfetch() function.

If the device memory pointer was returned from cuMemAlloc(), the offset is guaranteed to be 0 and NULL may be passed as the `ByteOffset` parameter.

The total number of elements (or texels) in the linear address range cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH. The number of elements is computed as (`bytes` / bytesPerElement), where bytesPerElement is determined from the data format and number of components set using cuTexRefSetFormat().

Parameters:

ByteOffset - Returned byte offset

hTexRef - Texture reference to bind

dptr - Device pointer to bind

bytes - Size of memory to bind in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.13 CUresult cuTexRefSetAddress2D (CUtexref *hTexRef*, const CUDA_ARRAY_DESCRIPTOR * *desc*, CUdeviceptr *dptr*, size_t *Pitch*)

Binds a linear address range to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to `hTexRef` is unbound.

Using a tex2D() function inside a kernel requires a call to either cuTexRefSetArray() to bind the corresponding texture reference to an array, or cuTexRefSetAddress2D() to bind the texture reference to linear memory.

Function calls to cuTexRefSetFormat() cannot follow calls to cuTexRefSetAddress2D() for the same texture reference.

It is required that `dptr` be aligned to the appropriate hardware-specific texture alignment. You can query this value using the device attribute CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT. If an unaligned `dptr` is supplied, CUDA_ERROR_INVALID_VALUE is returned.

`Pitch` has to be aligned to the hardware-specific texture pitch alignment. This value can be queried using the device attribute CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT. If an unaligned `Pitch` is supplied, CUDA_ERROR_INVALID_VALUE is returned.

Width and Height, which are specified in elements (or texels), cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH and CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT respectively. `Pitch`, which is specified in bytes, cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH.

Parameters:

hTexRef - Texture reference to bind

desc - Descriptor of CUDA array

dptr - Device pointer to bind

Pitch - Line pitch in bytes

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.14 CUresult cuTexRefSetAddressMode (CUtexref *hTexRef*, int *dim*, CUaddress_mode *am*)

Specifies the addressing mode `am` for the given dimension `dim` of the texture reference `hTexRef`. If `dim` is zero, the addressing mode is applied to the first parameter of the functions used to fetch from the texture; if `dim` is 1, the second, and so on. CUaddress_mode is defined as:

```
typedef enum CUaddress_mode_enum {
   CU_TR_ADDRESS_MODE_WRAP = 0,
   CU_TR_ADDRESS_MODE_CLAMP = 1,
   CU_TR_ADDRESS_MODE_MIRROR = 2,
   CU_TR_ADDRESS_MODE_BORDER = 3
} CUaddress_mode;
```

Note that this call has no effect if `hTexRef` is bound to linear memory. Also, if the flag, CU_TRSF_NORMALIZED_COORDINATES, is not set, the only supported address mode is CU_TR_ADDRESS_MODE_CLAMP.

Parameters:

hTexRef - Texture reference

dim - Dimension

am - Addressing mode to set

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.15 CUresult cuTexRefSetArray (CUtexref *hTexRef*, CUarray *hArray*, unsigned int *Flags*)

Binds the CUDA array `hArray` to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. `Flags` must be set to CU_TRSA_OVERRIDE_FORMAT. Any CUDA array previously bound to `hTexRef` is unbound.

Parameters:

hTexRef - Texture reference to bind

hArray - Array to bind

Flags - Options (must be CU_TRSA_OVERRIDE_FORMAT)

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.16 CUresult cuTexRefSetFilterMode (CUtexref *hTexRef*, CUfilter_mode *fm*)

Specifies the filtering mode `fm` to be used when reading memory through the texture reference `hTexRef`. CUfilter_mode_enum is defined as:

```
typedef enum CUfilter_mode_enum {
   CU_TR_FILTER_MODE_POINT = 0,
   CU_TR_FILTER_MODE_LINEAR = 1
} CUfilter_mode;
```

Note that this call has no effect if `hTexRef` is bound to linear memory.

Parameters:

hTexRef - Texture reference

fm - Filtering mode to set

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSet-Flags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGet-FilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.17 CUresult cuTexRefSetFlags (CUtexref *hTexRef*, unsigned int *Flags*)

Specifies optional flags via `Flags` to specify the behavior of data returned through the texture reference `hTexRef`. The valid flags are:

- CU_TRSF_READ_AS_INTEGER, which suppresses the default behavior of having the texture promote integer data to floating point data in the range [0, 1]. Note that texture with 32-bit integer format would not be promoted, regardless of whether or not this flag is specified;
- CU_TRSF_NORMALIZED_COORDINATES, which suppresses the default behavior of having the texture coordinates range from [0, Dim) where Dim is the width or height of the CUDA array. Instead, the texture coordinates [0, 1.0) reference the entire breadth of the array dimension;

Parameters:

hTexRef - Texture reference

Flags - Optional flags to set

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFil-terMode, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRe-fGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.18 CUresult cuTexRefSetFormat (CUtexref *hTexRef*, CUarray_format *fmt*, int *NumPackedComponents*)

Specifies the format of the data to be read by the texture reference `hTexRef`. `fmt` and `NumPackedComponents` are exactly analogous to the Format and NumChannels members of the CUDA_ARRAY_DESCRIPTOR structure: They specify the format of each component and the number of components per array element.

Parameters:

hTexRef - Texture reference

fmt - Format to set

NumPackedComponents - Number of components per array element

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFil-terMode, cuTexRefSetFlags, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRe-fGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.19 CUresult cuTexRefSetMaxAnisotropy (CUtexref *hTexRef*, unsigned int *maxAniso*)

Specifies the maximum aniostropy `maxAniso` to be used when reading memory through the texture reference `hTexRef`.

Note that this call has no effect if `hTexRef` is bound to linear memory.

Parameters:

hTexRef - Texture reference

maxAniso - Maximum anisotropy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSet-Flags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGet-FilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.20 CUresult cuTexRefSetMipmapFilterMode (CUtexref *hTexRef*, CUfilter_mode *fm*)

Specifies the mipmap filtering mode `fm` to be used when reading memory through the texture reference `hTexRef`. CUfilter_mode_enum is defined as:

```
typedef enum CUfilter_mode_enum {
   CU_TR_FILTER_MODE_POINT = 0,
   CU_TR_FILTER_MODE_LINEAR = 1
} CUfilter_mode;
```

Note that this call has no effect if `hTexRef` is not bound to a mipmapped array.

Parameters:

hTexRef - Texture reference

fm - Filtering mode to set

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.21 CUresult cuTexRefSetMipmapLevelBias (CUtexref *hTexRef*, float *bias*)

Specifies the mipmap level bias `bias` to be added to the specified mipmap level when reading memory through the texture reference `hTexRef`.

Note that this call has no effect if `hTexRef` is not bound to a mipmapped array.

Parameters:

hTexRef - Texture reference

bias - Mipmap level bias

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.22 CUresult cuTexRefSetMipmapLevelClamp (CUtexref *hTexRef*, float *minMipmapLevelClamp*, float *maxMipmapLevelClamp*)

Specifies the min/max mipmap level clamps, `minMipmapLevelClamp` and `maxMipmapLevelClamp` respectively, to be used when reading memory through the texture reference `hTexRef`.

Note that this call has no effect if `hTexRef` is not bound to a mipmapped array.

Parameters:

hTexRef - Texture reference

minMipmapLevelClamp - Mipmap min level clamp

maxMipmapLevelClamp - Mipmap max level clamp

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.43.2.23 CUresult cuTexRefSetMipmappedArray (CUtexref *hTexRef*, CUmipmappedArray *hMipmappedArray*, unsigned int *Flags*)

Binds the CUDA mipmapped array `hMipmappedArray` to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. `Flags` must be set to CU_TRSA_OVERRIDE_FORMAT. Any CUDA array previously bound to `hTexRef` is unbound.

Parameters:

hTexRef - Texture reference to bind

hMipmappedArray - Mipmapped array to bind

Flags - Options (must be CU_TRSA_OVERRIDE_FORMAT)

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

5.44 Texture Reference Management [DEPRECATED]

Functions

- CUresult cuTexRefCreate (CUtexref *pTexRef)

  Creates a texture reference.

- CUresult cuTexRefDestroy (CUtexref hTexRef)

  Destroys a texture reference.

5.44.1 Detailed Description

This section describes the deprecated texture reference management functions of the low-level CUDA driver application programming interface.

5.44.2 Function Documentation

5.44.2.1 CUresult cuTexRefCreate (CUtexref * *pTexRef*)

Deprecated

Creates a texture reference and returns its handle in `*pTexRef`. Once created, the application must call cuTexRefSetArray() or cuTexRefSetAddress() to associate the reference with allocated memory. Other texture reference functions are used to specify the format and interpretation (addressing, filtering, etc.) to be used when the memory is read through this texture reference.

Parameters:

pTexRef - Returned texture reference

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefDestroy

5.44.2.2 CUresult cuTexRefDestroy (CUtexref *hTexRef*)

Deprecated

Destroys the texture reference specified by `hTexRef`.

Parameters:

hTexRef - Texture reference to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexRefCreate

5.45 Surface Reference Management

Functions

- CUresult cuSurfRefGetArray (CUarray *phArray, CUsurfref hSurfRef)

  Passes back the CUDA array bound to a surface reference.

- CUresult cuSurfRefSetArray (CUsurfref hSurfRef, CUarray hArray, unsigned int Flags)

  Sets the CUDA array for a surface reference.

5.45.1 Detailed Description

This section describes the surface reference management functions of the low-level CUDA driver application programming interface.

5.45.2 Function Documentation

5.45.2.1 CUresult cuSurfRefGetArray (CUarray * *phArray*, CUsurfref *hSurfRef*)

Returns in `*phArray` the CUDA array bound to the surface reference `hSurfRef`, or returns CUDA_ERROR_INVALID_VALUE if the surface reference is not bound to any CUDA array.

Parameters:

phArray - Surface reference handle

hSurfRef - Surface reference handle

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuModuleGetSurfRef, cuSurfRefSetArray

5.45.2.2 CUresult cuSurfRefSetArray (CUsurfref *hSurfRef*, CUarray *hArray*, unsigned int *Flags*)

Sets the CUDA array `hArray` to be read and written by the surface reference `hSurfRef`. Any previous CUDA array state associated with the surface reference is superseded by this function. `Flags` must be set to 0. The CUDA_ARRAY3D_SURFACE_LDST flag must have been set for the CUDA array. Any CUDA array previously bound to `hSurfRef` is unbound.

Parameters:

hSurfRef - Surface reference handle

hArray - CUDA array handle

Flags - set to 0

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuModuleGetSurfRef, cuSurfRefGetArray

5.46 Texture Object Management

Functions

- CUresult cuTexObjectCreate (CUtexObject *pTexObject, const CUDA_RESOURCE_DESC *pResDesc, const CUDA_TEXTURE_DESC *pTexDesc, const CUDA_RESOURCE_VIEW_DESC *pResViewDesc)

  Creates a texture object.

- CUresult cuTexObjectDestroy (CUtexObject texObject)

  Destroys a texture object.

- CUresult cuTexObjectGetResourceDesc (CUDA_RESOURCE_DESC *pResDesc, CUtexObject texObject)

  Returns a texture object's resource descriptor.

- CUresult cuTexObjectGetResourceViewDesc (CUDA_RESOURCE_VIEW_DESC *pResViewDesc, CUtexObject texObject)

  Returns a texture object's resource view descriptor.

- CUresult cuTexObjectGetTextureDesc (CUDA_TEXTURE_DESC *pTexDesc, CUtexObject texObject)

  Returns a texture object's texture descriptor.

5.46.1 Detailed Description

This section describes the texture object management functions of the low-level CUDA driver application programming interface. The texture object API is only supported on devices of compute capability 3.0 or higher.

5.46.2 Function Documentation

5.46.2.1 CUresult cuTexObjectCreate (CUtexObject * *pTexObject*, const CUDA_RESOURCE_DESC * *pResDesc*, const CUDA_TEXTURE_DESC * *pTexDesc*, const CUDA_RESOURCE_VIEW_DESC * *pResViewDesc*)

Creates a texture object and returns it in `pTexObject`. `pResDesc` describes the data to texture from. `pTexDesc` describes how the data should be sampled. `pResViewDesc` is an optional argument that specifies an alternate format for the data described by `pResDesc`, and also describes the subresource region to restrict access to when texturing. `pResViewDesc` can only be specified if the type of resource is a CUDA array or a CUDA mipmapped array.

Texture objects are only supported on devices of compute capability 3.0 or higher.

The CUDA_RESOURCE_DESC structure is defined as:

```
typedef struct CUDA_RESOURCE_DESC_st
{
    CUresourcetype resType;

    union {
        struct {
            CUarray hArray;
        } array;
        struct {
            CUmipmappedArray hMipmappedArray;
        } mipmap;
        struct {
```

```
            CUdeviceptr devPtr;
            CUarray_format format;
            unsigned int numChannels;
            size_t sizeInBytes;
        } linear;
        struct {
            CUdeviceptr devPtr;
            CUarray_format format;
            unsigned int numChannels;
            size_t width;
            size_t height;
            size_t pitchInBytes;
        } pitch2D;
    } res;

    unsigned int flags;
} CUDA_RESOURCE_DESC;
```

where:

- CUDA_RESOURCE_DESC::resType specifies the type of resource to texture from. CUresourceType is defined as:

```
typedef enum CUresourcetype_enum {
    CU_RESOURCE_TYPE_ARRAY           = 0x00,
    CU_RESOURCE_TYPE_MIPMAPPED_ARRAY = 0x01,
    CU_RESOURCE_TYPE_LINEAR          = 0x02,
    CU_RESOURCE_TYPE_PITCH2D         = 0x03
} CUresourcetype;
```

If CUDA_RESOURCE_DESC::resType is set to CU_RESOURCE_TYPE_ARRAY, CUDA_RESOURCE_DESC::res::array::hArray must be set to a valid CUDA array handle.

If CUDA_RESOURCE_DESC::resType is set to CU_RESOURCE_TYPE_MIPMAPPED_ARRAY, CUDA_RESOURCE_DESC::res::mipmap::hMipmappedArray must be set to a valid CUDA mipmapped array handle.

If CUDA_RESOURCE_DESC::resType is set to CU_RESOURCE_TYPE_LINEAR, CUDA_RESOURCE_DESC::res::linear::devPtr must be set to a valid device pointer, that is aligned to CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT. CUDA_RESOURCE_DESC::res::linear::format and CUDA_RESOURCE_DESC::res::linear::numChannels describe the format of each component and the number of components per array element. CUDA_RESOURCE_DESC::res::linear::sizeInBytes specifies the size of the array in bytes. The total number of elements in the linear address range cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH. The number of elements is computed as (sizeInBytes / (sizeof(format) $*$ numChannels)).

If CUDA_RESOURCE_DESC::resType is set to CU_RESOURCE_TYPE_PITCH2D, CUDA_RESOURCE_DESC::res::pitch2D::devPtr must be set to a valid device pointer, that is aligned to CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT. CUDA_RESOURCE_DESC::res::pitch2D::format and CUDA_RESOURCE_DESC::res::pitch2D::numChannels describe the format of each component and the number of components per array element. CUDA_RESOURCE_DESC::res::pitch2D::width and CUDA_RESOURCE_DESC::res::pitch2D::height specify the width and height of the array in elements, and cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH and CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT respectively. CUDA_RESOURCE_DESC::res::pitch2D::pitchInBytes specifies the pitch between two rows in bytes and has to be aligned to CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT. Pitch cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH.

- flags must be set to zero.

The CUDA_TEXTURE_DESC struct is defined as

```
typedef struct CUDA_TEXTURE_DESC_st {
    CUaddress_mode addressMode[3];
    CUfilter_mode filterMode;
    unsigned int flags;
    unsigned int maxAnisotropy;
    CUfilter_mode mipmapFilterMode;
    float mipmapLevelBias;
    float minMipmapLevelClamp;
    float maxMipmapLevelClamp;
} CUDA_TEXTURE_DESC;
```

where

- CUDA_TEXTURE_DESC::addressMode specifies the addressing mode for each dimension of the texture data. CUaddress_mode is defined as:

  ```
  typedef enum CUaddress_mode_enum {
      CU_TR_ADDRESS_MODE_WRAP = 0,
      CU_TR_ADDRESS_MODE_CLAMP = 1,
      CU_TR_ADDRESS_MODE_MIRROR = 2,
      CU_TR_ADDRESS_MODE_BORDER = 3
  } CUaddress_mode;
  ```

  This is ignored if CUDA_RESOURCE_DESC::resType is CU_RESOURCE_TYPE_LINEAR. Also, if the flag, CU_TRSF_NORMALIZED_COORDINATES is not set, the only supported address mode is CU_TR_-ADDRESS_MODE_CLAMP.

- CUDA_TEXTURE_DESC::filterMode specifies the filtering mode to be used when fetching from the texture. CUfilter_mode is defined as:

  ```
  typedef enum CUfilter_mode_enum {
      CU_TR_FILTER_MODE_POINT = 0,
      CU_TR_FILTER_MODE_LINEAR = 1
  } CUfilter_mode;
  ```

  This is ignored if CUDA_RESOURCE_DESC::resType is CU_RESOURCE_TYPE_LINEAR.

- CUDA_TEXTURE_DESC::flags can be any combination of the following:
  - CU_TRSF_READ_AS_INTEGER, which suppresses the default behavior of having the texture promote integer data to floating point data in the range [0, 1]. Note that texture with 32-bit integer format would not be promoted, regardless of whether or not this flag is specified.
  - CU_TRSF_NORMALIZED_COORDINATES, which suppresses the default behavior of having the texture coordinates range from [0, Dim) where Dim is the width or height of the CUDA array. Instead, the texture coordinates [0, 1.0) reference the entire breadth of the array dimension; Note that for CUDA mipmapped arrays, this flag has to be set.

- CUDA_TEXTURE_DESC::maxAnisotropy specifies the maximum anistropy ratio to be used when doing anisotropic filtering. This value will be clamped to the range [1,16].

- CUDA_TEXTURE_DESC::mipmapFilterMode specifies the filter mode when the calculated mipmap level lies between two defined mipmap levels.

- CUDA_TEXTURE_DESC::mipmapLevelBias specifies the offset to be applied to the calculated mipmap level.

- CUDA_TEXTURE_DESC::minMipmapLevelClamp specifies the lower end of the mipmap level range to clamp access to.

- CUDA_TEXTURE_DESC::maxMipmapLevelClamp specifies the upper end of the mipmap level range to clamp access to.

The CUDA_RESOURCE_VIEW_DESC struct is defined as

```
typedef struct CUDA_RESOURCE_VIEW_DESC_st
{
    CUresourceViewFormat format;
    size_t width;
    size_t height;
    size_t depth;
    unsigned int firstMipmapLevel;
    unsigned int lastMipmapLevel;
    unsigned int firstLayer;
    unsigned int lastLayer;
} CUDA_RESOURCE_VIEW_DESC;
```

where:

- CUDA_RESOURCE_VIEW_DESC::format specifies how the data contained in the CUDA array or CUDA mipmapped array should be interpreted. Note that this can incur a change in size of the texture data. If the resource view format is a block compressed format, then the underlying CUDA array or CUDA mipmapped array has to have a base of format CU_AD_FORMAT_UNSIGNED_INT32. with 2 or 4 channels, depending on the block compressed format. For ex., BC1 and BC4 require the underlying CUDA array to have a format of CU_AD_FORMAT_UNSIGNED_INT32 with 2 channels. The other BC formats require the underlying resource to have the same base format but with 4 channels.

- CUDA_RESOURCE_VIEW_DESC::width specifies the new width of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original width of the resource. For non block compressed formats, this value has to be equal to that of the original resource.

- CUDA_RESOURCE_VIEW_DESC::height specifies the new height of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original height of the resource. For non block compressed formats, this value has to be equal to that of the original resource.

- CUDA_RESOURCE_VIEW_DESC::depth specifies the new depth of the texture data. This value has to be equal to that of the original resource.

- CUDA_RESOURCE_VIEW_DESC::firstMipmapLevel specifies the most detailed mipmap level. This will be the new mipmap level zero. For non-mipmapped resources, this value has to be zero.CUDA_TEXTURE_-DESC::minMipmapLevelClamp and CUDA_TEXTURE_DESC::maxMipmapLevelClamp will be relative to this value. For ex., if the firstMipmapLevel is set to 2, and a minMipmapLevelClamp of 1.2 is specified, then the actual minimum mipmap level clamp will be 3.2.

- CUDA_RESOURCE_VIEW_DESC::lastMipmapLevel specifies the least detailed mipmap level. For non-mipmapped resources, this value has to be zero.

- CUDA_RESOURCE_VIEW_DESC::firstLayer specifies the first layer index for layered textures. This will be the new layer zero. For non-layered resources, this value has to be zero.

- CUDA_RESOURCE_VIEW_DESC::lastLayer specifies the last layer index for layered textures. For non-layered resources, this value has to be zero.

Parameters:

pTexObject - Texture object to create

pResDesc - Resource descriptor

pTexDesc - Texture descriptor

pResViewDesc - Resource view descriptor

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexObjectDestroy

5.46.2.2 CUresult cuTexObjectDestroy (CUtexObject *texObject*)

Destroys the texture object specified by `texObject`.

Parameters:

texObject - Texture object to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexObjectCreate

5.46.2.3 CUresult cuTexObjectGetResourceDesc (CUDA_RESOURCE_DESC * *pResDesc*, CUtexObject *texObject*)

Returns the resource descriptor for the texture object specified by `texObject`.

Parameters:

pResDesc - Resource descriptor

texObject - Texture object

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexObjectCreate

5.46.2.4 CUresult cuTexObjectGetResourceViewDesc (CUDA_RESOURCE_VIEW_DESC * *pResViewDesc*, CUtexObject *texObject*)

Returns the resource view descriptor for the texture object specified by `texObject`. If no resource view was set for `texObject`, the CUDA_ERROR_INVALID_VALUE is returned.

Parameters:

pResViewDesc - Resource view descriptor

texObject - Texture object

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexObjectCreate

5.46.2.5 CUresult cuTexObjectGetTextureDesc (CUDA_TEXTURE_DESC * *pTexDesc*, CUtexObject *texObject*)

Returns the texture descriptor for the texture object specified by `texObject`.

Parameters:

pTexDesc - Texture descriptor

texObject - Texture object

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuTexObjectCreate

5.47 Surface Object Management

Functions

- CUresult cuSurfObjectCreate (CUsurfObject *pSurfObject, const CUDA_RESOURCE_DESC *pResDesc)
  Creates a surface object.
- CUresult cuSurfObjectDestroy (CUsurfObject surfObject)
  Destroys a surface object.
- CUresult cuSurfObjectGetResourceDesc (CUDA_RESOURCE_DESC *pResDesc, CUsurfObject surfObject)
  Returns a surface object's resource descriptor.

5.47.1 Detailed Description

This section describes the surface object management functions of the low-level CUDA driver application programming interface. The surface object API is only supported on devices of compute capability 3.0 or higher.

5.47.2 Function Documentation

5.47.2.1 CUresult cuSurfObjectCreate (CUsurfObject * *pSurfObject*, const CUDA_RESOURCE_DESC * *pResDesc*)

Creates a surface object and returns it in `pSurfObject`. `pResDesc` describes the data to perform surface load/stores on. CUDA_RESOURCE_DESC::resType must be CU_RESOURCE_TYPE_ARRAY and CUDA_RESOURCE_DESC::res::array::hArray must be set to a valid CUDA array handle. CUDA_RESOURCE_DESC::flags must be set to zero.

Surface objects are only supported on devices of compute capability 3.0 or higher.

Parameters:

pSurfObject - Surface object to create

pResDesc - Resource descriptor

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuSurfObjectDestroy

5.47.2.2 CUresult cuSurfObjectDestroy (CUsurfObject *surfObject*)

Destroys the surface object specified by `surfObject`.

Parameters:

surfObject - Surface object to destroy

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuSurfObjectCreate

5.47.2.3 CUresult cuSurfObjectGetResourceDesc (CUDA_RESOURCE_DESC * *pResDesc*, CUsurfObject *surfObject*)

Returns the resource descriptor for the surface object specified by `surfObject`.

Parameters:

pResDesc - Resource descriptor

surfObject - Surface object

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

See also:

cuSurfObjectCreate

5.48 Peer Context Memory Access

Functions

- CUresult cuCtxDisablePeerAccess (CUcontext peerContext)

  Disables direct access to memory allocations in a peer context and unregisters any registered allocations.

- CUresult cuCtxEnablePeerAccess (CUcontext peerContext, unsigned int Flags)

  Enables direct access to memory allocations in a peer context.

- CUresult cuDeviceCanAccessPeer (int ∗canAccessPeer, CUdevice dev, CUdevice peerDev)

  Queries if a device may directly access a peer device's memory.

5.48.1 Detailed Description

This section describes the direct peer context memory access functions of the low-level CUDA driver application programming interface.

5.48.2 Function Documentation

5.48.2.1 CUresult cuCtxDisablePeerAccess (CUcontext *peerContext*)

Returns CUDA_ERROR_PEER_ACCESS_NOT_ENABLED if direct peer access has not yet been enabled from `peerContext` to the current context.

Returns CUDA_ERROR_INVALID_CONTEXT if there is no current context, or if `peerContext` is not a valid context.

Parameters:

peerContext - Peer context to disable direct access to

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_PEER_ACCESS_NOT_ENABLED, CUDA_ERROR_INVALID_CONTEXT,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuDeviceCanAccessPeer, cuCtxEnablePeerAccess

5.48.2.2 CUresult cuCtxEnablePeerAccess (CUcontext *peerContext*, unsigned int *Flags*)

If both the current context and `peerContext` are on devices which support unified addressing (as may be queried using CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING) and same major compute capability, then on success all allocations from `peerContext` will immediately be accessible by the current context. See Unified Addressing for additional details.

Note that access granted by this call is unidirectional and that in order to access memory from the current context in `peerContext`, a separate symmetric call to cuCtxEnablePeerAccess() is required.

Returns CUDA_ERROR_PEER_ACCESS_UNSUPPORTED if cuDeviceCanAccessPeer() indicates that the CUdevice of the current context cannot directly access memory from the CUdevice of `peerContext`.

Returns CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED if direct access of `peerContext` from the current context has already been enabled.

Returns CUDA_ERROR_TOO_MANY_PEERS if direct peer access is not possible because hardware resources required for peer access have been exhausted.

Returns CUDA_ERROR_INVALID_CONTEXT if there is no current context, `peerContext` is not a valid context, or if the current context is `peerContext`.

Returns CUDA_ERROR_INVALID_VALUE if `Flags` is not 0.

Parameters:

peerContext - Peer context to enable direct access to from the current context

Flags - Reserved for future use and must be set to 0

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED, CUDA_ERROR_TOO_MANY_PEERS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_PEER_ACCESS_UNSUPPORTED, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuDeviceCanAccessPeer, cuCtxDisablePeerAccess

5.48.2.3 CUresult cuDeviceCanAccessPeer (int * *canAccessPeer*, CUdevice *dev*, CUdevice *peerDev*)

Returns in `*canAccessPeer` a value of 1 if contexts on `dev` are capable of directly accessing memory from contexts on `peerDev` and 0 otherwise. If direct access of `peerDev` from `dev` is possible, then access may be enabled on two specific contexts by calling cuCtxEnablePeerAccess().

Parameters:

canAccessPeer - Returned access capability

dev - Device from which allocations on `peerDev` are to be directly accessed.

peerDev - Device on which the allocations to be directly accessed by `dev` reside.

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_DEVICE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxEnablePeerAccess, cuCtxDisablePeerAccess

5.49 Graphics Interoperability

Functions

- CUresult cuGraphicsMapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)

  Map graphics resources for access by CUDA.

- CUresult cuGraphicsResourceGetMappedMipmappedArray (CUmipmappedArray *pMipmappedArray, CUgraphicsResource resource)

  Get a mipmapped array through which to access a mapped graphics resource.

- CUresult cuGraphicsResourceGetMappedPointer (CUdeviceptr *pDevPtr, size_t *pSize, CUgraphicsResource resource)

  Get a device pointer through which to access a mapped graphics resource.

- CUresult cuGraphicsResourceSetMapFlags (CUgraphicsResource resource, unsigned int flags)

  Set usage flags for mapping a graphics resource.

- CUresult cuGraphicsSubResourceGetMappedArray (CUarray *pArray, CUgraphicsResource resource, unsigned int arrayIndex, unsigned int mipLevel)

  Get an array through which to access a subresource of a mapped graphics resource.

- CUresult cuGraphicsUnmapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)

  Unmap graphics resources.

- CUresult cuGraphicsUnregisterResource (CUgraphicsResource resource)

  Unregisters a graphics resource for access by CUDA.

5.49.1 Detailed Description

This section describes the graphics interoperability functions of the low-level CUDA driver application programming interface.

5.49.2 Function Documentation

5.49.2.1 CUresult cuGraphicsMapResources (unsigned int *count*, CUgraphicsResource * *resources*, CUstream *hStream*)

Maps the `count` graphics resources in `resources` for access by CUDA.

The resources in `resources` may be accessed by CUDA until they are unmapped. The graphics API from which `resources` were registered should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any graphics calls issued before cuGraphicsMapResources() will complete before any subsequent CUDA work issued in `stream` begins.

If `resources` includes any duplicate entries then CUDA_ERROR_INVALID_HANDLE is returned. If any of `resources` are presently mapped for access by CUDA then CUDA_ERROR_ALREADY_MAPPED is returned.

Parameters:

count - Number of resources to map

resources - Resources to map for CUDA usage

hStream - Stream with which to synchronize

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_-MAPPED, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceGetMappedPointer cuGraphicsSubResourceGetMappedArray cuGraphicsUnmapResources

5.49.2.2 CUresult cuGraphicsResourceGetMappedMipmappedArray (CUmipmappedArray * *pMipmappedArray*, CUgraphicsResource *resource*)

Returns in `*pMipmappedArray` a mipmapped array through which the mapped graphics resource `resource`. The value set in `*pMipmappedArray` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via a mipmapped array and CUDA_ERROR_NOT_-MAPPED_AS_ARRAY is returned. If `resource` is not mapped then CUDA_ERROR_NOT_MAPPED is returned.

Parameters:

pMipmappedArray - Returned mipmapped array through which `resource` may be accessed

resource - Mapped resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED CUDA_ERROR_NOT_MAPPED_AS_ARRAY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceGetMappedPointer

5.49.2.3 CUresult cuGraphicsResourceGetMappedPointer (CUdeviceptr * *pDevPtr*, size_t * *pSize*, CUgraphicsResource *resource*)

Returns in `*pDevPtr` a pointer through which the mapped graphics resource `resource` may be accessed. Returns in `pSize` the size of the memory in bytes which may be accessed from that pointer. The value set in `pPointer` may change every time that `resource` is mapped.

If `resource` is not a buffer then it cannot be accessed via a pointer and CUDA_ERROR_NOT_MAPPED_AS_-POINTER is returned. If `resource` is not mapped then CUDA_ERROR_NOT_MAPPED is returned. *

Parameters:

pDevPtr - Returned pointer through which `resource` may be accessed

pSize - Returned size of the buffer accessible starting at `*pPointer`

resource - Mapped resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED CUDA_ERROR_NOT_MAPPED_AS_POINTER

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources, cuGraphicsSubResourceGetMappedArray

5.49.2.4 CUresult cuGraphicsResourceSetMapFlags (CUgraphicsResource *resource*, unsigned int *flags*)

Set `flags` for mapping the graphics resource `resource`.

Changes to `flags` will take effect the next time `resource` is mapped. The `flags` argument may be any of the following:

- CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- CU_GRAPHICS_MAP_RESOURCE_FLAGS_READONLY: Specifies that CUDA kernels which access this resource will not write to this resource.
- CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITEDISCARD: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `resource` is presently mapped for access by CUDA then CUDA_ERROR_ALREADY_MAPPED is returned. If `flags` is not one of the above values then CUDA_ERROR_INVALID_VALUE is returned.

Parameters:

resource - Registered resource to set flags for

flags - Parameters for resource mapping

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

5.49.2.5 CUresult cuGraphicsSubResourceGetMappedArray (CUarray * *pArray*, CUgraphicsResource *resource*, unsigned int *arrayIndex*, unsigned int *mipLevel*)

Returns in `*pArray` an array through which the subresource of the mapped graphics resource `resource` which corresponds to array index `arrayIndex` and mipmap level `mipLevel` may be accessed. The value set in `*pArray` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and CUDA_ERROR_NOT_MAPPED_AS_ARRAY is returned. If `arrayIndex` is not a valid array index for `resource` then CUDA_ERROR_INVALID_VALUE is returned. If `mipLevel` is not a valid mipmap level for `resource` then CUDA_ERROR_INVALID_VALUE is returned. If `resource` is not mapped then CUDA_ERROR_NOT_MAPPED is returned.

Parameters:

pArray - Returned array through which a subresource of `resource` may be accessed

resource - Mapped resource to access

arrayIndex - Array index for array textures or cubemap face index as defined by CUarray_cubemap_face for cubemap textures for the subresource to access

mipLevel - Mipmap level for the subresource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED CUDA_ERROR_NOT_MAPPED_AS_ARRAY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceGetMappedPointer

5.49.2.6 CUresult cuGraphicsUnmapResources (unsigned int *count*, CUgraphicsResource * *resources*, CUstream *hStream*)

Unmaps the `count` graphics resources in `resources`.

Once unmapped, the resources in `resources` may not be accessed by CUDA until they are mapped again.

This function provides the synchronization guarantee that any CUDA work issued in `stream` before cuGraphicsUnmapResources() will complete before any subsequently issued graphics work begins.

If `resources` includes any duplicate entries then CUDA_ERROR_INVALID_HANDLE is returned. If any of `resources` are not presently mapped for access by CUDA then CUDA_ERROR_NOT_MAPPED is returned.

Parameters:

count - Number of resources to unmap

resources - Resources to unmap

hStream - Stream with which to synchronize

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

5.49.2.7 CUresult cuGraphicsUnregisterResource (CUgraphicsResource *resource*)

Unregisters the graphics resource `resource` so it is not accessible by CUDA unless registered again.

If `resource` is invalid then CUDA_ERROR_INVALID_HANDLE is returned.

Parameters:

resource - Resource to unregister

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsD3D9RegisterResource, cuGraphicsD3D10RegisterResource, cuGraphicsD3D11RegisterResource, cuGraphicsGLRegisterBuffer, cuGraphicsGLRegisterImage

5.50 Profiler Control

Functions

- CUresult cuProfilerInitialize (const char ∗configFile, const char ∗outputFile, CUoutput_mode outputMode)

  Initialize the profiling.

- CUresult cuProfilerStart (void)

  Enable profiling.

- CUresult cuProfilerStop (void)

  Disable profiling.

5.50.1 Detailed Description

This section describes the profiler control functions of the low-level CUDA driver application programming interface.

5.50.2 Function Documentation

5.50.2.1 CUresult cuProfilerInitialize (const char ∗ *configFile*, const char ∗ *outputFile*, CUoutput_mode *outputMode*)

Using this API user can initialize the CUDA profiler by specifying the configuration file, output file and output file format. This API is generally used to profile different set of counters by looping the kernel launch. The `configFile` parameter can be used to select profiling options including profiler counters. Refer to the "Compute Command Line Profiler User Guide" for supported profiler options and counters.

Limitation: The CUDA profiler cannot be initialized with this API if another profiling tool is already active, as indicated by the CUDA_ERROR_PROFILER_DISABLED return code.

Typical usage of the profiling APIs is as follows:

for each set of counters/options

{

cuProfilerInitialize(); //Initialize profiling, set the counters or options in the config file

...

cuProfilerStart();

// code to be profiled

cuProfilerStop();

...

cuProfilerStart();

// code to be profiled

cuProfilerStop();

...

}

Parameters:

configFile - Name of the config file that lists the counters/options for profiling.

outputFile - Name of the outputFile where the profiling results will be stored.

outputMode - outputMode, can be CU_OUT_KEY_VALUE_PAIR or CU_OUT_CSV.

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_-ERROR_PROFILER_DISABLED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuProfilerStart, cuProfilerStop

5.50.2.2 CUresult cuProfilerStart (void)

Enables profile collection by the active profiling tool. If profiling is already enabled, then cuProfilerStart() has no effect.

cuProfilerStart and cuProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuProfilerInitialize, cuProfilerStop

5.50.2.3 CUresult cuProfilerStop (void)

Disables profile collection by the active profiling tool. If profiling is already disabled, then cuProfilerStop() has no effect.

cuProfilerStart and cuProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuProfilerInitialize, cuProfilerStart

5.51 OpenGL Interoperability

Modules

- OpenGL Interoperability [DEPRECATED]

Typedefs

- typedef enum CUGLDeviceList_enum CUGLDeviceList

Enumerations

- enum CUGLDeviceList_enum {
  CU_GL_DEVICE_LIST_ALL = 0x01,
  CU_GL_DEVICE_LIST_CURRENT_FRAME = 0x02,
  CU_GL_DEVICE_LIST_NEXT_FRAME = 0x03 }

Functions

- CUresult cuGLGetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, CUGLDeviceList deviceList)

  Gets the CUDA devices associated with the current OpenGL context.

- CUresult cuGraphicsGLRegisterBuffer (CUgraphicsResource *pCudaResource, GLuint buffer, unsigned int Flags)

  Registers an OpenGL buffer object.

- CUresult cuGraphicsGLRegisterImage (CUgraphicsResource *pCudaResource, GLuint image, GLenum target, unsigned int Flags)

  Register an OpenGL texture or renderbuffer object.

- CUresult cuWGLGetDevice (CUdevice *pDevice, HGPUNV hGpu)

  Gets the CUDA device associated with hGpu.

5.51.1 Detailed Description

This section describes the OpenGL interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of OpenGL resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interopability.

5.51.2 Typedef Documentation

5.51.2.1 typedef enum CUGLDeviceList_enum CUGLDeviceList

CUDA devices corresponding to an OpenGL device

5.51.3 Enumeration Type Documentation

5.51.3.1 enum CUGLDeviceList_enum

CUDA devices corresponding to an OpenGL device

Enumerator:

CU_GL_DEVICE_LIST_ALL The CUDA devices for all GPUs used by the current OpenGL context

CU_GL_DEVICE_LIST_CURRENT_FRAME The CUDA devices for the GPUs used by the current OpenGL context in its currently rendering frame

CU_GL_DEVICE_LIST_NEXT_FRAME The CUDA devices for the GPUs to be used by the current OpenGL context in the next frame

5.51.4 Function Documentation

5.51.4.1 CUresult cuGLGetDevices (unsigned int * *pCudaDeviceCount*, CUdevice * *pCudaDevices*, unsigned int *cudaDeviceCount*, CUGLDeviceList *deviceList*)

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the current OpenGL context. Also returns in `*pCudaDevices` at most cudaDeviceCount of the CUDA-compatible devices corresponding to the current OpenGL context. If any of the GPUs being used by the current OpenGL context are not CUDA capable then the call will return CUDA_ERROR_NO_DEVICE.

The `deviceList` argument may be any of the following:

- CU_GL_DEVICE_LIST_ALL: Query all devices used by the current OpenGL context.
- CU_GL_DEVICE_LIST_CURRENT_FRAME: Query the devices used by the current OpenGL context to render the current frame (in SLI).
- CU_GL_DEVICE_LIST_NEXT_FRAME: Query the devices used by the current OpenGL context to render the next frame (in SLI). Note that this is a prediction, it can't be guaranteed that this is correct in all cases.

Parameters:

pCudaDeviceCount - Returned number of CUDA devices.

pCudaDevices - Returned CUDA devices.

cudaDeviceCount - The size of the output device array pCudaDevices.

deviceList - The set of devices to return.

Returns:

CUDA_SUCCESS, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_INVALID_VALUE CUDA_ERROR_-INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuWGLGetDevice

5.51.4.2 CUresult cuGraphicsGLRegisterBuffer (CUgraphicsResource * *pCudaResource*, GLuint *buffer*, unsigned int *Flags*)

Registers the buffer object specified by `buffer` for access by CUDA. A handle to the registered object is returned as `pCudaResource`. The register flags `Flags` specify the intended usage, as follows:

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.
- CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

Parameters:

pCudaResource - Pointer to the returned object handle

buffer - name of buffer object to be registered

Flags - Register flags

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnregisterResource, cuGraphicsMapResources, cuGraphicsResourceGetMappedPointer

5.51.4.3 CUresult cuGraphicsGLRegisterImage (CUgraphicsResource * *pCudaResource*, GLuint *image*, GLenum *target*, unsigned int *Flags*)

Registers the texture or renderbuffer object specified by `image` for access by CUDA. A handle to the registered object is returned as `pCudaResource`.

`target` must match the type of the object, and must be one of GL_TEXTURE_2D, GL_TEXTURE_RECTANGLE, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_3D, GL_TEXTURE_2D_ARRAY, or GL_RENDERBUFFER.

The register flags `Flags` specify the intended usage, as follows:

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.
- CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.
- CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.

- CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER: Specifies that CUDA will perform texture gather operations on this resource.

The following image formats are supported. For brevity's sake, the list is abbreviated. For ex., {GL_R, GL_RG} X {8, 16} would expand to the following 4 formats {GL_R8, GL_R16, GL_RG8, GL_RG16} :

- GL_RED, GL_RG, GL_RGBA, GL_LUMINANCE, GL_ALPHA, GL_LUMINANCE_ALPHA, GL_INTENSITY
- {GL_R, GL_RG, GL_RGBA} X {8, 16, 16F, 32F, 8UI, 16UI, 32UI, 8I, 16I, 32I}
- {GL_LUMINANCE, GL_ALPHA, GL_LUMINANCE_ALPHA, GL_INTENSITY} X {8, 16, 16F_ARB, 32F_ARB, 8UI_EXT, 16UI_EXT, 32UI_EXT, 8I_EXT, 16I_EXT, 32I_EXT}

The following image classes are currently disallowed:

- Textures with borders
- Multisampled renderbuffers

Parameters:

pCudaResource - Pointer to the returned object handle

image - name of texture or renderbuffer object to be registered

target - Identifies the type of object specified by `image`

Flags - Register flags

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnregisterResource, cuGraphicsMapResources, cuGraphicsSubResourceGetMappedArray

5.51.4.4 CUresult cuWGLGetDevice (CUdevice * *pDevice*, HGPUNV *hGpu*)

Returns in `*pDevice` the CUDA device associated with a `hGpu`, if applicable.

Parameters:

pDevice - Device associated with hGpu

hGpu - Handle to a GPU, as queried via WGL_NV_gpu_affinity()

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject, cuGLUnmapBufferObjectAsync, cuGLSetBufferObjectMapFlags

5.52 OpenGL Interoperability [DEPRECATED]

Typedefs

- typedef enum CUGLmap_flags_enum CUGLmap_flags

Enumerations

- enum CUGLmap_flags_enum

Functions

- CUresult cuGLCtxCreate (CUcontext *pCtx, unsigned int Flags, CUdevice device)
  Create a CUDA context for interoperability with OpenGL.
- CUresult cuGLInit (void)
  Initializes OpenGL interoperability.
- CUresult cuGLMapBufferObject (CUdeviceptr *dptr, size_t *size, GLuint buffer)
  Maps an OpenGL buffer object.
- CUresult cuGLMapBufferObjectAsync (CUdeviceptr *dptr, size_t *size, GLuint buffer, CUstream hStream)
  Maps an OpenGL buffer object.
- CUresult cuGLRegisterBufferObject (GLuint buffer)
  Registers an OpenGL buffer object.
- CUresult cuGLSetBufferObjectMapFlags (GLuint buffer, unsigned int Flags)
  Set the map flags for an OpenGL buffer object.
- CUresult cuGLUnmapBufferObject (GLuint buffer)
  Unmaps an OpenGL buffer object.
- CUresult cuGLUnmapBufferObjectAsync (GLuint buffer, CUstream hStream)
  Unmaps an OpenGL buffer object.
- CUresult cuGLUnregisterBufferObject (GLuint buffer)
  Unregister an OpenGL buffer object.

5.52.1 Detailed Description

This section describes deprecated OpenGL interoperability functionality.

5.52.2 Typedef Documentation

5.52.2.1 typedef enum CUGLmap_flags_enum CUGLmap_flags

Flags to map or unmap a resource

5.52.3 Enumeration Type Documentation

5.52.3.1 enum CUGLmap_flags_enum

Flags to map or unmap a resource

5.52.4 Function Documentation

5.52.4.1 CUresult cuGLCtxCreate (CUcontext * *pCtx*, unsigned int *Flags*, CUdevice *device*)

Deprecated

This function is deprecated as of Cuda 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA context with an OpenGL context in order to achieve maximum interoperability performance.

Parameters:

pCtx - Returned CUDA context

Flags - Options for CUDA context creation

device - Device on which to create the context

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuGLInit, cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject, cuGLMapBufferObjectAsync, cuGLUnmapBufferObjectAsync, cuGLSetBufferObjectMapFlags, cuWGLGetDevice

5.52.4.2 CUresult cuGLInit (void)

Deprecated

This function is deprecated as of Cuda 3.0.

Initializes OpenGL interoperability. This function is deprecated and calling it is no longer required. It may fail if the needed OpenGL driver facilities are not available.

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject, cuGLMapBufferObjectAsync, cuGLUnmapBufferObjectAsync, cuGLSetBufferObjectMapFlags, cuWGLGetDevice

5.52.4.3 CUresult cuGLMapBufferObject (CUdeviceptr * *dptr*, size_t * *size*, GLuint *buffer*)

Deprecated

This function is deprecated as of Cuda 3.0.

Maps the buffer object specified by `buffer` into the address space of the current CUDA context and returns in `*dptr` and `*size` the base pointer and size of the resulting mapping.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

All streams in the current CUDA context are synchronized with the current GL context.

Parameters:

dptr - Returned mapped base pointer

size - Returned size of mapping

buffer - The name of the buffer object to map

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_MAP_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

5.52.4.4 CUresult cuGLMapBufferObjectAsync (CUdeviceptr * *dptr*, size_t * *size*, GLuint *buffer*, CUstream *hStream*)

Deprecated

This function is deprecated as of Cuda 3.0.

Maps the buffer object specified by `buffer` into the address space of the current CUDA context and returns in `*dptr` and `*size` the base pointer and size of the resulting mapping.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Stream `hStream` in the current CUDA context is synchronized with the current GL context.

Parameters:

dptr - Returned mapped base pointer

size - Returned size of mapping

buffer - The name of the buffer object to map

hStream - Stream to synchronize

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_MAP_FAILED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

5.52.4.5 CUresult cuGLRegisterBufferObject (GLuint *buffer*)

Deprecated

This function is deprecated as of Cuda 3.0.

Registers the buffer object specified by `buffer` for access by CUDA. This function must be called before CUDA can map the buffer object. There must be a valid OpenGL context bound to the current thread when this function is called, and the buffer name is resolved by that context.

Parameters:

buffer - The name of the buffer object to register.

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_ALREADY_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsGLRegisterBuffer

5.52.4.6 CUresult cuGLSetBufferObjectMapFlags (GLuint *buffer*, unsigned int *Flags*)

Deprecated

This function is deprecated as of Cuda 3.0.

Sets the map flags for the buffer object specified by `buffer`.

Changes to `Flags` will take effect the next time `buffer` is mapped. The `Flags` argument may be any of the following:

- CU_GL_MAP_RESOURCE_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- CU_GL_MAP_RESOURCE_FLAGS_READ_ONLY: Specifies that CUDA kernels which access this resource will not write to this resource.
- CU_GL_MAP_RESOURCE_FLAGS_WRITE_DISCARD: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `buffer` has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned. If `buffer` is presently mapped for access by CUDA, then CUDA_ERROR_ALREADY_MAPPED is returned.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Parameters:

buffer - Buffer object to unmap

Flags - Map flags

Returns:

CUDA_SUCCESS, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceSetMapFlags

5.52.4.7 CUresult cuGLUnmapBufferObject (GLuint *buffer*)

Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the buffer object specified by `buffer` for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

All streams in the current CUDA context are synchronized with the current GL context.

Parameters:

buffer - Buffer object to unmap

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnmapResources

5.52.4.8 CUresult cuGLUnmapBufferObjectAsync (GLuint *buffer*, CUstream *hStream*)

Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the buffer object specified by `buffer` for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Stream `hStream` in the current CUDA context is synchronized with the current GL context.

Parameters:

buffer - Name of the buffer object to unmap

hStream - Stream to synchronize

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnmapResources

5.52.4.9 CUresult cuGLUnregisterBufferObject (GLuint *buffer*)

Deprecated

This function is deprecated as of Cuda 3.0.

Unregisters the buffer object specified by `buffer`. This releases any resources associated with the registered buffer. After this call, the buffer may no longer be mapped for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Parameters:

buffer - Name of the buffer object to unregister

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnregisterResource

5.53 Direct3D 9 Interoperability

Modules

- Direct3D 9 Interoperability [DEPRECATED]

Typedefs

- typedef enum CUd3d9DeviceList_enum CUd3d9DeviceList

Enumerations

- enum CUd3d9DeviceList_enum {
  CU_D3D9_DEVICE_LIST_ALL = 0x01,
  CU_D3D9_DEVICE_LIST_CURRENT_FRAME = 0x02,
  CU_D3D9_DEVICE_LIST_NEXT_FRAME = 0x03 }

Functions

- CUresult cuD3D9CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, IDirect3DDevice9 *pD3DDevice)

  Create a CUDA context for interoperability with Direct3D 9.

- CUresult cuD3D9CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, IDirect3DDevice9 *pD3DDevice, CUdevice cudaDevice)

  Create a CUDA context for interoperability with Direct3D 9.

- CUresult cuD3D9GetDevice (CUdevice *pCudaDevice, const char *pszAdapterName)

  Gets the CUDA device corresponding to a display adapter.

- CUresult cuD3D9GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 *pD3D9Device, CUd3d9DeviceList deviceList)

  Gets the CUDA devices corresponding to a Direct3D 9 device.

- CUresult cuD3D9GetDirect3DDevice (IDirect3DDevice9 **ppD3DDevice)

  Get the Direct3D 9 device against which the current CUDA context was created.

- CUresult cuGraphicsD3D9RegisterResource (CUgraphicsResource *pCudaResource, IDirect3DResource9 *pD3DResource, unsigned int Flags)

  Register a Direct3D 9 resource for access by CUDA.

5.53.1 Detailed Description

This section describes the Direct3D 9 interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of Direct3D 9 resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interopability.

5.53.2 Typedef Documentation

5.53.2.1 typedef enum CUd3d9DeviceList_enum CUd3d9DeviceList

CUDA devices corresponding to a D3D9 device

5.53.3 Enumeration Type Documentation

5.53.3.1 enum CUd3d9DeviceList_enum

CUDA devices corresponding to a D3D9 device

Enumerator:

CU_D3D9_DEVICE_LIST_ALL The CUDA devices for all GPUs used by a D3D9 device

CU_D3D9_DEVICE_LIST_CURRENT_FRAME The CUDA devices for the GPUs used by a D3D9 device in its currently rendering frame

CU_D3D9_DEVICE_LIST_NEXT_FRAME The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

5.53.4 Function Documentation

5.53.4.1 CUresult cuD3D9CtxCreate (CUcontext * *pCtx*, CUdevice * *pCudaDevice*, unsigned int *Flags*, IDirect3DDevice9 * *pD3DDevice*)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device `pD3DDevice`, and associates the created CUDA context with the calling thread. The created CUcontext will be returned in `*pCtx`. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context. If `pCudaDevice` is non-NULL then the CUdevice on which this CUDA context was created will be returned in `*pCudaDevice`.

On success, this call will increase the internal reference count on `pD3DDevice`. This reference count will be decremented upon destruction of this context through cuCtxDestroy(). This context will cease to function if `pD3DDevice` is destroyed or encounters an error.

Note that this function is never required for correct functionality. Use of this function will result in accelerated interoperability only when the operating system is Windows Vista or Windows 7, and the device `pD3DDdevice` is not an IDirect3DDevice9Ex. In all other cirumstances, this function is not necessary.

Parameters:

pCtx - Returned newly created CUDA context

pCudaDevice - Returned pointer to the device on which the context was created

Flags - Context creation flags (see cuCtxCreate() for details)

pD3DDevice - Direct3D device to create interoperability context with

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D9GetDevice, cuGraphicsD3D9RegisterResource

5.53.4.2 CUresult cuD3D9CtxCreateOnDevice (CUcontext * *pCtx*, unsigned int *flags*, IDirect3DDevice9 * *pD3DDevice*, CUdevice *cudaDevice*)

Creates a new CUDA context, enables interoperability for that context with the Direct3D device `pD3DDevice`, and associates the created CUDA context with the calling thread. The created CUcontext will be returned in `*pCtx`. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context.

On success, this call will increase the internal reference count on `pD3DDevice`. This reference count will be decremented upon destruction of this context through cuCtxDestroy(). This context will cease to function if `pD3DDevice` is destroyed or encounters an error.

Note that this function is never required for correct functionality. Use of this function will result in accelerated interoperability only when the operating system is Windows Vista or Windows 7, and the device `pD3DDdevice` is not an IDirect3DDevice9Ex. In all other cirumstances, this function is not necessary.

Parameters:

pCtx - Returned newly created CUDA context

flags - Context creation flags (see cuCtxCreate() for details)

pD3DDevice - Direct3D device to create interoperability context with

cudaDevice - The CUDA device on which to create the context. This device must be among the devices returned when querying CU_D3D9_DEVICES_ALL from cuD3D9GetDevices.

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D9GetDevices, cuGraphicsD3D9RegisterResource

5.53.4.3 CUresult cuD3D9GetDevice (CUdevice * *pCudaDevice*, const char * *pszAdapterName*)

Returns in `*pCudaDevice` the CUDA-compatible device corresponding to the adapter name `pszAdapterName` obtained from EnumDisplayDevices() or IDirect3D9::GetAdapterIdentifier().

If no device on the adapter with name `pszAdapterName` is CUDA-compatible, then the call will fail.

Parameters:

pCudaDevice - Returned CUDA device corresponding to pszAdapterName

pszAdapterName - Adapter name to query for device

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D9CtxCreate

5.53.4.4 CUresult cuD3D9GetDevices (unsigned int * *pCudaDeviceCount*, CUdevice * *pCudaDevices*, unsigned int *cudaDeviceCount*, IDirect3DDevice9 * *pD3D9Device*, CUd3d9DeviceList *deviceList*)

Returns in `*pCudaDeviceCount` the number of CUDA-compatible device corresponding to the Direct3D 9 device `pD3D9Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 9 device `pD3D9Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return CUDA_ERROR_NO_DEVICE.

Parameters:

pCudaDeviceCount - Returned number of CUDA devices corresponding to `pD3D9Device`

pCudaDevices - Returned CUDA devices corresponding to `pD3D9Device`

cudaDeviceCount - The size of the output device array `pCudaDevices`

pD3D9Device - Direct3D 9 device to query for CUDA devices

deviceList - The set of devices to return. This set may be CU_D3D9_DEVICE_LIST_ALL for all devices, CU_D3D9_DEVICE_LIST_CURRENT_FRAME for the devices used to render the current frame (in SLI), or CU_D3D9_DEVICE_LIST_NEXT_FRAME for the devices used to render the next frame (in SLI).

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D9CtxCreate

5.53.4.5 CUresult cuD3D9GetDirect3DDevice (IDirect3DDevice9 ** *ppD3DDevice*)

Returns in `*ppD3DDevice` the Direct3D device against which this CUDA context was created in cuD3D9CtxCreate().

Parameters:

ppD3DDevice - Returned Direct3D device corresponding to CUDA context

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D9GetDevice

5.53.4.6 CUresult cuGraphicsD3D9RegisterResource (CUgraphicsResource * *pCudaResource*, IDirect3DResource9 * *pD3DResource*, unsigned int *Flags*)

Registers the Direct3D 9 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3Dresource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through cuGraphicsUnregisterResource().

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following.

- IDirect3DVertexBuffer9: may be accessed through a device pointer
- IDirect3DIndexBuffer9: may be accessed through a device pointer
- IDirect3DSurface9: may be accessed through an array. Only stand-alone objects of type IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- IDirect3DBaseTexture9: individual surfaces on this texture may be accessed through an array.

The `Flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used.
- CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.
- CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized for this context using cuD3D9CtxCreate then CUDA_ERROR_INVALID_CONTEXT is returned. If `pD3DResource` is of incorrect type or is already registered then CUDA_ERROR_INVALID_HANDLE is returned. If `pD3DResource` cannot be registered then CUDA_ERROR_UNKNOWN is returned. If `Flags` is not one of the above specified value then CUDA_ERROR_INVALID_VALUE is returned.

Parameters:

pCudaResource - Returned graphics resource handle

pD3DResource - Direct3D resource to register

Flags - Parameters for resource registration

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D9CtxCreate, cuGraphicsUnregisterResource, cuGraphicsMapResources, cuGraphicsSubRe-sourceGetMappedArray, cuGraphicsResourceGetMappedPointer

5.54 Direct3D 9 Interoperability [DEPRECATED]

Typedefs

- typedef enum CUd3d9map_flags_enum CUd3d9map_flags
- typedef enum CUd3d9register_flags_enum CUd3d9register_flags

Enumerations

- enum CUd3d9map_flags_enum
- enum CUd3d9register_flags_enum

Functions

- CUresult cuD3D9MapResources (unsigned int count, IDirect3DResource9 **ppResource)

  Map Direct3D resources for access by CUDA.

- CUresult cuD3D9RegisterResource (IDirect3DResource9 *pResource, unsigned int Flags)

  Register a Direct3D resource for access by CUDA.

- CUresult cuD3D9ResourceGetMappedArray (CUarray *pArray, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

  Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D9ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

  Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D9ResourceGetMappedPointer (CUdeviceptr *pDevPtr, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

  Get the pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D9ResourceGetMappedSize (size_t *pSize, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

  Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D9ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

  Get the dimensions of a registered surface.

- CUresult cuD3D9ResourceSetMapFlags (IDirect3DResource9 *pResource, unsigned int Flags)

  Set usage flags for mapping a Direct3D resource.

- CUresult cuD3D9UnmapResources (unsigned int count, IDirect3DResource9 **ppResource)

  Unmaps Direct3D resources.

- CUresult cuD3D9UnregisterResource (IDirect3DResource9 *pResource)

  Unregister a Direct3D resource.

5.54.1 Detailed Description

This section describes deprecated Direct3D 9 interoperability functionality.

5.54.2 Typedef Documentation

5.54.2.1 typedef enum CUd3d9map_flags_enum CUd3d9map_flags

Flags to map or unmap a resource

5.54.2.2 typedef enum CUd3d9register_flags_enum CUd3d9register_flags

Flags to register a resource

5.54.3 Enumeration Type Documentation

5.54.3.1 enum CUd3d9map_flags_enum

Flags to map or unmap a resource

5.54.3.2 enum CUd3d9register_flags_enum

Flags to register a resource

5.54.4 Function Documentation

5.54.4.1 CUresult cuD3D9MapResources (unsigned int *count*, IDirect3DResource9 ** *ppResource*)

Deprecated

This function is deprecated as of Cuda 3.0.

Maps the `count` Direct3D resources in `ppResource` for access by CUDA.

The resources in `ppResource` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before cuD3D9MapResources() will complete before any CUDA kernels issued after cuD3D9MapResources() begin.

If any of `ppResource` have not been registered for use with CUDA or if `ppResource` contains any duplicate entries, then CUDA_ERROR_INVALID_HANDLE is returned. If any of `ppResource` are presently mapped for access by CUDA, then CUDA_ERROR_ALREADY_MAPPED is returned.

Parameters:

count - Number of resources in ppResource

ppResource - Resources to map for CUDA usage

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_-MAPPED, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

5.54.4.2 CUresult cuD3D9RegisterResource (IDirect3DResource9 * *pResource*, unsigned int *Flags*)

Deprecated

This function is deprecated as of Cuda 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cuD3D9UnregisterResource(). Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through cuD3D9UnregisterResource().

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following.

- IDirect3DVertexBuffer9: Cannot be used with `Flags` set to CU_D3D9_REGISTER_FLAGS_ARRAY.
- IDirect3DIndexBuffer9: Cannot be used with `Flags` set to CU_D3D9_REGISTER_FLAGS_ARRAY.
- IDirect3DSurface9: Only stand-alone objects of type IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object. For restrictions on the `Flags` parameter, see type IDirect3DBaseTexture9.
- IDirect3DBaseTexture9: When a texture is registered, all surfaces associated with the all mipmap levels of all faces of the texture will be accessible to CUDA.

The `Flags` argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- CU_D3D9_REGISTER_FLAGS_NONE: Specifies that CUDA will access this resource through a CUdeviceptr. The pointer, size, and (for textures), pitch for each subresource of this allocation may be queried through cuD3D9ResourceGetMappedPointer(), cuD3D9ResourceGetMappedSize(), and cuD3D9ResourceGetMappedPitch() respectively. This option is valid for all resource types.
- CU_D3D9_REGISTER_FLAGS_ARRAY: Specifies that CUDA will access this resource through a CUarray queried on a sub-resource basis through cuD3D9ResourceGetMappedArray(). This option is only valid for resources of type IDirect3DSurface9 and subtypes of IDirect3DBaseTexture9.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Any resources allocated in D3DPOOL_SYSTEMMEM or D3DPOOL_MANAGED may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context, then CUDA_ERROR_INVALID_CONTEXT is returned. If `pResource` is of incorrect type (e.g. is a non-stand-alone IDirect3DSurface9) or is already registered, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` cannot be registered then CUDA_ERROR_UNKNOWN is returned.

Parameters:

pResource - Resource to register for CUDA access

Flags - Flags for resource registration

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsD3D9RegisterResource

5.54.4.3 CUresult cuD3D9ResourceGetMappedArray (CUarray * *pArray*, IDirect3DResource9 * *pResource*, unsigned int *Face*, unsigned int *Level*)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in `*pArray` an array through which the subresource of the mapped Direct3D resource `pResource` which corresponds to `Face` and `Level` may be accessed. The value set in `pArray` may change every time that `pResource` is mapped.

If `pResource` is not registered then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not registered with usage flags CU_D3D9_REGISTER_FLAGS_ARRAY then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of `Face` and `Level` parameters, see cuD3D9ResourceGetMappedPointer().

Parameters:

pArray - Returned array corresponding to subresource

pResource - Mapped resource to access

Face - Face of resource to access

Level - Level of resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsSubResourceGetMappedArray

5.54.4.4 CUresult cuD3D9ResourceGetMappedPitch (size_t * *pPitch*, size_t * *pPitchSlice*, IDirect3DResource9 * *pResource*, unsigned int *Face*, unsigned int *Level*)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in `*pPitch` and `*pPitchSlice` the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `Face` and `Level`. The values set in `pPitch` and `pPitchSlice` may change every time that `pResource` is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position **x**, **y** from the base pointer of the surface is:

y * **pitch** + (**bytes per pixel**) * **x**

For a 3D surface, the byte offset of the sample at position **x**, **y**, **z** from the base pointer of the surface is:

z* **slicePitch** + **y** * **pitch** + (**bytes per pixel**) * **x**

Both parameters `pPitch` and `pPitchSlice` are optional and may be set to NULL.

If `pResource` is not of type IDirect3DBaseTexture9 or one of its sub-types or if `pResource` has not been registered for use with CUDA, then cudaErrorInvalidResourceHandle is returned. If `pResource` was not registered with usage flags CU_D3D9_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped for access by CUDA then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of `Face` and `Level` parameters, see cuD3D9ResourceGetMappedPointer().

Parameters:

pPitch - Returned pitch of subresource

pPitchSlice - Returned Z-slice pitch of subresource

pResource - Mapped resource to access

Face - Face of resource to access

Level - Level of resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsSubResourceGetMappedArray

5.54.4.5 CUresult cuD3D9ResourceGetMappedPointer (CUdeviceptr * *pDevPtr*, IDirect3DResource9 * *pResource*, unsigned int *Face*, unsigned int *Level*)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in `*pDevPtr` the base pointer of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `Face` and `Level`. The value set in `pDevPtr` may change every time that `pResource` is mapped.

If `pResource` is not registered, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not registered with usage flags CU_D3D9_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped, then CUDA_ERROR_NOT_MAPPED is returned.

If `pResource` is of type IDirect3DCubeTexture9, then `Face` must one of the values enumerated by type D3DCUBEMAP_FACES. For all other types `Face` must be 0. If `Face` is invalid, then CUDA_ERROR_INVALID_VALUE is returned.

If `pResource` is of type IDirect3DBaseTexture9, then `Level` must correspond to a valid mipmap level. At present only mipmap level 0 is supported. For all other types `Level` must be 0. If `Level` is invalid, then CUDA_ERROR_INVALID_VALUE is returned.

Parameters:

pDevPtr - Returned pointer corresponding to subresource

pResource - Mapped resource to access

Face - Face of resource to access

Level - Level of resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceGetMappedPointer

5.54.4.6 CUresult cuD3D9ResourceGetMappedSize (size_t * *pSize*, IDirect3DResource9 * *pResource*, unsigned int *Face*, unsigned int *Level*)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in *pSize the size of the subresource of the mapped Direct3D resource pResource, which corresponds to Face and Level. The value set in pSize may change every time that pResource is mapped.

If pResource has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource was not registered with usage flags CU_D3D9_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of Face and Level parameters, see cuD3D9ResourceGetMappedPointer.

Parameters:

pSize - Returned size of subresource

pResource - Mapped resource to access

Face - Face of resource to access

Level - Level of resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceGetMappedPointer

5.54.4.7 CUresult cuD3D9ResourceGetSurfaceDimensions (size_t * *pWidth*, size_t * *pHeight*, size_t * *pDepth*, IDirect3DResource9 * *pResource*, unsigned int *Face*, unsigned int *Level*)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in *pWidth, *pHeight, and *pDepth the dimensions of the subresource of the mapped Direct3D resource pResource, which corresponds to Face and Level.

Because anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters pWidth, pHeight, and pDepth are optional. For 2D surfaces, the value returned in *pDepth will be 0.

If pResource is not of type IDirect3DBaseTexture9 or IDirect3DSurface9 or if pResource has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned.

For usage requirements of Face and Level parameters, see cuD3D9ResourceGetMappedPointer().

Parameters:

pWidth - Returned width of surface

pHeight - Returned height of surface

pDepth - Returned depth of surface

pResource - Registered resource to access

Face - Face of resource to access

Level - Level of resource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsSubResourceGetMappedArray

5.54.4.8 CUresult cuD3D9ResourceSetMapFlags (IDirect3DResource9 * *pResource*, unsigned int *Flags*)

Deprecated

This function is deprecated as of Cuda 3.0.

Set `Flags` for mapping the Direct3D resource `pResource`.

Changes to `Flags` will take effect the next time `pResource` is mapped. The `Flags` argument may be any of the following:

- CU_D3D9_MAPRESOURCE_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- CU_D3D9_MAPRESOURCE_FLAGS_READONLY: Specifies that CUDA kernels which access this resource will not write to this resource.
- CU_D3D9_MAPRESOURCE_FLAGS_WRITEDISCARD: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is presently mapped for access by CUDA, then CUDA_ERROR_ALREADY_MAPPED is returned.

Parameters:

pResource - Registered resource to set flags for

Flags - Parameters for resource mapping

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceSetMapFlags

5.54.4.9 CUresult cuD3D9UnmapResources (unsigned int *count*, IDirect3DResource9 ** *ppResource*)

Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the `count` Direct3D resources in `ppResource`.

This function provides the synchronization guarantee that any CUDA kernels issued before cuD3D9UnmapResources() will complete before any Direct3D calls issued after cuD3D9UnmapResources() begin.

If any of `ppResource` have not been registered for use with CUDA or if `ppResource` contains any duplicate entries, then CUDA_ERROR_INVALID_HANDLE is returned. If any of `ppResource` are not presently mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

Parameters:

count - Number of resources to unmap for CUDA

ppResource - Resources to unmap for CUDA

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnmapResources

5.54.4.10 CUresult cuD3D9UnregisterResource (IDirect3DResource9 * *pResource*)

Deprecated

This function is deprecated as of Cuda 3.0.

Unregisters the Direct3D resource `pResource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then CUDA_ERROR_INVALID_HANDLE is returned.

Parameters:

pResource - Resource to unregister

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnregisterResource

5.55 Direct3D 10 Interoperability

Modules

- Direct3D 10 Interoperability [DEPRECATED]

Typedefs

- typedef enum CUd3d10DeviceList_enum CUd3d10DeviceList

Enumerations

- enum CUd3d10DeviceList_enum {
  CU_D3D10_DEVICE_LIST_ALL = 0x01,
  CU_D3D10_DEVICE_LIST_CURRENT_FRAME = 0x02,
  CU_D3D10_DEVICE_LIST_NEXT_FRAME = 0x03 }

Functions

- CUresult cuD3D10GetDevice (CUdevice *pCudaDevice, IDXGIAdapter *pAdapter)
  Gets the CUDA device corresponding to a display adapter.
- CUresult cuD3D10GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device *pD3D10Device, CUd3d10DeviceList deviceList)
  Gets the CUDA devices corresponding to a Direct3D 10 device.
- CUresult cuGraphicsD3D10RegisterResource (CUgraphicsResource *pCudaResource, ID3D10Resource *pD3DResource, unsigned int Flags)
  Register a Direct3D 10 resource for access by CUDA.

5.55.1 Detailed Description

This section describes the Direct3D 10 interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of Direct3D 10 resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interopability.

5.55.2 Typedef Documentation

5.55.2.1 typedef enum CUd3d10DeviceList_enum CUd3d10DeviceList

CUDA devices corresponding to a D3D10 device

5.55.3 Enumeration Type Documentation

5.55.3.1 enum CUd3d10DeviceList_enum

CUDA devices corresponding to a D3D10 device

Enumerator:

CU_D3D10_DEVICE_LIST_ALL The CUDA devices for all GPUs used by a D3D10 device

CU_D3D10_DEVICE_LIST_CURRENT_FRAME The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame

CU_D3D10_DEVICE_LIST_NEXT_FRAME The CUDA devices for the GPUs to be used by a D3D10 device in the next frame

5.55.4 Function Documentation

5.55.4.1 CUresult cuD3D10GetDevice (CUdevice * *pCudaDevice*, IDXGIAdapter * *pAdapter*)

Returns in `*pCudaDevice` the CUDA-compatible device corresponding to the adapter `pAdapter` obtained from IDXGIFactory::EnumAdapters.

If no device on `pAdapter` is CUDA-compatible then the call will fail.

Parameters:

pCudaDevice - Returned CUDA device corresponding to `pAdapter`

pAdapter - Adapter to query for CUDA device

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D10GetDevices

5.55.4.2 CUresult cuD3D10GetDevices (unsigned int * *pCudaDeviceCount*, CUdevice * *pCudaDevices*, unsigned int *cudaDeviceCount*, ID3D10Device * *pD3D10Device*, CUd3d10DeviceList *deviceList*)

Returns in `*pCudaDeviceCount` the number of CUDA-compatible device corresponding to the Direct3D 10 device `pD3D10Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 10 device `pD3D10Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return CUDA_ERROR_NO_DEVICE.

Parameters:

pCudaDeviceCount - Returned number of CUDA devices corresponding to `pD3D10Device`

pCudaDevices - Returned CUDA devices corresponding to `pD3D10Device`

cudaDeviceCount - The size of the output device array `pCudaDevices`

pD3D10Device - Direct3D 10 device to query for CUDA devices

deviceList - The set of devices to return. This set may be CU_D3D10_DEVICE_LIST_ALL for all devices, CU_D3D10_DEVICE_LIST_CURRENT_FRAME for the devices used to render the current frame (in SLI), or CU_D3D10_DEVICE_LIST_NEXT_FRAME for the devices used to render the next frame (in SLI).

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_NO_DEVICE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_-ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D10GetDevice

5.55.4.3 CUresult cuGraphicsD3D10RegisterResource (CUgraphicsResource * *pCudaResource*, ID3D10Resource * *pD3DResource*, unsigned int *Flags*)

Registers the Direct3D 10 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3Dresource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through cuGraphicsUnregisterResource().

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following.

- ID3D10Buffer: may be accessed through a device pointer.
- ID3D10Texture1D: individual subresources of the texture may be accessed via arrays
- ID3D10Texture2D: individual subresources of the texture may be accessed via arrays
- ID3D10Texture3D: individual subresources of the texture may be accessed via arrays

The `Flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used.
- CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.
- CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If `pD3DResource` is of incorrect type or is already registered then CUDA_ERROR_INVALID_HANDLE is returned. If `pD3DResource` cannot be registered then CUDA_ERROR_UNKNOWN is returned. If `Flags` is not one of the above specified value then CUDA_ERROR_INVALID_VALUE is returned.

Parameters:

pCudaResource - Returned graphics resource handle

pD3DResource - Direct3D resource to register

Flags - Parameters for resource registration

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnregisterResource, cuGraphicsMapResources, cuGraphicsSubResourceGetMappedArray, cuGraphicsResourceGetMappedPointer

5.56 Direct3D 10 Interoperability [DEPRECATED]

Typedefs

- typedef enum CUD3D10map_flags_enum CUD3D10map_flags
- typedef enum CUD3D10register_flags_enum CUD3D10register_flags

Enumerations

- enum CUD3D10map_flags_enum
- enum CUD3D10register_flags_enum

Functions

- CUresult cuD3D10CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, ID3D10Device *pD3DDevice)

  Create a CUDA context for interoperability with Direct3D 10.

- CUresult cuD3D10CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, ID3D10Device *pD3DDevice, CUdevice cudaDevice)

  Create a CUDA context for interoperability with Direct3D 10.

- CUresult cuD3D10GetDirect3DDevice (ID3D10Device **ppD3DDevice)

  Get the Direct3D 10 device against which the current CUDA context was created.

- CUresult cuD3D10MapResources (unsigned int count, ID3D10Resource **ppResources)

  Map Direct3D resources for access by CUDA.

- CUresult cuD3D10RegisterResource (ID3D10Resource *pResource, unsigned int Flags)

  Register a Direct3D resource for access by CUDA.

- CUresult cuD3D10ResourceGetMappedArray (CUarray *pArray, ID3D10Resource *pResource, unsigned int SubResource)

  Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D10ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, ID3D10Resource *pResource, unsigned int SubResource)

  Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D10ResourceGetMappedPointer (CUdeviceptr *pDevPtr, ID3D10Resource *pResource, unsigned int SubResource)

  Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D10ResourceGetMappedSize (size_t *pSize, ID3D10Resource *pResource, unsigned int SubResource)

  Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

- CUresult cuD3D10ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, ID3D10Resource *pResource, unsigned int SubResource)

Get the dimensions of a registered surface.

- CUresult cuD3D10ResourceSetMapFlags (ID3D10Resource ∗pResource, unsigned int Flags)

  Set usage flags for mapping a Direct3D resource.

- CUresult cuD3D10UnmapResources (unsigned int count, ID3D10Resource ∗∗ppResources)

  Unmap Direct3D resources.

- CUresult cuD3D10UnregisterResource (ID3D10Resource ∗pResource)

  Unregister a Direct3D resource.

5.56.1 Detailed Description

This section describes deprecated Direct3D 10 interoperability functionality.

5.56.2 Typedef Documentation

5.56.2.1 typedef enum CUD3D10map_flags_enum CUD3D10map_flags

Flags to map or unmap a resource

5.56.2.2 typedef enum CUD3D10register_flags_enum CUD3D10register_flags

Flags to register a resource

5.56.3 Enumeration Type Documentation

5.56.3.1 enum CUD3D10map_flags_enum

Flags to map or unmap a resource

5.56.3.2 enum CUD3D10register_flags_enum

Flags to register a resource

5.56.4 Function Documentation

5.56.4.1 CUresult cuD3D10CtxCreate (CUcontext ∗ *pCtx*, CUdevice ∗ *pCudaDevice*, unsigned int *Flags*, ID3D10Device ∗ *pD3DDevice*)

Deprecated

This function is deprecated as of Cuda 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA context with a D3D10 device in order to achieve maximum interoperability performance.

Parameters:

pCtx - Returned newly created CUDA context

pCudaDevice - Returned pointer to the device on which the context was created

Flags - Context creation flags (see cuCtxCreate() for details)

pD3DDevice - Direct3D device to create interoperability context with

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D10GetDevice, cuGraphicsD3D10RegisterResource

5.56.4.2 CUresult cuD3D10CtxCreateOnDevice (CUcontext * *pCtx*, unsigned int *flags*, ID3D10Device * *pD3DDevice*, CUdevice *cudaDevice*)

Deprecated

This function is deprecated as of Cuda 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA context with a D3D10 device in order to achieve maximum interoperability performance.

Parameters:

pCtx - Returned newly created CUDA context

flags - Context creation flags (see cuCtxCreate() for details)

pD3DDevice - Direct3D device to create interoperability context with

cudaDevice - The CUDA device on which to create the context. This device must be among the devices returned when querying CU_D3D10_DEVICES_ALL from cuD3D10GetDevices.

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D10GetDevices, cuGraphicsD3D10RegisterResource

5.56.4.3 CUresult cuD3D10GetDirect3DDevice (ID3D10Device ** *ppD3DDevice*)

Deprecated

This function is deprecated as of Cuda 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA context with a D3D10 device in order to achieve maximum interoperability performance.

Parameters:

ppD3DDevice - Returned Direct3D device corresponding to CUDA context

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D10GetDevice

5.56.4.4 CUresult cuD3D10MapResources (unsigned int *count*, ID3D10Resource ** *ppResources*)

Deprecated

This function is deprecated as of Cuda 3.0.

Maps the `count` Direct3D resources in `ppResources` for access by CUDA.

The resources in `ppResources` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before cuD3D10MapResources() will complete before any CUDA kernels issued after cuD3D10MapResources() begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then CUDA_ERROR_INVALID_HANDLE is returned. If any of `ppResources` are presently mapped for access by CUDA, then CUDA_ERROR_ALREADY_MAPPED is returned.

Parameters:

count - Number of resources to map for CUDA

ppResources - Resources to map for CUDA

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_-MAPPED, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

5.56.4.5 CUresult cuD3D10RegisterResource (ID3D10Resource * *pResource*, unsigned int *Flags*)

Deprecated

This function is deprecated as of Cuda 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through cuD3D10UnregisterResource(). Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through cuD3D10UnregisterResource().

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following.

- ID3D10Buffer: Cannot be used with `Flags` set to CU_D3D10_REGISTER_FLAGS_ARRAY.
- ID3D10Texture1D: No restrictions.
- ID3D10Texture2D: No restrictions.
- ID3D10Texture3D: No restrictions.

The `Flags` argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- CU_D3D10_REGISTER_FLAGS_NONE: Specifies that CUDA will access this resource through a CUdeviceptr. The pointer, size, and (for textures), pitch for each subresource of this allocation may be queried through cuD3D10ResourceGetMappedPointer(), cuD3D10ResourceGetMappedSize(), and cuD3D10ResourceGetMappedPitch() respectively. This option is valid for all resource types.
- CU_D3D10_REGISTER_FLAGS_ARRAY: Specifies that CUDA will access this resource through a CUarray queried on a sub-resource basis through cuD3D10ResourceGetMappedArray(). This option is only valid for resources of type ID3D10Texture1D, ID3D10Texture2D, and ID3D10Texture3D.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context then CUDA_ERROR_INVALID_CONTEXT is returned. If `pResource` is of incorrect type or is already registered, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` cannot be registered, then CUDA_ERROR_UNKNOWN is returned.

Parameters:

pResource - Resource to register

Flags - Parameters for resource registration

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsD3D10RegisterResource

5.56.4.6 CUresult cuD3D10ResourceGetMappedArray (CUarray * *pArray*, ID3D10Resource * *pResource*, unsigned int *SubResource*)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in `*pArray` an array through which the subresource of the mapped Direct3D resource `pResource`, which corresponds to `SubResource` may be accessed. The value set in `pArray` may change every time that `pResource` is mapped.

If `pResource` is not registered, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not registered with usage flags CU_D3D10_REGISTER_FLAGS_ARRAY, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of the `SubResource` parameter, see cuD3D10ResourceGetMappedPointer().

Parameters:

pArray - Returned array corresponding to subresource

pResource - Mapped resource to access

SubResource - Subresource of pResource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsSubResourceGetMappedArray

5.56.4.7 CUresult cuD3D10ResourceGetMappedPitch (size_t * *pPitch*, size_t * *pPitchSlice*, ID3D10Resource * *pResource*, unsigned int *SubResource*)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in `*pPitch` and `*pPitchSlice` the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `SubResource`. The values set in `pPitch` and `pPitchSlice` may change every time that `pResource` is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position **x**, **y** from the base pointer of the surface is:

y * **pitch** + (**bytes per pixel**) * **x**

For a 3D surface, the byte offset of the sample at position **x**, **y**, **z** from the base pointer of the surface is:

z* **slicePitch** + **y** * **pitch** + (**bytes per pixel**) * **x**

Both parameters `pPitch` and `pPitchSlice` are optional and may be set to NULL.

If `pResource` is not of type IDirect3DBaseTexture10 or one of its sub-types or if `pResource` has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not registered with usage flags CU_D3D10_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of the `SubResource` parameter, see cuD3D10ResourceGetMappedPointer().

Parameters:

pPitch - Returned pitch of subresource

pPitchSlice - Returned Z-slice pitch of subresource

pResource - Mapped resource to access

SubResource - Subresource of pResource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsSubResourceGetMappedArray

5.56.4.8 CUresult cuD3D10ResourceGetMappedPointer (CUdeviceptr * *pDevPtr*, ID3D10Resource * *pResource*, unsigned int *SubResource*)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in *pDevPtr the base pointer of the subresource of the mapped Direct3D resource pResource, which corresponds to SubResource. The value set in pDevPtr may change every time that pResource is mapped.

If pResource is not registered, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource was not registered with usage flags CU_D3D10_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped, then CUDA_ERROR_NOT_MAPPED is returned.

If pResource is of type ID3D10Buffer, then SubResource must be 0. If pResource is of any other type, then the value of SubResource must come from the subresource calculation in D3D10CalcSubResource().

Parameters:

pDevPtr - Returned pointer corresponding to subresource

pResource - Mapped resource to access

SubResource - Subresource of pResource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceGetMappedPointer

5.56.4.9 CUresult cuD3D10ResourceGetMappedSize (size_t * *pSize*, ID3D10Resource * *pResource*, unsigned int *SubResource*)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in *pSize the size of the subresource of the mapped Direct3D resource pResource, which corresponds to SubResource. The value set in pSize may change every time that pResource is mapped.

If pResource has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource was not registered with usage flags CU_D3D10_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of the SubResource parameter, see cuD3D10ResourceGetMappedPointer().

Parameters:

pSize - Returned size of subresource

pResource - Mapped resource to access

SubResource - Subresource of pResource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceGetMappedPointer

5.56.4.10 CUresult cuD3D10ResourceGetSurfaceDimensions (size_t * *pWidth*, size_t * *pHeight*, size_t * *pDepth*, ID3D10Resource * *pResource*, unsigned int *SubResource*)

Deprecated

This function is deprecated as of Cuda 3.0.

Returns in `*pWidth`, `*pHeight`, and `*pDepth` the dimensions of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `SubResource`.

Because anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters `pWidth`, `pHeight`, and `pDepth` are optional. For 2D surfaces, the value returned in `*pDepth` will be 0.

If `pResource` is not of type IDirect3DBaseTexture10 or IDirect3DSurface10 or if `pResource` has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned.

For usage requirements of the `SubResource` parameter, see cuD3D10ResourceGetMappedPointer().

Parameters:

pWidth - Returned width of surface

pHeight - Returned height of surface

pDepth - Returned depth of surface

pResource - Registered resource to access

SubResource - Subresource of pResource to access

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsSubResourceGetMappedArray

5.56.4.11 CUresult cuD3D10ResourceSetMapFlags (ID3D10Resource * *pResource*, unsigned int *Flags*)

Deprecated

This function is deprecated as of Cuda 3.0.

Set flags for mapping the Direct3D resource `pResource`.

Changes to flags will take effect the next time `pResource` is mapped. The `Flags` argument may be any of the following.

- CU_D3D10_MAPRESOURCE_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- CU_D3D10_MAPRESOURCE_FLAGS_READONLY: Specifies that CUDA kernels which access this resource will not write to this resource.
- CU_D3D10_MAPRESOURCE_FLAGS_WRITEDISCARD: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is presently mapped for access by CUDA then CUDA_ERROR_ALREADY_MAPPED is returned.

Parameters:

pResource - Registered resource to set flags for

Flags - Parameters for resource mapping

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsResourceSetMapFlags

5.56.4.12 CUresult cuD3D10UnmapResources (unsigned int *count*, ID3D10Resource ** *ppResources*)

Deprecated

This function is deprecated as of Cuda 3.0.

Unmaps the `count` Direct3D resources in `ppResources`.

This function provides the synchronization guarantee that any CUDA kernels issued before cuD3D10UnmapResources() will complete before any Direct3D calls issued after cuD3D10UnmapResources() begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then CUDA_ERROR_INVALID_HANDLE is returned. If any of `ppResources` are not presently mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

Parameters:

count - Number of resources to unmap for CUDA

ppResources - Resources to unmap for CUDA

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnmapResources

5.56.4.13 CUresult cuD3D10UnregisterResource (ID3D10Resource * *pResource*)

Deprecated

This function is deprecated as of Cuda 3.0.

Unregisters the Direct3D resource `pResource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then CUDA_ERROR_INVALID_HANDLE is returned.

Parameters:

pResource - Resources to unregister

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnregisterResource

5.57 Direct3D 11 Interoperability

Modules

- Direct3D 11 Interoperability [DEPRECATED]

Typedefs

- typedef enum CUd3d11DeviceList_enum CUd3d11DeviceList

Enumerations

- enum CUd3d11DeviceList_enum {
  CU_D3D11_DEVICE_LIST_ALL = 0x01,
  CU_D3D11_DEVICE_LIST_CURRENT_FRAME = 0x02,
  CU_D3D11_DEVICE_LIST_NEXT_FRAME = 0x03 }

Functions

- CUresult cuD3D11GetDevice (CUdevice *pCudaDevice, IDXGIAdapter *pAdapter)
  Gets the CUDA device corresponding to a display adapter.
- CUresult cuD3D11GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device *pD3D11Device, CUd3d11DeviceList deviceList)
  Gets the CUDA devices corresponding to a Direct3D 11 device.
- CUresult cuGraphicsD3D11RegisterResource (CUgraphicsResource *pCudaResource, ID3D11Resource *pD3DResource, unsigned int Flags)
  Register a Direct3D 11 resource for access by CUDA.

5.57.1 Detailed Description

This section describes the Direct3D 11 interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of Direct3D 11 resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interopability.

5.57.2 Typedef Documentation

5.57.2.1 typedef enum CUd3d11DeviceList_enum CUd3d11DeviceList

CUDA devices corresponding to a D3D11 device

5.57.3 Enumeration Type Documentation

5.57.3.1 enum CUd3d11DeviceList_enum

CUDA devices corresponding to a D3D11 device

Enumerator:

CU_D3D11_DEVICE_LIST_ALL The CUDA devices for all GPUs used by a D3D11 device

CU_D3D11_DEVICE_LIST_CURRENT_FRAME The CUDA devices for the GPUs used by a D3D11 device in its currently rendering frame

CU_D3D11_DEVICE_LIST_NEXT_FRAME The CUDA devices for the GPUs to be used by a D3D11 device in the next frame

5.57.4 Function Documentation

5.57.4.1 CUresult cuD3D11GetDevice (CUdevice * *pCudaDevice*, IDXGIAdapter * *pAdapter*)

Returns in `*pCudaDevice` the CUDA-compatible device corresponding to the adapter `pAdapter` obtained from IDXGIFactory::EnumAdapters.

If no device on `pAdapter` is CUDA-compatible the call will return CUDA_ERROR_NO_DEVICE.

Parameters:

pCudaDevice - Returned CUDA device corresponding to `pAdapter`

pAdapter - Adapter to query for CUDA device

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D11GetDevices

5.57.4.2 CUresult cuD3D11GetDevices (unsigned int * *pCudaDeviceCount*, CUdevice * *pCudaDevices*, unsigned int *cudaDeviceCount*, ID3D11Device * *pD3D11Device*, CUd3d11DeviceList *deviceList*)

Returns in `*pCudaDeviceCount` the number of CUDA-compatible device corresponding to the Direct3D 11 device `pD3D11Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 11 device `pD3D11Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return CUDA_ERROR_NO_DEVICE.

Parameters:

pCudaDeviceCount - Returned number of CUDA devices corresponding to `pD3D11Device`

pCudaDevices - Returned CUDA devices corresponding to `pD3D11Device`

cudaDeviceCount - The size of the output device array `pCudaDevices`

pD3D11Device - Direct3D 11 device to query for CUDA devices

deviceList - The set of devices to return. This set may be CU_D3D11_DEVICE_LIST_ALL for all devices, CU_D3D11_DEVICE_LIST_CURRENT_FRAME for the devices used to render the current frame (in SLI), or CU_D3D11_DEVICE_LIST_NEXT_FRAME for the devices used to render the next frame (in SLI).

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D11GetDevice

5.57.4.3 CUresult cuGraphicsD3D11RegisterResource (CUgraphicsResource * *pCudaResource*, ID3D11Resource * *pD3DResource*, unsigned int *Flags*)

Registers the Direct3D 11 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3Dresource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through cuGraphicsUnregisterResource().

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following.

- ID3D11Buffer: may be accessed through a device pointer.
- ID3D11Texture1D: individual subresources of the texture may be accessed via arrays
- ID3D11Texture2D: individual subresources of the texture may be accessed via arrays
- ID3D11Texture3D: individual subresources of the texture may be accessed via arrays

The `Flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used.
- CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.
- CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.

- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If `pD3DResource` is of incorrect type or is already registered then CUDA_ERROR_INVALID_HANDLE is returned. If `pD3DResource` cannot be registered then CUDA_ERROR_UNKNOWN is returned. If `Flags` is not one of the above specified value then CUDA_ERROR_INVALID_VALUE is returned.

Parameters:

pCudaResource - Returned graphics resource handle

pD3DResource - Direct3D resource to register

Flags - Parameters for resource registration

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsUnregisterResource, cuGraphicsMapResources, cuGraphicsSubResourceGetMappedArray, cuGraphicsResourceGetMappedPointer

5.58 Direct3D 11 Interoperability [DEPRECATED]

Functions

- CUresult cuD3D11CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, ID3D11Device *pD3DDevice)

  Create a CUDA context for interoperability with Direct3D 11.

- CUresult cuD3D11CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, ID3D11Device *pD3DDevice, CUdevice cudaDevice)

  Create a CUDA context for interoperability with Direct3D 11.

- CUresult cuD3D11GetDirect3DDevice (ID3D11Device **ppD3DDevice)

  Get the Direct3D 11 device against which the current CUDA context was created.

5.58.1 Detailed Description

This section describes deprecated Direct3D 11 interoperability functionality.

5.58.2 Function Documentation

5.58.2.1 CUresult cuD3D11CtxCreate (CUcontext * *pCtx*, CUdevice * *pCudaDevice*, unsigned int *Flags*, ID3D11Device * *pD3DDevice*)

Deprecated

This function is deprecated as of Cuda 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA context with a D3D11 device in order to achieve maximum interoperability performance.

Parameters:

pCtx - Returned newly created CUDA context

pCudaDevice - Returned pointer to the device on which the context was created

Flags - Context creation flags (see cuCtxCreate() for details)

pD3DDevice - Direct3D device to create interoperability context with

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D11GetDevice, cuGraphicsD3D11RegisterResource

5.58.2.2 CUresult cuD3D11CtxCreateOnDevice (CUcontext * *pCtx*, unsigned int *flags*, ID3D11Device * *pD3DDevice*, CUdevice *cudaDevice*)

Deprecated

This function is deprecated as of Cuda 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA context with a D3D11 device in order to achieve maximum interoperability performance.

Parameters:

pCtx - Returned newly created CUDA context

flags - Context creation flags (see cuCtxCreate() for details)

pD3DDevice - Direct3D device to create interoperability context with

cudaDevice - The CUDA device on which to create the context. This device must be among the devices returned when querying CU_D3D11_DEVICES_ALL from cuD3D11GetDevices.

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D11GetDevices, cuGraphicsD3D11RegisterResource

5.58.2.3 CUresult cuD3D11GetDirect3DDevice (ID3D11Device ** *ppD3DDevice*)

Deprecated

This function is deprecated as of Cuda 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA context with a D3D11 device in order to achieve maximum interoperability performance.

Parameters:

ppD3DDevice - Returned Direct3D device corresponding to CUDA context

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuD3D11GetDevice

5.59 VDPAU Interoperability

Functions

- CUresult cuGraphicsVDPAURegisterOutputSurface (CUgraphicsResource *pCudaResource, VdpOutputSurface vdpSurface, unsigned int flags)

  Registers a VDPAU VdpOutputSurface object.

- CUresult cuGraphicsVDPAURegisterVideoSurface (CUgraphicsResource *pCudaResource, VdpVideoSurface vdpSurface, unsigned int flags)

  Registers a VDPAU VdpVideoSurface object.

- CUresult cuVDPAUCtxCreate (CUcontext *pCtx, unsigned int flags, CUdevice device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

  Create a CUDA context for interoperability with VDPAU.

- CUresult cuVDPAUGetDevice (CUdevice *pDevice, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

  Gets the CUDA device associated with a VDPAU device.

5.59.1 Detailed Description

This section describes the VDPAU interoperability functions of the low-level CUDA driver application programming interface.

5.59.2 Function Documentation

5.59.2.1 CUresult cuGraphicsVDPAURegisterOutputSurface (CUgraphicsResource * *pCudaResource*, VdpOutputSurface *vdpSurface*, unsigned int *flags*)

Registers the VdpOutputSurface specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `pCudaResource`. The surface's intended usage is specified using `flags`, as follows:

- CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.
- CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

The VdpOutputSurface is presented as an array of subresources that may be accessed using pointers returned by cuGraphicsSubResourceGetMappedArray. The exact number of valid `arrayIndex` values depends on the VDPAU surface format. The mapping is shown in the table below. `mipLevel` must be 0.

VdpRGBAFormat	arrayIndex	Size	Format	Content
VDP_RGBA_FORMAT_B8G8R8A8	0	w x h	ARGB8	Entire surface
VDP_RGBA_FORMAT_R10G10B10A2	0	w x h	A2BGR10	Entire surface

Parameters:

pCudaResource - Pointer to the returned object handle

vdpSurface - The VdpOutputSurface to be registered

flags - Map flags

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuVDPAUCtxCreate, cuGraphicsVDPAURegisterVideoSurface, cuGraphicsUnregisterResource, cuGraphicsResourceSetMapFlags, cuGraphicsMapResources, cuGraphicsUnmapResources, cuGraphicsSubResourceGetMappedArray, cuVDPAUGetDevice

5.59.2.2 CUresult cuGraphicsVDPAURegisterVideoSurface (CUgraphicsResource * *pCudaResource*, VdpVideoSurface *vdpSurface*, unsigned int *flags*)

Registers the VdpVideoSurface specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `pCudaResource`. The surface's intended usage is specified using `flags`, as follows:

- CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.
- CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

The VdpVideoSurface is presented as an array of subresources that may be accessed using pointers returned by cuGraphicsSubResourceGetMappedArray. The exact number of valid `arrayIndex` values depends on the VDPAU surface format. The mapping is shown in the table below. `mipLevel` must be 0.

VdpChromaType	arrayIndex	Size	Format	Content
VDP_CHROMA_TYPE_420	0	w x h/2	R8	Top-field luma
	1	w x h/2	R8	Bottom-field luma
	2	w/2 x h/4	R8G8	Top-field chroma
	3	w/2 x h/4	R8G8	Bottom-field chroma
VDP_CHROMA_TYPE_422	0	w x h/2	R8	Top-field luma
	1	w x h/2	R8	Bottom-field luma
	2	w/2 x h/2	R8G8	Top-field chroma
	3	w/2 x h/2	R8G8	Bottom-field chroma

Parameters:

pCudaResource - Pointer to the returned object handle

vdpSurface - The VdpVideoSurface to be registered

flags - Map flags

Returns:

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuVDPAUCtxCreate, cuGraphicsVDPAURegisterOutputSurface, cuGraphicsUnregisterResource, cuGraphicsResourceSetMapFlags, cuGraphicsMapResources, cuGraphicsUnmapResources, cuGraphicsSubResourceGetMappedArray, cuVDPAUGetDevice

5.59.2.3 CUresult cuVDPAUCtxCreate (CUcontext * *pCtx*, unsigned int *flags*, CUdevice *device*, VdpDevice *vdpDevice*, VdpGetProcAddress * *vdpGetProcAddress*)

Creates a new CUDA context, initializes VDPAU interoperability, and associates the CUDA context with the calling thread. It must be called before performing any other VDPAU interoperability operations. It may fail if the needed VDPAU driver facilities are not available. For usage of the `flags` parameter, see cuCtxCreate().

Parameters:

pCtx - Returned CUDA context

flags - Options for CUDA context creation

device - Device on which to create the context

vdpDevice - The VdpDevice to interop with

vdpGetProcAddress - VDPAU's VdpGetProcAddress function pointer

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuGraphicsVDPAURegisterVideoSurface, cuGraphicsVDPAURegisterOutputSurface, cuGraphicsUnregisterResource, cuGraphicsResourceSetMapFlags, cuGraphicsMapResources, cuGraphicsUnmapResources, cuGraphicsSubResourceGetMappedArray, cuVDPAUGetDevice

5.59.2.4 CUresult cuVDPAUGetDevice (CUdevice * *pDevice*, VdpDevice *vdpDevice*, VdpGetProcAddress * *vdpGetProcAddress*)

Returns in `*pDevice` the CUDA device associated with a `vdpDevice`, if applicable.

Parameters:

pDevice - Device associated with vdpDevice

vdpDevice - A VdpDevice handle

vdpGetProcAddress - VDPAU's VdpGetProcAddress function pointer

Returns:

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_-ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Note:

Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxCreate, cuVDPAUCtxCreate, cuGraphicsVDPAURegisterVideoSurface, cuGraphicsVDPAURegisterOut-putSurface, cuGraphicsUnregisterResource, cuGraphicsResourceSetMapFlags, cuGraphicsMapResources, cu-GraphicsUnmapResources, cuGraphicsSubResourceGetMappedArray

5.60 Mathematical Functions

CUDA mathematical functions are always available in device code. Some functions are also available in host code as indicated.

Note that floating-point functions are overloaded for different argument types. For example, the log() function has the following prototypes:

```
double log(double x);
float log(float x);
float logf(float x);
```

5.61 Single Precision Mathematical Functions

Functions

- __device__ float acosf (float x)

  Calculate the arc cosine of the input argument.

- __device__ float acoshf (float x)

  Calculate the nonnegative arc hyperbolic cosine of the input argument.

- __device__ float asinf (float x)

  Calculate the arc sine of the input argument.

- __device__ float asinhf (float x)

  Calculate the arc hyperbolic sine of the input argument.

- __device__ float atan2f (float x, float y)

  Calculate the arc tangent of the ratio of first and second input arguments.

- __device__ float atanf (float x)

  Calculate the arc tangent of the input argument.

- __device__ float atanhf (float x)

  Calculate the arc hyperbolic tangent of the input argument.

- __device__ float cbrtf (float x)

  Calculate the cube root of the input argument.

- __device__ float ceilf (float x)

  Calculate ceiling of the input argument.

- __device__ float copysignf (float x, float y)

  Create value with given magnitude, copying sign of second value.

- __device__ float cosf (float x)

  Calculate the cosine of the input argument.

- __device__ float coshf (float x)

  Calculate the hyperbolic cosine of the input argument.

- __device__ float cospif (float x)

  Calculate the cosine of the input argument $\times \pi$.

- __device__ float erfcf (float x)

  Calculate the complementary error function of the input argument.

- __device__ float erfcinvf (float y)

  Calculate the inverse complementary error function of the input argument.

- __device__ float erfcxf (float x)

Calculate the scaled complementary error function of the input argument.

- __device__ float erff (float x)

  Calculate the error function of the input argument.

- __device__ float erfinvf (float y)

  Calculate the inverse error function of the input argument.

- __device__ float exp10f (float x)

  Calculate the base 10 exponential of the input argument.

- __device__ float exp2f (float x)

  Calculate the base 2 exponential of the input argument.

- __device__ float expf (float x)

  Calculate the base e *exponential of the input argument.*

- __device__ float expm1f (float x)

  Calculate the base e *exponential of the input argument, minus 1.*

- __device__ float fabsf (float x)

  Calculate the absolute value of its argument.

- __device__ float fdimf (float x, float y)

  Compute the positive difference between x *and* y.

- __device__ float fdividef (float x, float y)

  Divide two floating point values.

- __device__ float floorf (float x)

  Calculate the largest integer less than or equal to x.

- __device__ float fmaf (float x, float y, float z)

  Compute $x \times y + z$ *as a single operation.*

- __device__ float fmaxf (float x, float y)

  Determine the maximum numeric value of the arguments.

- __device__ float fminf (float x, float y)

  Determine the minimum numeric value of the arguments.

- __device__ float fmodf (float x, float y)

  Calculate the floating-point remainder of x / y.

- __device__ float frexpf (float x, int *nptr)

  Extract mantissa and exponent of a floating-point value.

- __device__ float hypotf (float x, float y)

  Calculate the square root of the sum of squares of two arguments.

- __device__ int ilogbf (float x)

  Compute the unbiased integer exponent of the argument.

- __device__ int isfinite (float a)

  Determine whether argument is finite.

- __device__ int isinf (float a)

  Determine whether argument is infinite.

- __device__ int isnan (float a)

  Determine whether argument is a NaN.

- __device__ float j0f (float x)

  Calculate the value of the Bessel function of the first kind of order 0 for the input argument.

- __device__ float j1f (float x)

  Calculate the value of the Bessel function of the first kind of order 1 for the input argument.

- __device__ float jnf (int n, float x)

  Calculate the value of the Bessel function of the first kind of order n for the input argument.

- __device__ float ldexpf (float x, int exp)

  Calculate the value of $x \cdot 2^{exp}$.

- __device__ float lgammaf (float x)

  Calculate the natural logarithm of the absolute value of the gamma function of the input argument.

- __device__ long long int llrintf (float x)

  Round input to nearest integer value.

- __device__ long long int llroundf (float x)

  Round to nearest integer value.

- __device__ float log10f (float x)

  Calculate the base 10 logarithm of the input argument.

- __device__ float log1pf (float x)

  Calculate the value of $log_e(1 + x)$ $\lfloor x \rfloor$.

- __device__ float log2f (float x)

  Calculate the base 2 logarithm of the input argument.

- __device__ float logbf (float x)

  Calculate the floating point representation of the exponent of the input argument.

- __device__ float logf (float x)

  Calculate the natural logarithm of the input argument.

- __device__ long int lrintf (float x)

  Round input to nearest integer value.

- __device__ long int lroundf (float x)

  Round to nearest integer value.

- __device__ float modff (float x, float *iptr)

  Break down the input argument into fractional and integral parts.

- __device__ float nanf (const char *tagp)

  Returns "Not a Number" value.

- __device__ float nearbyintf (float x)

  Round the input argument to the nearest integer.

- __device__ float nextafterf (float x, float y)

  Return next representable single-precision floating-point value afer argument.

- __device__ float normcdff (float y)

  Calculate the standard normal cumulative distribution function.

- __device__ float normcdfinvf (float y)

  Calculate the inverse of the standard normal cumulative distribution function.

- __device__ float powf (float x, float y)

  Calculate the value of first argument to the power of second argument.

- __device__ float rcbrtf (float x)

  Calculate reciprocal cube root function.

- __device__ float remainderf (float x, float y)

  Compute single-precision floating-point remainder.

- __device__ float remquof (float x, float y, int *quo)

  Compute single-precision floating-point remainder and part of quotient.

- __device__ float rintf (float x)

  Round input to nearest integer value in floating-point.

- __device__ float roundf (float x)

  Round to nearest integer value in floating-point.

- __device__ float rsqrtf (float x)

  Calculate the reciprocal of the square root of the input argument.

- __device__ float scalblnf (float x, long int n)

  Scale floating-point input by integer power of two.

- __device__ float scalbnf (float x, int n)

  Scale floating-point input by integer power of two.

- __device__ int signbit (float a)

Return the sign bit of the input.

- __device__ void sincosf (float x, float *sptr, float *cptr)

  Calculate the sine and cosine of the first input argument.

- __device__ void sincospif (float x, float *sptr, float *cptr)

  Calculate the sine and cosine of the first input argument $\times\pi$.

- __device__ float sinf (float x)

  Calculate the sine of the input argument.

- __device__ float sinhf (float x)

  Calculate the hyperbolic sine of the input argument.

- __device__ float sinpif (float x)

  Calculate the sine of the input argument $\times\pi$.

- __device__ float sqrtf (float x)

  Calculate the square root of the input argument.

- __device__ float tanf (float x)

  Calculate the tangent of the input argument.

- __device__ float tanhf (float x)

  Calculate the hyperbolic tangent of the input argument.

- __device__ float tgammaf (float x)

  Calculate the gamma function of the input argument.

- __device__ float truncf (float x)

  Truncate input argument to the integral part.

- __device__ float y0f (float x)

  Calculate the value of the Bessel function of the second kind of order 0 for the input argument.

- __device__ float y1f (float x)

  Calculate the value of the Bessel function of the second kind of order 1 for the input argument.

- __device__ float ynf (int n, float x)

  Calculate the value of the Bessel function of the second kind of order n for the input argument.

5.61.1 Detailed Description

This section describes single precision mathematical functions.

5.61.2 Function Documentation

5.61.2.1 __device__ float acosf (float *x*)

Calculate the principal value of the arc cosine of the input argument x.

Returns:

Result will be in radians, in the interval [0, π] for x inside [-1, +1].

- acosf(1) returns +0.
- acosf(x) returns NaN for x outside [-1, +1].

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.2 __device__ float acoshf (float *x*)

Calculate the nonnegative arc hyperbolic cosine of the input argument x (measured in radians).

Returns:

Result will be in the interval [0, $+\infty$].

- acoshf(1) returns 0.
- acoshf(x) returns NaN for x in the interval [$-\infty$, 1).

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.3 __device__ float asinf (float *x*)

Calculate the principal value of the arc sine of the input argument x.

Returns:

Result will be in radians, in the interval [- $\pi/2$, + $\pi/2$] for x inside [-1, +1].

- asinf(0) returns +0.
- asinf(x) returns NaN for x outside [-1, +1].

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.4 __device__ float asinhf (float *x*)

Calculate the arc hyperbolic sine of the input argument x (measured in radians).

Returns:

- asinhf(0) returns 1.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.5 __device__ float atan2f (float *x*, float *y*)

Calculate the principal value of the arc tangent of the ratio of first and second input arguments x / y. The quadrant of the result is determined by the signs of inputs x and y.

Returns:

Result will be in radians, in the interval [- π , + π].

- atan2f(0, 1) returns +0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.6 __device__ float atanf (float *x*)

Calculate the principal value of the arc tangent of the input argument x.

Returns:

Result will be in radians, in the interval [- $\pi/2$, + $\pi/2$].

- atanf(0) returns +0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.7 __device__ float atanhf (float *x*)

Calculate the arc hyperbolic tangent of the input argument x (measured in radians).

Returns:

- atanhf(± 0) returns ± 0 .
- atanhf(± 1) returns $\pm \infty$.
- atanhf(x) returns NaN for x outside interval [-1, 1].

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.8 __device__ float cbrtf (float *x*)

Calculate the cube root of x, $x^{1/3}$.

Returns:

Returns $x^{1/3}$.

- cbrtf(± 0) returns ± 0 .
- cbrtf($\pm \infty$) returns $\pm \infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.9 __device__ float ceilf (float *x*)

Compute the smallest integer value not less than x.

Returns:

Returns $\lceil x \rceil$ expressed as a floating-point number.

- ceilf(± 0) returns ± 0 .
- ceilf($\pm \infty$) returns $\pm \infty$.

5.61.2.10 __device__ float copysignf (float *x*, float *y*)

Create a floating-point value with the magnitude x and the sign of y.

Returns:

Returns a value with the magnitude of x and the sign of y.

5.61.2.11 __device__ float cosf (float *x*)

Calculate the cosine of the input argument x (measured in radians).

Returns:

- cosf(0) returns 1.
- cosf($\pm \infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.
This function is affected by the `-use_fast_math` compiler flag. See the CUDA C Programming Guide, Appendix C, Table C-3 for a complete list of functions affected.

5.61.2.12 __device__ float coshf (float *x*)

Calculate the hyperbolic cosine of the input argument x (measured in radians).

Returns:

- coshf(0) returns 1.
- coshf($\pm \infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.13 __device__ float cospif (float *x*)

Calculate the cosine of x $\times \pi$ (measured in radians), where x is the input argument.

Returns:

- cospif(± 0) returns 1.
- cospif($\pm \infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.14 __device__ float erfcf (float *x*)

Calculate the complementary error function of the input argument x, 1 - erf(x).

Returns:

- erfcf($-\infty$) returns 2.
- erfcf($+\infty$) returns +0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.15 __device__ float erfcinvf (float *y*)

Calculate the inverse complementary error function of the input argument y, for y in the interval [0, 2]. The inverse complementary error function find the value x that satisfies the equation y = erfc(x), for $0 \leq y \leq 2$, and $-\infty \leq x \leq \infty$.

Returns:

- erfcinvf(0) returns $+\infty$.
- erfcinvf(2) returns $-\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.16 __device__ float erfcxf (float *x*)

Calculate the scaled complementary error function of the input argument x, $e^{x^2} \cdot \text{erfc}(x)$.

Returns:

- erfcxf($-\infty$) returns $+\infty$
- erfcxf($+\infty$) returns +0
- erfcxf(x) returns $+\infty$ if the correctly calculated value is outside the single floating point range.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.17 __device__ float erff (float *x*)

Calculate the value of the error function for the input argument x, $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

Returns:

- erff(± 0) returns ± 0 .
- erff($\pm \infty$) returns ± 1 .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.18 __device__ float erfinvf (float *y*)

Calculate the inverse error function of the input argument y, for y in the interval [-1, 1]. The inverse error function finds the value x that satisfies the equation y = erf(x), for $-1 \leq y \leq 1$, and $-\infty \leq x \leq \infty$.

Returns:

- erfinvf(1) returns $+\infty$.
- erfinvf(-1) returns $-\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.19 __device__ float exp10f (float *x*)

Calculate the base 10 exponential of the input argument x.

Returns:

Returns 10^x .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.
This function is affected by the `-use_fast_math` compiler flag. See the CUDA C Programming Guide, Appendix C, Table C-3 for a complete list of functions affected.

5.61.2.20 __device__ float exp2f (float *x*)

Calculate the base 2 exponential of the input argument x.

Returns:

Returns 2^x .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.21 __device__ float expf (float *x*)

Calculate the base e exponential of the input argument x, e^x .

Returns:

Returns e^x .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.
This function is affected by the `-use_fast_math` compiler flag. See the CUDA C Programming Guide, Appendix C, Table C-3 for a complete list of functions affected.

5.61.2.22 __device__ float expm1f (float *x*)

Calculate the base e exponential of the input argument x, minus 1.

Returns:

Returns $e^x - 1$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.23 __device__ float fabsf (float *x*)

Calculate the absolute value of the input argument x.

Returns:

Returns the absolute value of its argument.

- fabs($\pm\infty$) returns $+\infty$.
- fabs(± 0) returns 0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.24 __device__ float fdimf (float *x*, float *y*)

Compute the positive difference between x and y. The positive difference is x - y when $x > y$ and +0 otherwise.

Returns:

Returns the positive difference between x and y.

- fdimf(x, y) returns x - y if $x > y$.
- fdimf(x, y) returns +0 if $x \leq y$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.25 __device__ float fdividef (float *x*, float *y*)

Compute x divided by y. If -use_fast_math is specified, use __fdividef() for higher performance, otherwise use normal division.

Returns:

Returns x / y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.
This function is affected by the -use_fast_math compiler flag. See the CUDA C Programming Guide, Appendix C, Table C-3 for a complete list of functions affected.

5.61.2.26 __device__ float floorf (float *x*)

Calculate the largest integer value which is less than or equal to x.

Returns:

Returns $log_e(1 + x)$ $\lfloor x \rfloor$ expressed as a floating-point number.

- floorf($\pm\infty$) returns $\pm\infty$.
- floorf(± 0) returns ± 0 .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.27 __device__ float fmaf (float *x*, float *y*, float *z*)

Compute the value of $x \times y + z$ as a single ternary operation. After computing the value to infinite precision, the value is rounded once.

Returns:

Returns the rounded value of $x \times y + z$ as a single operation.

- fmaf($\pm\infty$, ± 0 , z) returns NaN.
- fmaf(± 0 , $\pm\infty$, z) returns NaN.
- fmaf(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- fmaf(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.28 __device__ float fmaxf (float *x*, float *y*)

Determines the maximum numeric value of the arguments x and y. Treats NaN arguments as missing data. If one argument is a NaN and the other is legitimate numeric value, the numeric value is chosen.

Returns:

Returns the maximum numeric values of the arguments x and y.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.29 __device__ float fminf (float *x*, float *y*)

Determines the minimum numeric value of the arguments x and y. Treats NaN arguments as missing data. If one argument is a NaN and the other is legitimate numeric value, the numeric value is chosen.

Returns:

Returns the minimum numeric values of the arguments x and y.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.30 __device__ float fmodf (float *x*, float *y*)

Calculate the floating-point remainder of x / y. The absolute value of the computed value is always less than y's absolute value and will have the same sign as x.

Returns:

- Returns the floating point remainder of x / y.
- fmodf(± 0 , y) returns ± 0 if y is not zero.
- fmodf(x, y) returns NaN and raised an invalid floating point exception if x is $\pm\infty$ or y is zero.
- fmodf(x, y) returns zero if y is zero or the result would overflow.
- fmodf(x, $\pm\infty$) returns x if x is finite.
- fmodf(x, 0) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.31 __device__ float frexpf (float *x*, int * *nptr*)

Decomposes the floating-point value x into a component m for the normalized fraction element and another term n for the exponent. The absolute value of m will be greater than or equal to 0.5 and less than 1.0 or it will be equal to 0; $x = m \cdot 2^n$. The integer exponent n will be stored in the location to which nptr points.

Returns:

Returns the fractional component m.

- frexp(0, nptr) returns 0 for the fractional component and zero for the integer component.
- frexp(± 0 , nptr) returns ± 0 and stores zero in the location pointed to by nptr.
- frexp($\pm \infty$, nptr) returns $\pm \infty$ and stores an unspecified value in the location to which nptr points.
- frexp(NaN, y) returns a NaN and stores an unspecified value in the location to which nptr points.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.32 __device__ float hypotf (float *x*, float *y*)

Calculates the length of the hypotenuse of a right triangle whose two sides have lengths x and y without undue overflow or underflow.

Returns:

Returns the length of the hypotenuse $\sqrt{x^2 + y^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.33 __device__ int ilogbf (float *x*)

Calculates the unbiased integer exponent of the input argument x.

Returns:

- If successful, returns the unbiased exponent of the argument.
- ilogbf(0) returns INT_MIN.
- ilogbf(NaN) returns NaN.
- ilogbf(x) returns INT_MAX if x is ∞ or the correct value is greater than INT_MAX.
- ilogbf(x) return INT_MIN if the correct value is less than INT_MIN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.34 __device__ int isfinite (float *a*)

Determine whether the floating-point value a is a finite value (zero, subnormal, or normal and not infinity or NaN).

Returns:

Returns a nonzero value if and only if a is a finite value.

5.61.2.35 __device__ int isinf (float *a*)

Determine whether the floating-point value a is an infinite value (positive or negative).

Returns:

Returns a nonzero value if and only if a is a infinite value.

5.61.2.36 __device__ int isnan (float *a*)

Determine whether the floating-point value a is a NaN.

Returns:

Returns a nonzero value if and only if a is a NaN value.

5.61.2.37 __device__ float j0f (float *x*)

Calculate the value of the Bessel function of the first kind of order 0 for the input argument x, $J_0(x)$.

Returns:

Returns the value of the Bessel function of the first kind of order 0.

- j0f($\pm\infty$) returns +0.
- j0f(NaN) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.38 __device__ float j1f (float *x*)

Calculate the value of the Bessel function of the first kind of order 1 for the input argument x, $J_1(x)$.

Returns:

Returns the value of the Bessel function of the first kind of order 1.

- j1f(± 0) returns ± 0 .
- j1f($\pm\infty$) returns +0.
- j1f(NaN) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.39 __device__ float jnf (int *n*, float *x*)

Calculate the value of the Bessel function of the first kind of order n for the input argument x, $J_n(x)$.

Returns:

Returns the value of the Bessel function of the first kind of order n.

- jnf(n, NaN) returns NaN.
- jnf(n, x) returns NaN for $n < 0$.
- jnf(n, $+\infty$) returns +0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.40 __device__ float ldexpf (float *x*, int *exp*)

Calculate the value of $x \cdot 2^{exp}$ of the input arguments x and exp.

Returns:

- ldexpf(x) returns $\pm\infty$ if the correctly calculated value is outside the single floating point range.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.41 __device__ float lgammaf (float *x*)

Calculate the natural logarithm of the absolute value of the gamma function of the input argument x, namely the value of $log_e| \int_0^\infty e^{-t}t^{x-1}dt|$.

Returns:

- lgammaf(1) returns +0.
- lgammaf(2) returns +0.
- lgammaf(x) returns $\pm\infty$ if the correctly calculated value is outside the single floating point range.
- lgammaf(x) returns $+\infty$ if $x \leq 0$.
- lgammaf($-\infty$) returns $-\infty$.
- lgammaf($+\infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.42 __device__ long long int llrintf (float *x*)

Round x to the nearest integer value, with halfway cases rounded towards zero. If the result is outside the range of the return type, the result is undefined.

Returns:

Returns rounded integer value.

5.61.2.43 __device__ long long int llroundf (float *x*)

Round x to the nearest integer value, with halfway cases rounded away from zero. If the result is outside the range of the return type, the result is undefined.

Returns:

Returns rounded integer value.

Note:

This function may be slower than alternate rounding methods. See llrintf().

5.61.2.44 __device__ float log10f (float *x*)

Calculate the base 10 logarithm of the input argument x.

Returns:

- log10f(± 0) returns $-\infty$.
- log10f(1) returns +0.
- log10f(x) returns NaN for x < 0.
- log10f($+\infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.45 __device__ float log1pf (float *x*)

Calculate the value of $log_e(1 + x)$ $\lfloor x \rfloor$ of the input argument x.

Returns:

- log1pf(± 0) returns $-\infty$.
- log1pf(-1) returns +0.
- log1pf(x) returns NaN for x $<$ -1.
- log1pf($+\infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.46 __device__ float log2f (float *x*)

Calculate the base 2 logarithm of the input argument x.

Returns:

- log2f(± 0) returns $-\infty$.

- log2f(1) returns +0.
- log2f(x) returns NaN for $x < 0$.
- log2f($+\infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.47 __device__ float logbf (float *x*)

Calculate the floating point representation of the exponent of the input argument x.

Returns:

- logbf ± 0 returns $-\infty$
- logbf $+\infty$ returns $+\infty$

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.48 __device__ float logf (float *x*)

Calculate the natural logarithm of the input argument x.

Returns:

- logf(± 0) returns $-\infty$.
- logf(1) returns +0.
- logf(x) returns NaN for $x < 0$.
- logf($+\infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.49 __device__ long int lrintf (float *x*)

Round x to the nearest integer value, with halfway cases rounded towards zero. If the result is outside the range of the return type, the result is undefined.

Returns:

Returns rounded integer value.

5.61.2.50 __device__ long int lroundf (float *x*)

Round `x` to the nearest integer value, with halfway cases rounded away from zero. If the result is outside the range of the return type, the result is undefined.

Returns:

Returns rounded integer value.

Note:

This function may be slower than alternate rounding methods. See lrintf().

5.61.2.51 __device__ float modff (float *x*, float * *iptr*)

Break down the argument `x` into fractional and integral parts. The integral part is stored in the argument `iptr`. Fractional and integral parts are given the same sign as the argument `x`.

Returns:

- modff($\pm x$, `iptr`) returns a result with the same sign as `x`.
- modff($\pm\infty$, `iptr`) returns ± 0 and stores $\pm\infty$ in the object pointed to by `iptr`.
- modff(NaN, `iptr`) stores a NaN in the object pointed to by `iptr` and returns a NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.52 __device__ float nanf (const char * *tagp*)

Return a representation of a quiet NaN. Argument `tagp` selects one of the possible representations.

Returns:

- nanf(`tagp`) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.53 __device__ float nearbyintf (float *x*)

Round argument `x` to an integer value in single precision floating-point format.

Returns:

- nearbyintf(± 0) returns ± 0 .
- nearbyintf($\pm\infty$) returns $\pm\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.54 __device__ float nextafterf (float *x*, float *y*)

Calculate the next representable single-precision floating-point value following x in the direction of y. For example, if y is greater than x, nextafterf() returns the smallest representable number greater than x

Returns:

- nextafterf($\pm\infty$, y) returns $\pm\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.55 __device__ float normcdff (float *y*)

Calculate the cumulative distribution function of the standard normal distribution for input argument y, $\Phi(y)$.

Returns:

- normcdff($+\infty$) returns 1
- normcdff($-\infty$) returns +0

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.56 __device__ float normcdfinvf (float *y*)

Calculate the inverse of the standard normal cumulative distribution function for input argument y, $\Phi^{-1}(y)$. The function is defined for input values in the interval $(0, 1)$.

Returns:

- normcdfinvf(0) returns $-\infty$.
- normcdfinvf(1) returns $+\infty$.
- normcdfinvf(x) returns NaN if x is not in the interval [0,1].

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.57 __device__ float powf (float *x*, float *y*)

Calculate the value of x to the power of y.

Returns:

- powf(± 0 , y) returns $\pm\infty$ for y an integer less than 0.
- powf(± 0 , y) returns ± 0 for y an odd integer greater than 0.
- powf(± 0 , y) returns +0 for $y > 0$ and not and odd integer.
- powf(-1, $\pm\infty$) returns 1.

- powf(+1, y) returns 1 for any y, even a NaN.
- powf(x, ± 0) returns 1 for any x, even a NaN.
- powf(x, y) returns a NaN for finite x < 0 and finite non-integer y.
- powf(x, $-\infty$) returns $+\infty$ for $|x| < 1$.
- powf(x, $-\infty$) returns +0 for $|x| > 1$.
- powf(x, $+\infty$) returns +0 for $|x| < 1$.
- powf(x, $+\infty$) returns $+\infty$ for $|x| > 1$.
- powf($-\infty$, y) returns -0 for y an odd integer less than 0.
- powf($-\infty$, y) returns +0 for y < 0 and not an odd integer.
- powf($-\infty$, y) returns $-\infty$ for y an odd integer greater than 0.
- powf($-\infty$, y) returns $+\infty$ for y > 0 and not an odd integer.
- powf($+\infty$, y) returns +0 for y < 0.
- powf($+\infty$, y) returns $+\infty$ for y > 0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.58 __device__ float rcbrtf (float *x*)

Calculate reciprocal cube root function of x

Returns:

- rcbrt(± 0) returns $\pm\infty$.
- rcbrt($\pm\infty$) returns ± 0 .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.59 __device__ float remainderf (float *x*, float *y*)

Compute single-precision floating-point remainder r of dividing x by y for nonzero y. Thus $r = x - ny$. The value n is the integer value nearest $\frac{x}{y}$. In the case when $|n - \frac{x}{y}| = \frac{1}{2}$, the even n value is chosen.

Returns:

- remainderf(x, 0) returns NaN.
- remainderf($\pm\infty$, y) returns NaN.
- remainderf(x, $\pm\infty$) returns x for finite x.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.60 __device__ float remquof (float *x*, float *y*, int * *quo*)

Compute a double-precision floating-point remainder in the same way as the remainderf() function. Argument quo returns part of quotient upon division of x by y. Value quo has the same sign as $\frac{x}{y}$ and may not be the exact quotient but agrees with the exact quotient in the low order 3 bits.

Returns:

Returns the remainder.

- remquof(x, 0, quo) returns NaN.
- remquof($\pm\infty$, y, quo) returns NaN.
- remquof(x, $\pm\infty$, quo) returns x.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.61 __device__ float rintf (float *x*)

Round x to the nearest integer value in floating-point format, with halfway cases rounded towards zero.

Returns:

Returns rounded integer value.

5.61.2.62 __device__ float roundf (float *x*)

Round x to the nearest integer value in floating-point format, with halfway cases rounded away from zero.

Returns:

Returns rounded integer value.

Note:

This function may be slower than alternate rounding methods. See rintf().

5.61.2.63 __device__ float rsqrtf (float *x*)

Calculate the reciprocal of the nonnegative square root of x, $1/\sqrt{x}$.

Returns:

Returns $1/\sqrt{x}$.

- rsqrtf($+\infty$) returns +0.
- rsqrtf(± 0) returns $\pm\infty$.
- rsqrtf(x) returns NaN if x is less than 0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.64 __device__ float scalblnf (float *x*, long int *n*)

Scale x by 2^n by efficient manipulation of the floating-point exponent.

Returns:

Returns $x * 2^n$.

- scalblnf(± 0 , n) returns ± 0 .
- scalblnf(x, 0) returns x.
- scalblnf($\pm\infty$, n) returns $\pm\infty$.

5.61.2.65 __device__ float scalbnf (float *x*, int *n*)

Scale x by 2^n by efficient manipulation of the floating-point exponent.

Returns:

Returns $x * 2^n$.

- scalbnf(± 0 , n) returns ± 0 .
- scalbnf(x, 0) returns x.
- scalbnf($\pm\infty$, n) returns $\pm\infty$.

5.61.2.66 __device__ int signbit (float *a*)

Determine whether the floating-point value a is negative.

Returns:

Returns a nonzero value if and only if a is negative. Reports the sign bit of all values including infinities, zeros, and NaNs.

5.61.2.67 __device__ void sincosf (float *x*, float * *sptr*, float * *cptr*)

Calculate the sine and cosine of the first input argument x (measured in radians). The results for sine and cosine are written into the second argument, sptr, and, respectively, third argument, cptr.

Returns:

- none

See also:

sinf() and cosf().

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.
This function is affected by the -use_fast_math compiler flag. See the CUDA C Programming Guide, Appendix C, Table C-3 for a complete list of functions affected.

5.61.2.68 __device__ void sincospif (float *x*, float * *sptr*, float * *cptr*)

Calculate the sine and cosine of the first input argument, x (measured in radians), $\times\pi$. The results for sine and cosine are written into the second argument, sptr, and, respectively, third argument, cptr.

Returns:

- none

See also:

sinpif() and cospif().

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.69 __device__ float sinf (float *x*)

Calculate the sine of the input argument x (measured in radians).

Returns:

- sinf(± 0) returns ± 0 .
- sinf($\pm\infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.
This function is affected by the -use_fast_math compiler flag. See the CUDA C Programming Guide, Appendix C, Table C-3 for a complete list of functions affected.

5.61.2.70 __device__ float sinhf (float *x*)

Calculate the hyperbolic sine of the input argument x (measured in radians).

Returns:

- sinhf(± 0) returns ± 0 .
- sinhf($\pm\infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.71 __device__ float sinpif (float *x*)

Calculate the sine of x $\times\pi$ (measured in radians), where x is the input argument.

Returns:

- sinpif(± 0) returns ± 0 .
- sinpif($\pm\infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.72 __device__ float sqrtf (float *x*)

Calculate the nonnegative square root of x, $\sqrt{x}$.

Returns:

Returns $\sqrt{x}$.

- sqrtf(± 0) returns ± 0 .
- sqrtf($+\infty$) returns $+\infty$.
- sqrtf(x) returns NaN if x is less than 0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.73 __device__ float tanf (float *x*)

Calculate the tangent of the input argument x (measured in radians).

Returns:

- tanf(± 0) returns ± 0 .
- tanf($\pm\infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.
This function is affected by the `-use_fast_math` compiler flag. See the CUDA C Programming Guide, Appendix C, Table C-3 for a complete list of functions affected.

5.61.2.74 __device__ float tanhf (float *x*)

Calculate the hyperbolic tangent of the input argument x (measured in radians).

Returns:

- tanhf(± 0) returns ± 0 .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.75 __device__ float tgammaf (float *x*)

Calculate the gamma function of the input argument x, namely the value of $\int_0^\infty e^{-t} t^{x-1} dt$.

Returns:

- tgammaf(± 0) returns $\pm\infty$.
- tgammaf(2) returns +0.

- tgammaf(x) returns $\pm\infty$ if the correctly calculated value is outside the single floating point range.
- tgammaf(x) returns NaN if x < 0.
- tgammaf($-\infty$) returns NaN.
- tgammaf($+\infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.76 __device__ float truncf (float *x*)

Round x to the nearest integer value that does not exceed x in magnitude.

Returns:

Returns truncated integer value.

5.61.2.77 __device__ float y0f (float *x*)

Calculate the value of the Bessel function of the second kind of order 0 for the input argument x, $Y_0(x)$.

Returns:

Returns the value of the Bessel function of the second kind of order 0.

- y0f(0) returns $-\infty$.
- y0f(x) returns NaN for x < 0.
- y0f($+\infty$) returns +0.
- y0f(NaN) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.78 __device__ float y1f (float *x*)

Calculate the value of the Bessel function of the second kind of order 1 for the input argument x, $Y_1(x)$.

Returns:

Returns the value of the Bessel function of the second kind of order 1.

- y1f(0) returns $-\infty$.
- y1f(x) returns NaN for x < 0.
- y1f($+\infty$) returns +0.
- y1f(NaN) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.61.2.79 __device__ float ynf (int *n*, float *x*)

Calculate the value of the Bessel function of the second kind of order n for the input argument x, $Y_n(x)$.

Returns:

Returns the value of the Bessel function of the second kind of order n.

- ynf(n, x) returns NaN for $n < 0$.
- ynf(n, 0) returns $-\infty$.
- ynf(n, x) returns NaN for $x < 0$.
- ynf(n, $+\infty$) returns +0.
- ynf(n, NaN) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.62 Double Precision Mathematical Functions

Functions

- __device__ double acos (double x)

  Calculate the arc cosine of the input argument.

- __device__ double acosh (double x)

  Calculate the nonnegative arc hyperbolic cosine of the input argument.

- __device__ double asin (double x)

  Calculate the arc sine of the input argument.

- __device__ double asinh (double x)

  Calculate the arc hyperbolic sine of the input argument.

- __device__ double atan (double x)

  Calculate the arc tangent of the input argument.

- __device__ double atan2 (double x, double y)

  Calculate the arc tangent of the ratio of first and second input arguments.

- __device__ double atanh (double x)

  Calculate the arc hyperbolic tangent of the input argument.

- __device__ double cbrt (double x)

  Calculate the cube root of the input argument.

- __device__ double ceil (double x)

  Calculate ceiling of the input argument.

- __device__ double copysign (double x, double y)

  Create value with given magnitude, copying sign of second value.

- __device__ double cos (double x)

  Calculate the cosine of the input argument.

- __device__ double cosh (double x)

  Calculate the hyperbolic cosine of the input argument.

- __device__ double cospi (double x)

  Calculate the cosine of the input argument $\times \pi$.

- __device__ double erf (double x)

  Calculate the error function of the input argument.

- __device__ double erfc (double x)

  Calculate the complementary error function of the input argument.

- __device__ double erfcinv (double y)

Calculate the inverse complementary error function of the input argument.

- __device__ double erfcx (double x)

  Calculate the scaled complementary error function of the input argument.

- __device__ double erfinv (double y)

  Calculate the inverse error function of the input argument.

- __device__ double exp (double x)

  Calculate the base e exponential of the input argument.

- __device__ double exp10 (double x)

  Calculate the base 10 exponential of the input argument.

- __device__ double exp2 (double x)

  Calculate the base 2 exponential of the input argument.

- __device__ double expm1 (double x)

  Calculate the base e exponential of the input argument, minus 1.

- __device__ double fabs (double x)

  Calculate the absolute value of the input argument.

- __device__ double fdim (double x, double y)

  Compute the positive difference between x *and* y.

- __device__ double floor (double x)

  Calculate the largest integer less than or equal to x.

- __device__ double fma (double x, double y, double z)

  Compute $x \times y + z$ as a single operation.

- __device__ double fmax (double, double)

  Determine the maximum numeric value of the arguments.

- __device__ double fmin (double x, double y)

  Determine the minimum numeric value of the arguments.

- __device__ double fmod (double x, double y)

  Calculate the floating-point remainder of x / y.

- __device__ double frexp (double x, int *nptr)

  Extract mantissa and exponent of a floating-point value.

- __device__ double hypot (double x, double y)

  Calculate the square root of the sum of squares of two arguments.

- __device__ int ilogb (double x)

  Compute the unbiased integer exponent of the argument.

- __device__ int isfinite (double a)

  Determine whether argument is finite.

- __device__ int isinf (double a)

  Determine whether argument is infinite.

- __device__ int isnan (double a)

  Determine whether argument is a NaN.

- __device__ double j0 (double x)

  Calculate the value of the Bessel function of the first kind of order 0 for the input argument.

- __device__ double j1 (double x)

  Calculate the value of the Bessel function of the first kind of order 1 for the input argument.

- __device__ double jn (int n, double x)

  Calculate the value of the Bessel function of the first kind of order n for the input argument.

- __device__ double ldexp (double x, int exp)

  Calculate the value of $x \cdot 2^{exp}$.

- __device__ double lgamma (double x)

  Calculate the natural logarithm of the absolute value of the gamma function of the input argument.

- __device__ long long int llrint (double x)

  Round input to nearest integer value.

- __device__ long long int llround (double x)

  Round to nearest integer value.

- __device__ double log (double x)

  Calculate the base e *logarithm of the input argument.*

- __device__ double log10 (double x)

  Calculate the base 10 logarithm of the input argument.

- __device__ double log1p (double x)

  Calculate the value of $log_e(1 + x)$ $\lfloor x \rfloor$.

- __device__ double log2 (double x)

  Calculate the base 2 logarithm of the input argument.

- __device__ double logb (double x)

  Calculate the floating point representation of the exponent of the input argument.

- __device__ long int lrint (double x)

  Round input to nearest integer value.

- __device__ long int lround (double x)

  Round to nearest integer value.

- __device__ double modf (double x, double *iptr)

  Break down the input argument into fractional and integral parts.

- __device__ double nan (const char *tagp)

  Returns "Not a Number" value.

- __device__ double nearbyint (double x)

  Round the input argument to the nearest integer.

- __device__ double nextafter (double x, double y)

  Return next representable double-precision floating-point value after argument.

- __device__ double normcdf (double y)

  Calculate the standard normal cumulative distribution function.

- __device__ double normcdfinv (double y)

  Calculate the inverse of the standard normal cumulative distribution function.

- __device__ double pow (double x, double y)

  Calculate the value of first argument to the power of second argument.

- __device__ double rcbrt (double x)

  Calculate reciprocal cube root function.

- __device__ double remainder (double x, double y)

  Compute double-precision floating-point remainder.

- __device__ double remquo (double x, double y, int *quo)

  Compute double-precision floating-point remainder and part of quotient.

- __device__ double rint (double x)

  Round to nearest integer value in floating-point.

- __device__ double round (double x)

  Round to nearest integer value in floating-point.

- __device__ double rsqrt (double x)

  Calculate the reciprocal of the square root of the input argument.

- __device__ double scalbln (double x, long int n)

  Scale floating-point input by integer power of two.

- __device__ double scalbn (double x, int n)

  Scale floating-point input by integer power of two.

- __device__ int signbit (double a)

  Return the sign bit of the input.

- __device__ double sin (double x)

Calculate the sine of the input argument.

- __device__ void sincos (double x, double *sptr, double *cptr)

  Calculate the sine and cosine of the first input argument.

- __device__ void sincospi (double x, double *sptr, double *cptr)

  Calculate the sine and cosine of the first input argument $\times\pi$.

- __device__ double sinh (double x)

  Calculate the hyperbolic sine of the input argument.

- __device__ double sinpi (double x)

  Calculate the sine of the input argument $\times\pi$.

- __device__ double sqrt (double x)

  Calculate the square root of the input argument.

- __device__ double tan (double x)

  Calculate the tangent of the input argument.

- __device__ double tanh (double x)

  Calculate the hyperbolic tangent of the input argument.

- __device__ double tgamma (double x)

  Calculate the gamma function of the input argument.

- __device__ double trunc (double x)

  Truncate input argument to the integral part.

- __device__ double y0 (double x)

  Calculate the value of the Bessel function of the second kind of order 0 for the input argument.

- __device__ double y1 (double x)

  Calculate the value of the Bessel function of the second kind of order 1 for the input argument.

- __device__ double yn (int n, double x)

  Calculate the value of the Bessel function of the second kind of order n for the input argument.

5.62.1 Detailed Description

This section describes double precision mathematical functions.

5.62.2 Function Documentation

5.62.2.1 __device__ double acos (double *x*)

Calculate the principal value of the arc cosine of the input argument `x`.

Returns:

Result will be in radians, in the interval [0, π] for x inside [-1, +1].

- acos(1) returns +0.
- acos(x) returns NaN for x outside [-1, +1].

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.2 __device__ double acosh (double *x*)

Calculate the nonnegative arc hyperbolic cosine of the input argument x (measured in radians).

Returns:

Result will be in the interval [0, $+\infty$].

- acosh(1) returns 0.
- acosh(x) returns NaN for x in the interval [$-\infty$, 1).

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.3 __device__ double asin (double *x*)

Calculate the principal value of the arc sine of the input argument x.

Returns:

Result will be in radians, in the interval [- π /2, + π /2] for x inside [-1, +1].

- asin(0) returns +0.
- asin(x) returns NaN for x outside [-1, +1].

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.4 __device__ double asinh (double *x*)

Calculate the arc hyperbolic sine of the input argument x (measured in radians).

Returns:

- asinh(0) returns 1.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.5 __device__ double atan (double *x*)

Calculate the principal value of the arc tangent of the input argument x.

Returns:

Result will be in radians, in the interval [- π /2, + π /2].

- atan(0) returns +0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.6 __device__ double atan2 (double *x*, double *y*)

Calculate the principal value of the arc tangent of the ratio of first and second input arguments x / y. The quadrant of the result is determined by the signs of inputs x and y.

Returns:

Result will be in radians, in the interval [- π /, + π].

- atan2(0, 1) returns +0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.7 __device__ double atanh (double *x*)

Calculate the arc hyperbolic tangent of the input argument x (measured in radians).

Returns:

- atanh(± 0) returns ± 0 .
- atanh(± 1) returns $\pm \infty$.
- atanh(x) returns NaN for x outside interval [-1, 1].

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.8 __device__ double cbrt (double *x*)

Calculate the cube root of x, $x^{1/3}$.

Returns:

Returns $x^{1/3}$.

- cbrt(± 0) returns ± 0 .
- cbrt($\pm \infty$) returns $\pm \infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.9 __device__ double ceil (double *x*)

Compute the smallest integer value not less than x.

Returns:

Returns $\lceil x \rceil$ expressed as a floating-point number.

- ceil(± 0) returns ± 0 .
- ceil($\pm \infty$) returns $\pm \infty$.

5.62.2.10 __device__ double copysign (double *x*, double *y*)

Create a floating-point value with the magnitude x and the sign of y.

Returns:

Returns a value with the magnitude of x and the sign of y.

5.62.2.11 __device__ double cos (double *x*)

Calculate the cosine of the input argument x (measured in radians).

Returns:

- cos(± 0) returns 1.
- cos($\pm \infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.12 __device__ double cosh (double *x*)

Calculate the hyperbolic cosine of the input argument x (measured in radians).

Returns:

- cosh(0) returns 1.
- cosh($\pm \infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.13 __device__ double cospi (double *x*)

Calculate the cosine of x $\times \pi$ (measured in radians), where x is the input argument.

Returns:

- cospi(± 0) returns 1.

- cospi($\pm\infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.14 __device__ double erf (double *x*)

Calculate the value of the error function for the input argument x, $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

Returns:

- erf(± 0) returns ± 0 .
- erf($\pm\infty$) returns ± 1 .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.15 __device__ double erfc (double *x*)

Calculate the complementary error function of the input argument x, 1 - erf(x).

Returns:

- erfc($-\infty$) returns 2.
- erfc($+\infty$) returns +0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.16 __device__ double erfcinv (double *y*)

Calculate the inverse complementary error function of the input argument y, for y in the interval [0, 2]. The inverse complementary error function find the value x that satisfies the equation y = erfc(x), for $0 \leq y \leq 2$, and $-\infty \leq x \leq \infty$.

Returns:

- erfcinv(0) returns $+\infty$.
- erfcinv(2) returns $-\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.17 __device__ double erfcx (double *x*)

Calculate the scaled complementary error function of the input argument x, $e^{x^2} \cdot \text{erfc}(x)$.

Returns:

- erfcx($-\infty$) returns $+\infty$
- erfcx($+\infty$) returns +0
- erfcx(x) returns $+\infty$ if the correctly calculated value is outside the double floating point range.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.18 __device__ double erfinv (double *y*)

Calculate the inverse error function of the input argument y, for y in the interval [-1, 1]. The inverse error function finds the value x that satisfies the equation y = erf(x), for $-1 \leq y \leq 1$, and $-\infty \leq x \leq \infty$.

Returns:

- erfinv(1) returns $+\infty$.
- erfinv(-1) returns $-\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.19 __device__ double exp (double *x*)

Calculate the base e exponential of the input argument x.

Returns:

Returns e^x .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.20 __device__ double exp10 (double *x*)

Calculate the base 10 exponential of the input argument x.

Returns:

Returns 10^x .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.21 __device__ double exp2 (double *x*)

Calculate the base 2 exponential of the input argument x.

Returns:

Returns 2^x .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.22 __device__ double expm1 (double *x*)

Calculate the base e exponential of the input argument x, minus 1.

Returns:

Returns $e^x - 1$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.23 __device__ double fabs (double *x*)

Calculate the absolute value of the input argument x.

Returns:

Returns the absolute value of the input argument.

- fabs($\pm\infty$) returns $+\infty$.
- fabs(± 0) returns 0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.24 __device__ double fdim (double *x*, double *y*)

Compute the positive difference between x and y. The positive difference is x - y when x $>$ y and +0 otherwise.

Returns:

Returns the positive difference between x and y.

- fdim(x, y) returns x - y if x $>$ y.
- fdim(x, y) returns +0 if x $\leq$ y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.62.2.25 __device__ double floor (double *x*)

Calculates the largest integer value which is less than or equal to x.

Returns:

Returns $log_e(1 + x)$ $\lfloor x \rfloor$ expressed as a floating-point number.

- floor($\pm\infty$) returns $\pm\infty$.
- floor(± 0) returns ± 0 .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.26 __device__ double fma (double *x*, double *y*, double *z*)

Compute the value of $x \times y + z$ as a single ternary operation. After computing the value to infinite precision, the value is rounded once.

Returns:

Returns the rounded value of $x \times y + z$ as a single operation.

- fma($\pm\infty$, ± 0 , z) returns NaN.
- fma(± 0 , $\pm\infty$, z) returns NaN.
- fma(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- fma(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.27 __device__ double fmax (double, double)

Determines the maximum numeric value of the arguments x and y. Treats NaN arguments as missing data. If one argument is a NaN and the other is legitimate numeric value, the numeric value is chosen.

Returns:

Returns the maximum numeric values of the arguments x and y.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.28 __device__ double fmin (double *x*, double *y*)

Determines the minimum numeric value of the arguments x and y. Treats NaN arguments as missing data. If one argument is a NaN and the other is legitimate numeric value, the numeric value is chosen.

Returns:

Returns the minimum numeric values of the arguments x and y.

- If both arguments are NaN, returns NaN.
- If one argument is NaN, returns the numeric argument.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.29 __device__ double fmod (double *x*, double *y*)

Calculate the floating-point remainder of x / y. The absolute value of the computed value is always less than y's absolute value and will have the same sign as x.

Returns:

- Returns the floating point remainder of x / y.
- fmod(± 0 , y) returns ± 0 if y is not zero.
- fmod(x, y) returns NaN and raised an invalid floating point exception if x is $\pm\infty$ or y is zero.
- fmod(x, y) returns zero if y is zero or the result would overflow.
- fmod(x, $\pm\infty$) returns x if x is finite.
- fmod(x, 0) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.30 __device__ double frexp (double *x*, int * *nptr*)

Decompose the floating-point value x into a component m for the normalized fraction element and another term n for the exponent. The absolute value of m will be greater than or equal to 0.5 and less than 1.0 or it will be equal to 0; $x = m \cdot 2^n$. The integer exponent n will be stored in the location to which nptr points.

Returns:

Returns the fractional component m.

- frexp(0, nptr) returns 0 for the fractional component and zero for the integer component.
- frexp(± 0 , nptr) returns ± 0 and stores zero in the location pointed to by nptr.
- frexp($\pm\infty$, nptr) returns $\pm\infty$ and stores an unspecified value in the location to which nptr points.
- frexp(NaN, y) returns a NaN and stores an unspecified value in the location to which nptr points.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.31 __device__ double hypot (double *x*, double *y*)

Calculate the length of the hypotenuse of a right triangle whose two sides have lengths x and y without undue overflow or underflow.

Returns:

Returns the length of the hypotenuse $\sqrt{x^2+y^2}$. If the correct value would overflow, returns $+\infty$. If the correct value would underflow, returns 0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.32 __device__ int ilogb (double *x*)

Calculates the unbiased integer exponent of the input argument x.

Returns:

- If successful, returns the unbiased exponent of the argument.
- ilogb(0) returns INT_MIN.
- ilogb(NaN) returns NaN.
- ilogb(x) returns INT_MAX if x is ∞ or the correct value is greater than INT_MAX.
- ilogb(x) return INT_MIN if the correct value is less than INT_MIN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.33 __device__ int isfinite (double *a*)

Determine whether the floating-point value a is a finite value (zero, subnormal, or normal and not infinity or NaN).

Returns:

Returns a nonzero value if and only if a is a finite value.

5.62.2.34 __device__ int isinf (double *a*)

Determine whether the floating-point value a is an infinite value (positive or negative).

Returns:

Returns a nonzero value if and only if a is a infinite value.

5.62.2.35 __device__ int isnan (double *a*)

Determine whether the floating-point value a is a NaN.

Returns:

Returns a nonzero value if and only if a is a NaN value.

5.62.2.36 __device__ double j0 (double *x*)

Calculate the value of the Bessel function of the first kind of order 0 for the input argument x, $J_0(x)$.

Returns:

Returns the value of the Bessel function of the first kind of order 0.

- j0($\pm\infty$) returns +0.
- j0(NaN) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.37 __device__ double j1 (double *x*)

Calculate the value of the Bessel function of the first kind of order 1 for the input argument x, $J_1(x)$.

Returns:

Returns the value of the Bessel function of the first kind of order 1.

- j1(± 0) returns ± 0 .
- j1($\pm\infty$) returns +0.
- j1(NaN) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.38 __device__ double jn (int *n*, double *x*)

Calculate the value of the Bessel function of the first kind of order n for the input argument x, $J_n(x)$.

Returns:

Returns the value of the Bessel function of the first kind of order n.

- jn(n, NaN) returns NaN.
- jn(n, x) returns NaN for $n < 0$.
- jn(n, $+\infty$) returns +0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.39 __device__ double ldexp (double *x*, int *exp*)

Calculate the value of $x \cdot 2^{exp}$ of the input arguments x and exp.

Returns:

- ldexp(x) returns $\pm\infty$ if the correctly calculated value is outside the double floating point range.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.40 __device__ double lgamma (double *x*)

Calculate the natural logarithm of the absolute value of the gamma function of the input argument x, namely the value of $\log_e \left| \int_0^\infty e^{-t} t^{x-1} dt \right|$

Returns:

- lgamma(1) returns +0.
- lgamma(2) returns +0.
- lgamma(x) returns $\pm\infty$ if the correctly calculated value is outside the double floating point range.
- lgamma(x) returns $+\infty$ if x ≤ 0.
- lgamma($-\infty$) returns $-\infty$.
- lgamma($+\infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.41 __device__ long long int llrint (double *x*)

Round x to the nearest integer value, with halfway cases rounded towards zero. If the result is outside the range of the return type, the result is undefined.

Returns:

Returns rounded integer value.

5.62.2.42 __device__ long long int llround (double *x*)

Round x to the nearest integer value, with halfway cases rounded away from zero. If the result is outside the range of the return type, the result is undefined.

Returns:

Returns rounded integer value.

Note:

This function may be slower than alternate rounding methods. See llrint().

5.62.2.43 __device__ double log (double *x*)

Calculate the base e logarithm of the input argument x.

Returns:

- log(± 0) returns $-\infty$.

- log(1) returns +0.
- log(x) returns NaN for x < 0.
- log($+\infty$) returns $+\infty$

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.44 __device__ double log10 (double *x*)

Calculate the base 10 logarithm of the input argument x.

Returns:

- log10(± 0) returns $-\infty$.
- log10(1) returns +0.
- log10(x) returns NaN for x < 0.
- log10($+\infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.45 __device__ double log1p (double *x*)

Calculate the value of $log_e(1 + x)$ $\lfloor x \rfloor$ of the input argument x.

Returns:

- log1p(± 0) returns $-\infty$.
- log1p(-1) returns +0.
- log1p(x) returns NaN for x $<$ -1.
- log1p($+\infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.46 __device__ double log2 (double *x*)

Calculate the base 2 logarithm of the input argument x.

Returns:

- log2(± 0) returns $-\infty$.
- log2(1) returns +0.
- log2(x) returns NaN for x < 0.
- log2($+\infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.47 __device__ double logb (double *x*)

Calculate the floating point representation of the exponent of the input argument x.

Returns:

- logb ± 0 returns $-\infty$
- logb $\pm\infty$ returns $+\infty$

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.48 __device__ long int lrint (double *x*)

Round x to the nearest integer value, with halfway cases rounded towards zero. If the result is outside the range of the return type, the result is undefined.

Returns:

Returns rounded integer value.

5.62.2.49 __device__ long int lround (double *x*)

Round x to the nearest integer value, with halfway cases rounded away from zero. If the result is outside the range of the return type, the result is undefined.

Returns:

Returns rounded integer value.

Note:

This function may be slower than alternate rounding methods. See lrint().

5.62.2.50 __device__ double modf (double *x*, double * *iptr*)

Break down the argument x into fractional and integral parts. The integral part is stored in the argument iptr. Fractional and integral parts are given the same sign as the argument x.

Returns:

- modf($\pm x$, iptr) returns a result with the same sign as x.
- modf($\pm\infty$, iptr) returns ± 0 and stores $\pm\infty$ in the object pointed to by iptr.
- modf(NaN, iptr) stores a NaN in the object pointed to by iptr and returns a NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.51 __device__ double nan (const char * *tagp*)

Return a representation of a quiet NaN. Argument `tagp` selects one of the possible representations.

Returns:

- nan(`tagp`) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.52 __device__ double nearbyint (double *x*)

Round argument `x` to an integer value in double precision floating-point format.

Returns:

- nearbyint(± 0) returns ± 0 .
- nearbyint($\pm \infty$) returns $\pm \infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.53 __device__ double nextafter (double *x*, double *y*)

Calculate the next representable double-precision floating-point value following `x` in the direction of `y`. For example, if `y` is greater than `x`, nextafter() returns the smallest representable number greater than `x`

Returns:

- nextafter($\pm \infty$, `y`) returns $\pm \infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.54 __device__ double normcdf (double *y*)

Calculate the cumulative distribution function of the standard normal distribution for input argument `y`, $\Phi(y)$.

Returns:

- normcdf($+\infty$) returns 1
- normcdf($-\infty$) returns +0

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.55 __device__ double normcdfinv (double *y*)

Calculate the inverse of the standard normal cumulative distribution function for input argument y, $\Phi^{-1}(y)$. The function is defined for input values in the interval $(0, 1)$.

Returns:

- normcdfinv(0) returns $-\infty$.
- normcdfinv(1) returns $+\infty$.
- normcdfinv(x) returns NaN if x is not in the interval [0,1].

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.56 __device__ double pow (double *x*, double *y*)

Calculate the value of x to the power of y

Returns:

- pow(± 0 , y) returns $\pm\infty$ for y an integer less than 0.
- pow(± 0 , y) returns ± 0 for y an odd integer greater than 0.
- pow(± 0 , y) returns +0 for $y > 0$ and not and odd integer.
- pow(-1, $\pm\infty$) returns 1.
- pow(+1, y) returns 1 for any y, even a NaN.
- pow(x, ± 0) returns 1 for any x, even a NaN.
- pow(x, y) returns a NaN for finite $x < 0$ and finite non-integer y.
- pow(x, $-\infty$) returns $+\infty$ for $|x| < 1$.
- pow(x, $-\infty$) returns +0 for $|x| > 1$.
- pow(x, $+\infty$) returns +0 for $|x| < 1$.
- pow(x, $+\infty$) returns $+\infty$ for $|x| > 1$.
- pow($-\infty$, y) returns -0 for y an odd integer less than 0.
- pow($-\infty$, y) returns +0 for $y < 0$ and not an odd integer.
- pow($-\infty$, y) returns $-\infty$ for y an odd integer greater than 0.
- pow($-\infty$, y) returns $+\infty$ for $y > 0$ and not an odd integer.
- pow($+\infty$, y) returns +0 for $y < 0$.
- pow($+\infty$, y) returns $+\infty$ for $y > 0$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.57 __device__ double rcbrt (double *x*)

Calculate reciprocal cube root function of x

Returns:

- rcbrt(± 0) returns $\pm\infty$.
- rcbrt($\pm\infty$) returns ± 0 .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.58 __device__ double remainder (double *x*, double *y*)

Compute double-precision floating-point remainder r of dividing x by y for nonzero y. Thus $r = x - ny$. The value n is the integer value nearest $\frac{x}{y}$. In the case when $|n - \frac{x}{y}| = \frac{1}{2}$, the even n value is chosen.

Returns:

- remainder(x, 0) returns NaN.
- remainder($\pm\infty$, y) returns NaN.
- remainder(x, $\pm\infty$) returns x for finite x.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.59 __device__ double remquo (double *x*, double *y*, int * *quo*)

Compute a double-precision floating-point remainder in the same way as the remainder() function. Argument quo returns part of quotient upon division of x by y. Value quo has the same sign as $\frac{x}{y}$ and may not be the exact quotient but agrees with the exact quotient in the low order 3 bits.

Returns:

Returns the remainder.

- remquo(x, 0, quo) returns NaN.
- remquo($\pm\infty$, y, quo) returns NaN.
- remquo(x, $\pm\infty$, quo) returns x.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.60 __device__ double rint (double *x*)

Round x to the nearest integer value in floating-point format, with halfway cases rounded towards zero.

Returns:

Returns rounded integer value.

5.62.2.61 __device__ double round (double *x*)

Round x to the nearest integer value in floating-point format, with halfway cases rounded away from zero.

Returns:

Returns rounded integer value.

Note:

This function may be slower than alternate rounding methods. See rint().

5.62.2.62 __device__ double rsqrt (double *x*)

Calculate the reciprocal of the nonnegative square root of x, $1/\sqrt{x}$.

Returns:

Returns $1/\sqrt{x}$.

- rsqrt($+\infty$) returns +0.
- rsqrt(± 0) returns $\pm\infty$.
- rsqrt(x) returns NaN if x is less than 0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.63 __device__ double scalbln (double *x*, long int *n*)

Scale x by 2^n by efficient manipulation of the floating-point exponent.

Returns:

Returns $x * 2^n$.

- scalbln(± 0 , n) returns ± 0 .
- scalbln(x, 0) returns x.
- scalbln($\pm\infty$, n) returns $\pm\infty$.

5.62.2.64 __device__ double scalbn (double *x*, int *n*)

Scale x by 2^n by efficient manipulation of the floating-point exponent.

Returns:

Returns $x * 2^n$.

- scalbn(± 0 , n) returns ± 0 .
- scalbn(x, 0) returns x.
- scalbn($\pm\infty$, n) returns $\pm\infty$.

5.62.2.65 __device__ int signbit (double *a*)

Determine whether the floating-point value `a` is negative.

Returns:

Returns a nonzero value if and only if `a` is negative. Reports the sign bit of all values including infinities, zeros, and NaNs.

5.62.2.66 __device__ double sin (double *x*)

Calculate the sine of the input argument `x` (measured in radians).

Returns:

- sin(± 0) returns ± 0 .
- sin($\pm \infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.67 __device__ void sincos (double *x*, double * *sptr*, double * *cptr*)

Calculate the sine and cosine of the first input argument `x` (measured in radians). The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.

Returns:

- none

See also:

sin() and cos().

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.68 __device__ void sincospi (double *x*, double * *sptr*, double * *cptr*)

Calculate the sine and cosine of the first input argument, `x` (measured in radians), $\times \pi$. The results for sine and cosine are written into the second argument, `sptr`, and, respectively, third argument, `cptr`.

Returns:

- none

See also:

sinpi() and cospi().

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.69 __device__ double sinh (double *x*)

Calculate the hyperbolic sine of the input argument x (measured in radians).

Returns:

- sinh(± 0) returns ± 0 .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.70 __device__ double sinpi (double *x*)

Calculate the sine of x $\times \pi$ (measured in radians), where x is the input argument.

Returns:

- sinpi(± 0) returns ± 0 .
- sinpi($\pm \infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.71 __device__ double sqrt (double *x*)

Calculate the nonnegative square root of x, $\sqrt{x}$.

Returns:

Returns $\sqrt{x}$.

- sqrt(± 0) returns ± 0 .
- sqrt($+\infty$) returns $+\infty$.
- sqrt(x) returns NaN if x is less than 0.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.72 __device__ double tan (double *x*)

Calculate the tangent of the input argument x (measured in radians).

Returns:

- tan(± 0) returns ± 0 .
- tan($\pm \infty$) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.73 __device__ double tanh (double *x*)

Calculate the hyperbolic tangent of the input argument x (measured in radians).

Returns:

- tanh(± 0) returns ± 0 .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.74 __device__ double tgamma (double *x*)

Calculate the gamma function of the input argument x, namely the value of $\int_0^\infty e^{-t} t^{x-1} dt$.

Returns:

- tgamma(± 0) returns $\pm \infty$.
- tgamma(2) returns +0.
- tgamma(x) returns $\pm \infty$ if the correctly calculated value is outside the double floating point range.
- tgamma(x) returns NaN if $x < 0$.
- tgamma($-\infty$) returns NaN.
- tgamma($+\infty$) returns $+\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.75 __device__ double trunc (double *x*)

Round x to the nearest integer value that does not exceed x in magnitude.

Returns:

Returns truncated integer value.

5.62.2.76 __device__ double y0 (double *x*)

Calculate the value of the Bessel function of the second kind of order 0 for the input argument x, $Y_0(x)$.

Returns:

Returns the value of the Bessel function of the second kind of order 0.

- y0(0) returns $-\infty$.
- y0(x) returns NaN for $x < 0$.
- y0($+\infty$) returns +0.
- y0(NaN) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.77 __device__ double y1 (double *x*)

Calculate the value of the Bessel function of the second kind of order 1 for the input argument x, $Y_1(x)$.

Returns:

Returns the value of the Bessel function of the second kind of order 1.

- y1(0) returns $-\infty$.
- y1(x) returns NaN for x < 0.
- y1($+\infty$) returns +0.
- y1(NaN) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.62.2.78 __device__ double yn (int *n*, double *x*)

Calculate the value of the Bessel function of the second kind of order n for the input argument x, $Y_n(x)$.

Returns:

Returns the value of the Bessel function of the second kind of order n.

- yn(n, x) returns NaN for n < 0.
- yn(n, 0) returns $-\infty$.
- yn(n, x) returns NaN for x < 0.
- yn(n, $+\infty$) returns +0.
- yn(n, NaN) returns NaN.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.63 Single Precision Intrinsics

Functions

- __device__ __cudart_builtin__ float __cosf (float x)
  Calculate the fast approximate cosine of the input argument.
- __device__ __cudart_builtin__ float __exp10f (float x)
  Calculate the fast approximate base 10 exponential of the input argument.
- __device__ __cudart_builtin__ float __expf (float x)
  Calculate the fast approximate base e exponential of the input argument.
- __device__ float __fadd_rd (float x, float y)
  Add two floating point values in round-down mode.
- __device__ float __fadd_rn (float x, float y)
  Add two floating point values in round-to-nearest-even mode.
- __device__ float __fadd_ru (float x, float y)
  Add two floating point values in round-up mode.
- __device__ float __fadd_rz (float x, float y)
  Add two floating point values in round-towards-zero mode.
- __device__ float __fdiv_rd (float x, float y)
  Divide two floating point values in round-down mode.
- __device__ float __fdiv_rn (float x, float y)
  Divide two floating point values in round-to-nearest-even mode.
- __device__ float __fdiv_ru (float x, float y)
  Divide two floating point values in round-up mode.
- __device__ float __fdiv_rz (float x, float y)
  Divide two floating point values in round-towards-zero mode.
- __device__ float __fdividef (float x, float y)
  Calculate the fast approximate division of the input arguments.
- __device__ float __fmaf_rd (float x, float y, float z)
  Compute $x \times y + z$ as a single operation, in round-down mode.
- __device__ float __fmaf_rn (float x, float y, float z)
  Compute $x \times y + z$ as a single operation, in round-to-nearest-even mode.
- __device__ float __fmaf_ru (float x, float y, float z)
  Compute $x \times y + z$ as a single operation, in round-up mode.
- __device__ float __fmaf_rz (float x, float y, float z)

Compute $x \times y + z$ as a single operation, in round-towards-zero mode.

- __device__ float __fmul_rd (float x, float y)

  Multiply two floating point values in round-down mode.

- __device__ float __fmul_rn (float x, float y)

  Multiply two floating point values in round-to-nearest-even mode.

- __device__ float __fmul_ru (float x, float y)

  Multiply two floating point values in round-up mode.

- __device__ float __fmul_rz (float x, float y)

  Multiply two floating point values in round-towards-zero mode.

- __device__ float __frcp_rd (float x)

  Compute $\frac{1}{x}$ in round-down mode.

- __device__ float __frcp_rn (float x)

  Compute $\frac{1}{x}$ in round-to-nearest-even mode.

- __device__ float __frcp_ru (float x)

  Compute $\frac{1}{x}$ in round-up mode.

- __device__ float __frcp_rz (float x)

  Compute $\frac{1}{x}$ in round-towards-zero mode.

- __device__ float __frsqrt_rn (float x)

  Compute $1/\sqrt{x}$ in round-to-nearest-even mode.

- __device__ float __fsqrt_rd (float x)

  Compute $\sqrt{x}$ in round-down mode.

- __device__ float __fsqrt_rn (float x)

  Compute $\sqrt{x}$ in round-to-nearest-even mode.

- __device__ float __fsqrt_ru (float x)

  Compute $\sqrt{x}$ in round-up mode.

- __device__ float __fsqrt_rz (float x)

  Compute $\sqrt{x}$ in round-towards-zero mode.

- __device__ __cudart_builtin__ float __log10f (float x)

  Calculate the fast approximate base 10 logarithm of the input argument.

- __device__ __cudart_builtin__ float __log2f (float x)

  Calculate the fast approximate base 2 logarithm of the input argument.

- __device__ __cudart_builtin__ float __logf (float x)

  Calculate the fast approximate base e logarithm of the input argument.

- __device__ __cudart_builtin__ float __powf (float x, float y)

  Calculate the fast approximate of x^y .

- __device__ float __saturatef (float x)

  Clamp the input argument to [+0.0, 1.0].

- __device__ __cudart_builtin__ void __sincosf (float x, float *sptr, float *cptr)

  Calculate the fast approximate of sine and cosine of the first input argument.

- __device__ __cudart_builtin__ float __sinf (float x)

  Calculate the fast approximate sine of the input argument.

- __device__ __cudart_builtin__ float __tanf (float x)

  Calculate the fast approximate tangent of the input argument.

5.63.1 Detailed Description

This section describes single precision intrinsic functions that are only supported in device code.

5.63.2 Function Documentation

5.63.2.1 __device__ __cudart_builtin__ float __cosf (float *x*)

Calculate the fast approximate cosine of the input argument x, measured in radians.

Returns:

Returns the approximate cosine of x.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-4. Input and output in the denormal range is flushed to sign preserving 0.0.

5.63.2.2 __device__ __cudart_builtin__ float __exp10f (float *x*)

Calculate the fast approximate base 10 exponential of the input argument x, 10^x .

Returns:

Returns an approximation to 10^x .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-4. Most input and output values around denormal range are flushed to sign preserving 0.0.

5.63.2.3 __device__ __cudart_builtin__ float __expf (float *x*)

Calculate the fast approximate base e exponential of the input argument x, e^x .

Returns:

Returns an approximation to e^x .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-4. Most input and output values around denormal range are flushed to sign preserving 0.0.

5.63.2.4 __device__ float __fadd_rd (float *x*, float *y*)

Compute the sum of x and y in round-down (to negative infinity) mode.

Returns:

Returns x + y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1. This operation will never be merged into a single multiply-add instruction.

5.63.2.5 __device__ float __fadd_rn (float *x*, float *y*)

Compute the sum of x and y in round-to-nearest-even rounding mode.

Returns:

Returns x + y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1. This operation will never be merged into a single multiply-add instruction.

5.63.2.6 __device__ float __fadd_ru (float *x*, float *y*)

Compute the sum of x and y in round-up (to positive infinity) mode.

Returns:

Returns x + y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1. This operation will never be merged into a single multiply-add instruction.

5.63.2.7 __device__ float __fadd_rz (float *x*, float *y*)

Compute the sum of x and y in round-towards-zero mode.

Returns:

Returns x + y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1. This operation will never be merged into a single multiply-add instruction.

5.63.2.8 __device__ float __fdiv_rd (float *x*, float *y*)

Divide two floating point values x by y in round-down (to negative infinity) mode.

Returns:

Returns x / y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.9 __device__ float __fdiv_rn (float *x*, float *y*)

Divide two floating point values x by y in round-to-nearest-even mode.

Returns:

Returns x / y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.10 __device__ float __fdiv_ru (float *x*, float *y*)

Divide two floating point values x by y in round-up (to positive infinity) mode.

Returns:

Returns x / y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.11 __device__ float __fdiv_rz (float *x*, float *y*)

Divide two floating point values x by y in round-towards-zero mode.

Returns:

Returns x / y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.12 __device__ float __fdividef (float *x*, float *y*)

Calculate the fast approximate division of x by y.

Returns:

Returns x / y.

- __fdividef(∞ , y) returns NaN for $2^{126} < y < 2^{128}$.
- __fdividef(x, y) returns 0 for $2^{126} < y < 2^{128}$ and $x \neq \infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-4.

5.63.2.13 __device__ float __fmaf_rd (float *x*, float *y*, float *z*)

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-down (to negative infinity) mode.

Returns:

Returns the rounded value of $x \times y + z$ as a single operation.

- fmaf($\pm\infty$, ± 0 , z) returns NaN.
- fmaf(± 0 , $\pm\infty$, z) returns NaN.
- fmaf(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- fmaf(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.14 __device__ float __fmaf_rn (float *x*, float *y*, float *z*)

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-to-nearest-even mode.

Returns:

Returns the rounded value of $x \times y + z$ as a single operation.

- fmaf($\pm\infty$, ± 0 , z) returns NaN.
- fmaf(± 0 , $\pm\infty$, z) returns NaN.
- fmaf(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- fmaf(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.15 __device__ float __fmaf_ru (float *x*, float *y*, float *z*)

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-up (to positive infinity) mode.

Returns:

Returns the rounded value of $x \times y + z$ as a single operation.

- fmaf($\pm\infty$, ± 0 , z) returns NaN.
- fmaf(± 0 , $\pm\infty$, z) returns NaN.
- fmaf(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- fmaf(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.16 __device__ float __fmaf_rz (float *x*, float *y*, float *z*)

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-towards-zero mode.

Returns:

Returns the rounded value of $x \times y + z$ as a single operation.

- fmaf($\pm\infty$, ± 0 , z) returns NaN.
- fmaf(± 0 , $\pm\infty$, z) returns NaN.
- fmaf(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$.
- fmaf(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.17 __device__ float __fmul_rd (float *x*, float *y*)

Compute the product of x and y in round-down (to negative infinity) mode.

Returns:

Returns x $*$ y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.
This operation will never be merged into a single multiply-add instruction.

5.63.2.18 __device__ float __fmul_rn (float *x*, float *y*)

Compute the product of x and y in round-to-nearest-even mode.

Returns:

Returns x * y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.
This operation will never be merged into a single multiply-add instruction.

5.63.2.19 __device__ float __fmul_ru (float *x*, float *y*)

Compute the product of x and y in round-up (to positive infinity) mode.

Returns:

Returns x * y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.
This operation will never be merged into a single multiply-add instruction.

5.63.2.20 __device__ float __fmul_rz (float *x*, float *y*)

Compute the product of x and y in round-towards-zero mode.

Returns:

Returns x * y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.
This operation will never be merged into a single multiply-add instruction.

5.63.2.21 __device__ float __frcp_rd (float *x*)

Compute the reciprocal of x in round-down (to negative infinity) mode.

Returns:

Returns $\frac{1}{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.22 __device__ float __frcp_rn (float *x*)

Compute the reciprocal of x in round-to-nearest-even mode.

Returns:

Returns $\frac{1}{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.23 __device__ float __frcp_ru (float *x*)

Compute the reciprocal of x in round-up (to positive infinity) mode.

Returns:

Returns $\frac{1}{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.24 __device__ float __frcp_rz (float *x*)

Compute the reciprocal of x in round-towards-zero mode.

Returns:

Returns $\frac{1}{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.25 __device__ float __frsqrt_rn (float *x*)

Compute the reciprocal square root of x in round-to-nearest-even mode.

Returns:

Returns $1/\sqrt{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.26 __device__ float __fsqrt_rd (float *x*)

Compute the square root of x in round-down (to negative infinity) mode.

Returns:

Returns $\sqrt{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.27 __device__ float __fsqrt_rn (float *x*)

Compute the square root of x in round-to-nearest-even mode.

Returns:

Returns $\sqrt{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.28 __device__ float __fsqrt_ru (float *x*)

Compute the square root of x in round-up (to positive infinity) mode.

Returns:

Returns $\sqrt{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.29 __device__ float __fsqrt_rz (float *x*)

Compute the square root of x in round-towards-zero mode.

Returns:

Returns $\sqrt{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-1.

5.63.2.30 __device__ __cudart_builtin__ float __log10f (float *x*)

Calculate the fast approximate base 10 logarithm of the input argument x.

Returns:

Returns an approximation to $\log_{10}(x)$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-4. Most input and output values around denormal range are flushed to sign preserving 0.0.

5.63.2.31 __device__ __cudart_builtin__ float __log2f (float *x*)

Calculate the fast approximate base 2 logarithm of the input argument x.

Returns:

Returns an approximation to $\log_2(x)$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-4. Input and output in the denormal range is flushed to sign preserving 0.0.

5.63.2.32 __device__ __cudart_builtin__ float __logf (float *x*)

Calculate the fast approximate base e logarithm of the input argument x.

Returns:

Returns an approximation to $\log_e(x)$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-4. Most input and output values around denormal range are flushed to sign preserving 0.0.

5.63.2.33 __device__ __cudart_builtin__ float __powf (float *x*, float *y*)

Calculate the fast approximate of x, the first input argument, raised to the power of y, the second input argument, x^y .

Returns:

Returns an approximation to x^y .

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-4. Most input and output values around denormal range are flushed to sign preserving 0.0.

5.63.2.34 __device__ float __saturatef (float *x*)

Clamp the input argument x to be within the interval [+0.0, 1.0].

Returns:

- __saturatef(x) returns 0 if $x < 0$.
- __saturatef(x) returns 1 if $x > 1$.
- __saturatef(x) returns x if $0 \leq x \leq 1$.
- __saturatef(NaN) returns 0.

5.63.2.35 __device__ __cudart_builtin__ void __sincosf (float *x*, float * *sptr*, float * *cptr*)

Calculate the fast approximate of sine and cosine of the first input argument x (measured in radians). The results for sine and cosine are written into the second argument, sptr, and, respectively, third argument, cptr.

Returns:

- none

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-4.
Denorm input/output is flushed to sign preserving 0.0.

5.63.2.36 __device__ __cudart_builtin__ float __sinf (float *x*)

Calculate the fast approximate sine of the input argument x, measured in radians.

Returns:

Returns the approximate sine of x.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-4.
Input and output in the denormal range is flushed to sign preserving 0.0.

5.63.2.37 __device__ __cudart_builtin__ float __tanf (float *x*)

Calculate the fast approximate tangent of the input argument x, measured in radians.

Returns:

Returns the approximate tangent of x.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-4.
The result is computed as the fast divide of __sinf() by __cosf(). Denormal input and output are flushed to sign-preserving 0.0 at each step of the computation.

5.64 Double Precision Intrinsics

Functions

- __device__ double __dadd_rd (double x, double y)
  Add two floating point values in round-down mode.
- __device__ double __dadd_rn (double x, double y)
  Add two floating point values in round-to-nearest-even mode.
- __device__ double __dadd_ru (double x, double y)
  Add two floating point values in round-up mode.
- __device__ double __dadd_rz (double x, double y)
  Add two floating point values in round-towards-zero mode.
- __device__ double __ddiv_rd (double x, double y)
  Divide two floating point values in round-down mode.
- __device__ double __ddiv_rn (double x, double y)
  Divide two floating point values in round-to-nearest-even mode.
- __device__ double __ddiv_ru (double x, double y)
  Divide two floating point values in round-up mode.
- __device__ double __ddiv_rz (double x, double y)
  Divide two floating point values in round-towards-zero mode.
- __device__ double __dmul_rd (double x, double y)
  Multiply two floating point values in round-down mode.
- __device__ double __dmul_rn (double x, double y)
  Multiply two floating point values in round-to-nearest-even mode.
- __device__ double __dmul_ru (double x, double y)
  Multiply two floating point values in round-up mode.
- __device__ double __dmul_rz (double x, double y)
  Multiply two floating point values in round-towards-zero mode.
- __device__ double __drcp_rd (double x)
  Compute $\frac{1}{x}$ *in round-down mode.*
- __device__ double __drcp_rn (double x)
  Compute $\frac{1}{x}$ *in round-to-nearest-even mode.*
- __device__ double __drcp_ru (double x)
  Compute $\frac{1}{x}$ *in round-up mode.*
- __device__ double __drcp_rz (double x)

Compute $\frac{1}{x}$ in round-towards-zero mode.

- __device__ double __dsqrt_rd (double x)

  Compute $\sqrt{x}$ in round-down mode.

- __device__ double __dsqrt_rn (double x)

  Compute $\sqrt{x}$ in round-to-nearest-even mode.

- __device__ double __dsqrt_ru (double x)

  Compute $\sqrt{x}$ in round-up mode.

- __device__ double __dsqrt_rz (double x)

  Compute $\sqrt{x}$ in round-towards-zero mode.

- __device__ double __fma_rd (double x, double y, double z)

  Compute $x \times y + z$ as a single operation in round-down mode.

- __device__ double __fma_rn (double x, double y, double z)

  Compute $x \times y + z$ as a single operation in round-to-nearest-even mode.

- __device__ double __fma_ru (double x, double y, double z)

  Compute $x \times y + z$ as a single operation in round-up mode.

- __device__ double __fma_rz (double x, double y, double z)

  Compute $x \times y + z$ as a single operation in round-towards-zero mode.

5.64.1 Detailed Description

This section describes double precision intrinsic functions that are only supported in device code.

5.64.2 Function Documentation

5.64.2.1 __device__ double __dadd_rd (double *x*, double *y*)

Adds two floating point values `x` and `y` in round-down (to negative infinity) mode.

Returns:

Returns `x` + `y`.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.
This operation will never be merged into a single multiply-add instruction.

5.64.2.2 __device__ double __dadd_rn (double *x*, double *y*)

Adds two floating point values x and y in round-to-nearest-even mode.

Returns:

Returns x + y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2. This operation will never be merged into a single multiply-add instruction.

5.64.2.3 __device__ double __dadd_ru (double *x*, double *y*)

Adds two floating point values x and y in round-up (to positive infinity) mode.

Returns:

Returns x + y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2. This operation will never be merged into a single multiply-add instruction.

5.64.2.4 __device__ double __dadd_rz (double *x*, double *y*)

Adds two floating point values x and y in round-towards-zero mode.

Returns:

Returns x + y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2. This operation will never be merged into a single multiply-add instruction.

5.64.2.5 __device__ double __ddiv_rd (double *x*, double *y*)

Divides two floating point values x by y in round-down (to negative infinity) mode.

Returns:

Returns x / y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2. Requires compute capability >= 2.0.

5.64.2.6 __device__ double __ddiv_rn (double *x*, double *y*)

Divides two floating point values x by y in round-to-nearest-even mode.

Returns:

Returns x / y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.
Requires compute capability >= 2.0.

5.64.2.7 __device__ double __ddiv_ru (double *x*, double *y*)

Divides two floating point values x by y in round-up (to positive infinity) mode.

Returns:

Returns x / y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.
Requires compute capability >= 2.0.

5.64.2.8 __device__ double __ddiv_rz (double *x*, double *y*)

Divides two floating point values x by y in round-towards-zero mode.

Returns:

Returns x / y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.
Requires compute capability >= 2.0.

5.64.2.9 __device__ double __dmul_rd (double *x*, double *y*)

Multiplies two floating point values x and y in round-down (to negative infinity) mode.

Returns:

Returns x * y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.
This operation will never be merged into a single multiply-add instruction.

5.64.2.10 __device__ double __dmul_rn (double *x*, double *y*)

Multiplies two floating point values x and y in round-to-nearest-even mode.

Returns:

Returns x ∗ y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2. This operation will never be merged into a single multiply-add instruction.

5.64.2.11 __device__ double __dmul_ru (double *x*, double *y*)

Multiplies two floating point values x and y in round-up (to positive infinity) mode.

Returns:

Returns x ∗ y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2. This operation will never be merged into a single multiply-add instruction.

5.64.2.12 __device__ double __dmul_rz (double *x*, double *y*)

Multiplies two floating point values x and y in round-towards-zero mode.

Returns:

Returns x ∗ y.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2. This operation will never be merged into a single multiply-add instruction.

5.64.2.13 __device__ double __drcp_rd (double *x*)

Compute the reciprocal of x in round-down (to negative infinity) mode.

Returns:

Returns $\frac{1}{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2. Requires compute capability >= 2.0.

5.64.2.14 __device__ double __drcp_rn (double *x*)

Compute the reciprocal of x in round-to-nearest-even mode.

Returns:

Returns $\frac{1}{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.
Requires compute capability >= 2.0.

5.64.2.15 __device__ double __drcp_ru (double *x*)

Compute the reciprocal of x in round-up (to positive infinity) mode.

Returns:

Returns $\frac{1}{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.
Requires compute capability >= 2.0.

5.64.2.16 __device__ double __drcp_rz (double *x*)

Compute the reciprocal of x in round-towards-zero mode.

Returns:

Returns $\frac{1}{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.
Requires compute capability >= 2.0.

5.64.2.17 __device__ double __dsqrt_rd (double *x*)

Compute the square root of x in round-down (to negative infinity) mode.

Returns:

Returns $\sqrt{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.
Requires compute capability >= 2.0.

5.64.2.18 __device__ double __dsqrt_rn (double *x*)

Compute the square root of x in round-to-nearest-even mode.

Returns:

Returns $\sqrt{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2. Requires compute capability >= 2.0.

5.64.2.19 __device__ double __dsqrt_ru (double *x*)

Compute the square root of x in round-up (to positive infinity) mode.

Returns:

Returns $\sqrt{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2. Requires compute capability >= 2.0.

5.64.2.20 __device__ double __dsqrt_rz (double *x*)

Compute the square root of x in round-towards-zero mode.

Returns:

Returns $\sqrt{x}$.

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2. Requires compute capability >= 2.0.

5.64.2.21 __device__ double __fma_rd (double *x*, double *y*, double *z*)

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-down (to negative infinity) mode.

Returns:

Returns the rounded value of $x \times y + z$ as a single operation.

- fmaf($\pm\infty$, ± 0 , z) returns NaN.
- fmaf(± 0 , $\pm\infty$, z) returns NaN.
- fmaf(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$
- fmaf(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.64.2.22 __device__ double __fma_rn (double *x*, double *y*, double *z*)

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-to-nearest-even mode.

Returns:

Returns the rounded value of $x \times y + z$ as a single operation.

- fmaf($\pm\infty$, ± 0 , z) returns NaN.
- fmaf(± 0 , $\pm\infty$, z) returns NaN.
- fmaf(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$
- fmaf(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.64.2.23 __device__ double __fma_ru (double *x*, double *y*, double *z*)

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-up (to postive infinity) mode.

Returns:

Returns the rounded value of $x \times y + z$ as a single operation.

- fmaf($\pm\infty$, ± 0 , z) returns NaN.
- fmaf(± 0 , $\pm\infty$, z) returns NaN.
- fmaf(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$
- fmaf(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.64.2.24 __device__ double __fma_rz (double *x*, double *y*, double *z*)

Computes the value of $x \times y + z$ as a single ternary operation, rounding the result once in round-towards-zero mode.

Returns:

Returns the rounded value of $x \times y + z$ as a single operation.

- fmaf($\pm\infty$, ± 0 , z) returns NaN.
- fmaf(± 0 , $\pm\infty$, z) returns NaN.
- fmaf(x, y, $-\infty$) returns NaN if $x \times y$ is an exact $+\infty$
- fmaf(x, y, $+\infty$) returns NaN if $x \times y$ is an exact $-\infty$

Note:

For accuracy information for this function see the CUDA C Programming Guide, Appendix C, Table C-2.

5.65 Integer Intrinsics

Functions

- __device__ unsigned int __brev (unsigned int x)
  Reverse the bit order of a 32 bit unsigned integer.
- __device__ unsigned long long int __brevll (unsigned long long int x)
  Reverse the bit order of a 64 bit unsigned integer.
- __device__ unsigned int __byte_perm (unsigned int x, unsigned int y, unsigned int s)
  Return selected bytes from two 32 bit unsigned integers.
- __device__ int __clz (int x)
  Return the number of consecutive high-order zero bits in a 32 bit integer.
- __device__ int __clzll (long long int x)
  Count the number of consecutive high-order zero bits in a 64 bit integer.
- __device__ int __ffs (int x)
  Find the position of the least significant bit set to 1 in a 32 bit integer.
- __device__ int __ffsll (long long int x)
  Find the position of the least significant bit set to 1 in a 64 bit integer.
- __device__ int __mul24 (int x, int y)
  Calculate the least significant 32 bits of the product of the least significant 24 bits of two integers.
- __device__ long long int __mul64hi (long long int x, long long int y)
  Calculate the most significant 64 bits of the product of the two 64 bit integers.
- __device__ int __mulhi (int x, int y)
  Calculate the most significant 32 bits of the product of the two 32 bit integers.
- __device__ int __popc (unsigned int x)
  Count the number of bits that are set to 1 in a 32 bit integer.
- __device__ int __popcll (unsigned long long int x)
  Count the number of bits that are set to 1 in a 64 bit integer.
- __device__ unsigned int __sad (int x, int y, unsigned int z)
  Calculate $|x - y| + z$, the sum of absolute difference.
- __device__ unsigned int __umul24 (unsigned int x, unsigned int y)
  Calculate the least significant 32 bits of the product of the least significant 24 bits of two unsigned integers.
- __device__ unsigned long long int __umul64hi (unsigned long long int x, unsigned long long int y)
  Calculate the most significant 64 bits of the product of the two 64 unsigned bit integers.
- __device__ unsigned int __umulhi (unsigned int x, unsigned int y)

Calculate the most significant 32 bits of the product of the two 32 bit unsigned integers.

- __device__ unsigned int __usad (unsigned int x, unsigned int y, unsigned int z)
  Calculate $|x - y| + z$, the sum of absolute difference.

5.65.1 Detailed Description

This section describes integer intrinsic functions that are only supported in device code.

5.65.2 Function Documentation

5.65.2.1 __device__ unsigned int __brev (unsigned int *x*)

Reverses the bit order of the 32 bit unsigned integer `x`.

Returns:

Returns the bit-reversed value of `x`. i.e. bit N of the return value corresponds to bit 31-N of `x`.

5.65.2.2 __device__ unsigned long long int __brevll (unsigned long long int *x*)

Reverses the bit order of the 64 bit unsigned integer `x`.

Returns:

Returns the bit-reversed value of `x`. i.e. bit N of the return value corresponds to bit 63-N of `x`.

5.65.2.3 __device__ unsigned int __byte_perm (unsigned int *x*, unsigned int *y*, unsigned int *s*)

byte_perm(x,y,s) returns a 32-bit integer consisting of four bytes from eight input bytes provided in the two input integers `x` and `y`, as specified by a selector, `s`.

The input bytes are indexed as follows:

```
input[0] = x<0:7>   input[1] = x<8:15>
input[2] = x<16:23> input[3] = x<24:31>
input[4] = y<0:7>   input[5] = y<8:15>
input[6] = y<16:23> input[7] = y<24:31>
```

The selector indices are stored in 4-bit nibbles (with the upper 16-bits of the selector not being used):

```
selector[0] = s<0:3>  selector[1] = s<4:7>
selector[2] = s<8:11> selector[3] = s<12:15>
```

Returns:

The returned value r is computed to be: `result[n] := input[selector[n]]` where `result[n]` is the nth byte of r.

5.65.2.4 __device__ int __clz (int *x*)

Count the number of consecutive leading zero bits, starting at the most significant bit (bit 31) of x.

Returns:

Returns a value between 0 and 32 inclusive representing the number of zero bits.

5.65.2.5 __device__ int __clzll (long long int *x*)

Count the number of consecutive leading zero bits, starting at the most significant bit (bit 63) of x.

Returns:

Returns a value between 0 and 64 inclusive representing the number of zero bits.

5.65.2.6 __device__ int __ffs (int *x*)

Find the position of the first (least significant) bit set to 1 in x, where the least significant bit position is 1.

Returns:

Returns a value between 0 and 32 inclusive representing the position of the first bit set.

- __ffs(0) returns 0.

5.65.2.7 __device__ int __ffsll (long long int *x*)

Find the position of the first (least significant) bit set to 1 in x, where the least significant bit position is 1.

Returns:

Returns a value between 0 and 64 inclusive representing the position of the first bit set.

- __ffsll(0) returns 0.

5.65.2.8 __device__ int __mul24 (int *x*, int *y*)

Calculate the least significant 32 bits of the product of the least significant 24 bits of x and y. The high order 8 bits of x and y are ignored.

Returns:

Returns the least significant 32 bits of the product x * y.

5.65.2.9 __device__ long long int __mul64hi (long long int *x*, long long int *y*)

Calculate the most significant 64 bits of the 128-bit product x * y, where x and y are 64-bit integers.

Returns:

Returns the most significant 64 bits of the product x * y.

5.65.2.10 __device__ int __mulhi (int *x*, int *y*)

Calculate the most significant 32 bits of the 64-bit product x * y, where x and y are 32-bit integers.

Returns:

Returns the most significant 32 bits of the product x * y.

5.65.2.11 __device__ int __popc (unsigned int *x*)

Count the number of bits that are set to 1 in x.

Returns:

Returns a value between 0 and 32 inclusive representing the number of set bits.

5.65.2.12 __device__ int __popcll (unsigned long long int *x*)

Count the number of bits that are set to 1 in x.

Returns:

Returns a value between 0 and 64 inclusive representing the number of set bits.

5.65.2.13 __device__ unsigned int __sad (int *x*, int *y*, unsigned int *z*)

Calculate $|x - y| + z$, the 32-bit sum of the third argument z plus and the absolute value of the difference between the first argument, x, and second argument, y.

Inputs x and y are signed 32-bit integers, input z is a 32-bit unsigned integer.

Returns:

Returns $|x - y| + z$.

5.65.2.14 __device__ unsigned int __umul24 (unsigned int *x*, unsigned int *y*)

Calculate the least significant 32 bits of the product of the least significant 24 bits of x and y. The high order 8 bits of x and y are ignored.

Returns:

Returns the least significant 32 bits of the product x * y.

5.65.2.15 __device__ unsigned long long int __umul64hi (unsigned long long int *x*, unsigned long long int *y*)

Calculate the most significant 64 bits of the 128-bit product x * y, where x and y are 64-bit unsigned integers.

Returns:

Returns the most significant 64 bits of the product x * y.

5.65.2.16 __device__ unsigned int __umulhi (unsigned int *x*, unsigned int *y*)

Calculate the most significant 32 bits of the 64-bit product x * y, where x and y are 32-bit unsigned integers.

Returns:

Returns the most significant 32 bits of the product x * y.

5.65.2.17 __device__ unsigned int __usad (unsigned int *x*, unsigned int *y*, unsigned int *z*)

Calculate $|x - y| + z$, the 32-bit sum of the third argument z plus and the absolute value of the difference between the first argument, x, and second argument, y.

Inputs x, y, and z are unsigned 32-bit integers.

Returns:

Returns $|x - y| + z$.

5.66 Type Casting Intrinsics

Functions

- __device__ float __double2float_rd (double x)
  Convert a double to a float in round-down mode.
- __device__ float __double2float_rn (double x)
  Convert a double to a float in round-to-nearest-even mode.
- __device__ float __double2float_ru (double x)
  Convert a double to a float in round-up mode.
- __device__ float __double2float_rz (double x)
  Convert a double to a float in round-towards-zero mode.
- __device__ int __double2hiint (double x)
  Reinterpret high 32 bits in a double as a signed integer.
- __device__ int __double2int_rd (double x)
  Convert a double to a signed int in round-down mode.
- __device__ int __double2int_rn (double x)
  Convert a double to a signed int in round-to-nearest-even mode.
- __device__ int __double2int_ru (double x)
  Convert a double to a signed int in round-up mode.
- __device__ int __double2int_rz (double)
  Convert a double to a signed int in round-towards-zero mode.
- __device__ long long int __double2ll_rd (double x)
  Convert a double to a signed 64-bit int in round-down mode.
- __device__ long long int __double2ll_rn (double x)
  Convert a double to a signed 64-bit int in round-to-nearest-even mode.
- __device__ long long int __double2ll_ru (double x)
  Convert a double to a signed 64-bit int in round-up mode.
- __device__ long long int __double2ll_rz (double)
  Convert a double to a signed 64-bit int in round-towards-zero mode.
- __device__ int __double2loint (double x)
  Reinterpret low 32 bits in a double as a signed integer.
- __device__ unsigned int __double2uint_rd (double x)
  Convert a double to an unsigned int in round-down mode.
- __device__ unsigned int __double2uint_rn (double x)

Convert a double to an unsigned int in round-to-nearest-even mode.

- __device__ unsigned int __double2uint_ru (double x)

  Convert a double to an unsigned int in round-up mode.

- __device__ unsigned int __double2uint_rz (double)

  Convert a double to an unsigned int in round-towards-zero mode.

- __device__ unsigned long long int __double2ull_rd (double x)

  Convert a double to an unsigned 64-bit int in round-down mode.

- __device__ unsigned long long int __double2ull_rn (double x)

  Convert a double to an unsigned 64-bit int in round-to-nearest-even mode.

- __device__ unsigned long long int __double2ull_ru (double x)

  Convert a double to an unsigned 64-bit int in round-up mode.

- __device__ unsigned long long int __double2ull_rz (double)

  Convert a double to an unsigned 64-bit int in round-towards-zero mode.

- __device__ long long int __double_as_longlong (double x)

  Reinterpret bits in a double as a 64-bit signed integer.

- __device__ unsigned short __float2half_rn (float x)

  Convert a single-precision float to a half-precision float in round-to-nearest-even mode.

- __device__ int __float2int_rd (float x)

  Convert a float to a signed integer in round-down mode.

- __device__ int __float2int_rn (float x)

  Convert a float to a signed integer in round-to-nearest-even mode.

- __device__ int __float2int_ru (float)

  Convert a float to a signed integer in round-up mode.

- __device__ int __float2int_rz (float x)

  Convert a float to a signed integer in round-towards-zero mode.

- __device__ long long int __float2ll_rd (float x)

  Convert a float to a signed 64-bit integer in round-down mode.

- __device__ long long int __float2ll_rn (float x)

  Convert a float to a signed 64-bit integer in round-to-nearest-even mode.

- __device__ long long int __float2ll_ru (float x)

  Convert a float to a signed 64-bit integer in round-up mode.

- __device__ long long int __float2ll_rz (float x)

  Convert a float to a signed 64-bit integer in round-towards-zero mode.

- __device__ unsigned int __float2uint_rd (float x)
  Convert a float to an unsigned integer in round-down mode.
- __device__ unsigned int __float2uint_rn (float x)
  Convert a float to an unsigned integer in round-to-nearest-even mode.
- __device__ unsigned int __float2uint_ru (float x)
  Convert a float to an unsigned integer in round-up mode.
- __device__ unsigned int __float2uint_rz (float x)
  Convert a float to an unsigned integer in round-towards-zero mode.
- __device__ unsigned long long int __float2ull_rd (float x)
  Convert a float to an unsigned 64-bit integer in round-down mode.
- __device__ unsigned long long int __float2ull_rn (float x)
  Convert a float to an unsigned 64-bit integer in round-to-nearest-even mode.
- __device__ unsigned long long int __float2ull_ru (float x)
  Convert a float to an unsigned 64-bit integer in round-up mode.
- __device__ unsigned long long int __float2ull_rz (float x)
  Convert a float to an unsigned 64-bit integer in round-towards-zero mode.
- __device__ int __float_as_int (float x)
  Reinterpret bits in a float as a signed integer.
- __device__ float __half2float (unsigned short x)
  Convert a half-precision float to a single-precision float in round-to-nearest-even mode.
- __device__ double __hiloint2double (int hi, int lo)
  Reinterpret high and low 32-bit integer values as a double.
- __device__ double __int2double_rn (int x)
  Convert a signed int to a double.
- __device__ float __int2float_rd (int x)
  Convert a signed integer to a float in round-down mode.
- __device__ float __int2float_rn (int x)
  Convert a signed integer to a float in round-to-nearest-even mode.
- __device__ float __int2float_ru (int x)
  Convert a signed integer to a float in round-up mode.
- __device__ float __int2float_rz (int x)
  Convert a signed integer to a float in round-towards-zero mode.
- __device__ float __int_as_float (int x)
  Reinterpret bits in an integer as a float.

- __device__ double __ll2double_rd (long long int x)
  Convert a signed 64-bit int to a double in round-down mode.
- __device__ double __ll2double_rn (long long int x)
  Convert a signed 64-bit int to a double in round-to-nearest-even mode.
- __device__ double __ll2double_ru (long long int x)
  Convert a signed 64-bit int to a double in round-up mode.
- __device__ double __ll2double_rz (long long int x)
  Convert a signed 64-bit int to a double in round-towards-zero mode.
- __device__ float __ll2float_rd (long long int x)
  Convert a signed integer to a float in round-down mode.
- __device__ float __ll2float_rn (long long int x)
  Convert a signed 64-bit integer to a float in round-to-nearest-even mode.
- __device__ float __ll2float_ru (long long int x)
  Convert a signed integer to a float in round-up mode.
- __device__ float __ll2float_rz (long long int x)
  Convert a signed integer to a float in round-towards-zero mode.
- __device__ double __longlong_as_double (long long int x)
  Reinterpret bits in a 64-bit signed integer as a double.
- __device__ double __uint2double_rn (unsigned int x)
  Convert an unsigned int to a double.
- __device__ float __uint2float_rd (unsigned int x)
  Convert an unsigned integer to a float in round-down mode.
- __device__ float __uint2float_rn (unsigned int x)
  Convert an unsigned integer to a float in round-to-nearest-even mode.
- __device__ float __uint2float_ru (unsigned int x)
  Convert an unsigned integer to a float in round-up mode.
- __device__ float __uint2float_rz (unsigned int x)
  Convert an unsigned integer to a float in round-towards-zero mode.
- __device__ double __ull2double_rd (unsigned long long int x)
  Convert an unsigned 64-bit int to a double in round-down mode.
- __device__ double __ull2double_rn (unsigned long long int x)
  Convert an unsigned 64-bit int to a double in round-to-nearest-even mode.
- __device__ double __ull2double_ru (unsigned long long int x)

Convert an unsigned 64-bit int to a double in round-up mode.

- __device__ double __ull2double_rz (unsigned long long int x)
  Convert an unsigned 64-bit int to a double in round-towards-zero mode.
- __device__ float __ull2float_rd (unsigned long long int x)
  Convert an unsigned integer to a float in round-down mode.
- __device__ float __ull2float_rn (unsigned long long int x)
  Convert an unsigned integer to a float in round-to-nearest-even mode.
- __device__ float __ull2float_ru (unsigned long long int x)
  Convert an unsigned integer to a float in round-up mode.
- __device__ float __ull2float_rz (unsigned long long int x)
  Convert an unsigned integer to a float in round-towards-zero mode.

5.66.1 Detailed Description

This section describes type casting intrinsic functions that are only supported in device code.

5.66.2 Function Documentation

5.66.2.1 __device__ float __double2float_rd (double *x*)

Convert the double-precision floating point value x to a single-precision floating point value in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.2 __device__ float __double2float_rn (double *x*)

Convert the double-precision floating point value x to a single-precision floating point value in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.3 __device__ float __double2float_ru (double *x*)

Convert the double-precision floating point value x to a single-precision floating point value in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.4 __device__ float __double2float_rz (double *x*)

Convert the double-precision floating point value x to a single-precision floating point value in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.5 __device__ int __double2hiint (double *x*)

Reinterpret the high 32 bits in the double-precision floating point value x as a signed integer.

Returns:

Returns reinterpreted value.

5.66.2.6 __device__ int __double2int_rd (double *x*)

Convert the double-precision floating point value x to a signed integer value in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.7 __device__ int __double2int_rn (double *x*)

Convert the double-precision floating point value x to a signed integer value in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.8 __device__ int __double2int_ru (double *x*)

Convert the double-precision floating point value x to a signed integer value in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.9 __device__ int __double2int_rz (double)

Convert the double-precision floating point value x to a signed integer value in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.10 __device__ long long int __double2ll_rd (double *x*)

Convert the double-precision floating point value x to a signed 64-bit integer value in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.11 __device__ long long int __double2ll_rn (double *x*)

Convert the double-precision floating point value x to a signed 64-bit integer value in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.12 __device__ long long int __double2ll_ru (double *x*)

Convert the double-precision floating point value x to a signed 64-bit integer value in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.13 __device__ long long int __double2ll_rz (double)

Convert the double-precision floating point value x to a signed 64-bit integer value in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.14 __device__ int __double2loint (double *x*)

Reinterpret the low 32 bits in the double-precision floating point value x as a signed integer.

Returns:

Returns reinterpreted value.

5.66.2.15 __device__ unsigned int __double2uint_rd (double *x*)

Convert the double-precision floating point value x to an unsigned integer value in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.16 __device__ unsigned int __double2uint_rn (double *x*)

Convert the double-precision floating point value x to an unsigned integer value in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.17 __device__ unsigned int __double2uint_ru (double *x*)

Convert the double-precision floating point value x to an unsigned integer value in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.18 __device__ unsigned int __double2uint_rz (double)

Convert the double-precision floating point value x to an unsigned integer value in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.19 __device__ unsigned long long int __double2ull_rd (double *x*)

Convert the double-precision floating point value x to an unsigned 64-bit integer value in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.20 __device__ unsigned long long int __double2ull_rn (double *x*)

Convert the double-precision floating point value x to an unsigned 64-bit integer value in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.21 __device__ unsigned long long int __double2ull_ru (double *x*)

Convert the double-precision floating point value x to an unsigned 64-bit integer value in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.22 __device__ unsigned long long int __double2ull_rz (double)

Convert the double-precision floating point value `x` to an unsigned 64-bit integer value in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.23 __device__ long long int __double_as_longlong (double *x*)

Reinterpret the bits in the double-precision floating point value `x` as a signed 64-bit integer.

Returns:

Returns reinterpreted value.

5.66.2.24 __device__ unsigned short __float2half_rn (float *x*)

Convert the single-precision float value `x` to a half-precision floating point value represented in `unsigned short` format, in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.25 __device__ int __float2int_rd (float *x*)

Convert the single-precision floating point value `x` to a signed integer in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.26 __device__ int __float2int_rn (float *x*)

Convert the single-precision floating point value `x` to a signed integer in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.27 __device__ int __float2int_ru (float)

Convert the single-precision floating point value `x` to a signed integer in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.28 __device__ int __float2int_rz (float *x*)

Convert the single-precision floating point value x to a signed integer in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.29 __device__ long long int __float2ll_rd (float *x*)

Convert the single-precision floating point value x to a signed 64-bit integer in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.30 __device__ long long int __float2ll_rn (float *x*)

Convert the single-precision floating point value x to a signed 64-bit integer in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.31 __device__ long long int __float2ll_ru (float *x*)

Convert the single-precision floating point value x to a signed 64-bit integer in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.32 __device__ long long int __float2ll_rz (float *x*)

Convert the single-precision floating point value x to a signed 64-bit integer in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.33 __device__ unsigned int __float2uint_rd (float *x*)

Convert the single-precision floating point value x to an unsigned integer in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.34 __device__ unsigned int __float2uint_rn (float *x*)

Convert the single-precision floating point value x to an unsigned integer in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.35 __device__ unsigned int __float2uint_ru (float *x*)

Convert the single-precision floating point value x to an unsigned integer in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.36 __device__ unsigned int __float2uint_rz (float *x*)

Convert the single-precision floating point value x to an unsigned integer in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.37 __device__ unsigned long long int __float2ull_rd (float *x*)

Convert the single-precision floating point value x to an unsigned 64-bit integer in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.38 __device__ unsigned long long int __float2ull_rn (float *x*)

Convert the single-precision floating point value x to an unsigned 64-bit integer in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.39 __device__ unsigned long long int __float2ull_ru (float *x*)

Convert the single-precision floating point value x to an unsigned 64-bit integer in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.40 __device__ unsigned long long int __float2ull_rz (float *x*)

Convert the single-precision floating point value `x` to an unsigned 64-bit integer in round-towards_zero mode.

Returns:

Returns converted value.

5.66.2.41 __device__ int __float_as_int (float *x*)

Reinterpret the bits in the single-precision floating point value `x` as a signed integer.

Returns:

Returns reinterpreted value.

5.66.2.42 __device__ float __half2float (unsigned short *x*)

Convert the half-precision floating point value `x` represented in `unsigned short` format to a single-precision floating point value.

Returns:

Returns converted value.

5.66.2.43 __device__ double __hiloint2double (int *hi*, int *lo*)

Reinterpret the integer value of `hi` as the high 32 bits of a double-precision floating point value and the integer value of `lo` as the low 32 bits of the same double-precision floating point value.

Returns:

Returns reinterpreted value.

5.66.2.44 __device__ double __int2double_rn (int *x*)

Convert the signed integer value `x` to a double-precision floating point value.

Returns:

Returns converted value.

5.66.2.45 __device__ float __int2float_rd (int *x*)

Convert the signed integer value `x` to a single-precision floating point value in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.46 __device__ float __int2float_rn (int *x*)

Convert the signed integer value x to a single-precision floating point value in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.47 __device__ float __int2float_ru (int *x*)

Convert the signed integer value x to a single-precision floating point value in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.48 __device__ float __int2float_rz (int *x*)

Convert the signed integer value x to a single-precision floating point value in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.49 __device__ float __int_as_float (int *x*)

Reinterpret the bits in the signed integer value x as a single-precision floating point value.

Returns:

Returns reinterpreted value.

5.66.2.50 __device__ double __ll2double_rd (long long int *x*)

Convert the signed 64-bit integer value x to a double-precision floating point value in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.51 __device__ double __ll2double_rn (long long int *x*)

Convert the signed 64-bit integer value x to a double-precision floating point value in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.52 __device__ double __ll2double_ru (long long int *x*)

Convert the signed 64-bit integer value x to a double-precision floating point value in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.53 __device__ double __ll2double_rz (long long int *x*)

Convert the signed 64-bit integer value x to a double-precision floating point value in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.54 __device__ float __ll2float_rd (long long int *x*)

Convert the signed integer value x to a single-precision floating point value in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.55 __device__ float __ll2float_rn (long long int *x*)

Convert the signed 64-bit integer value x to a single-precision floating point value in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.56 __device__ float __ll2float_ru (long long int *x*)

Convert the signed integer value x to a single-precision floating point value in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.57 __device__ float __ll2float_rz (long long int *x*)

Convert the signed integer value x to a single-precision floating point value in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.58 __device__ double __longlong_as_double (long long int *x*)

Reinterpret the bits in the 64-bit signed integer value x as a double-precision floating point value.

Returns:

Returns reinterpreted value.

5.66.2.59 __device__ double __uint2double_rn (unsigned int *x*)

Convert the unsigned integer value x to a double-precision floating point value.

Returns:

Returns converted value.

5.66.2.60 __device__ float __uint2float_rd (unsigned int *x*)

Convert the unsigned integer value x to a single-precision floating point value in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.61 __device__ float __uint2float_rn (unsigned int *x*)

Convert the unsigned integer value x to a single-precision floating point value in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.62 __device__ float __uint2float_ru (unsigned int *x*)

Convert the unsigned integer value x to a single-precision floating point value in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.63 __device__ float __uint2float_rz (unsigned int *x*)

Convert the unsigned integer value x to a single-precision floating point value in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.64 __device__ double __ull2double_rd (unsigned long long int *x*)

Convert the unsigned 64-bit integer value x to a double-precision floating point value in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.65 __device__ double __ull2double_rn (unsigned long long int *x*)

Convert the unsigned 64-bit integer value x to a double-precision floating point value in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.66 __device__ double __ull2double_ru (unsigned long long int *x*)

Convert the unsigned 64-bit integer value x to a double-precision floating point value in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.67 __device__ double __ull2double_rz (unsigned long long int *x*)

Convert the unsigned 64-bit integer value x to a double-precision floating point value in round-towards-zero mode.

Returns:

Returns converted value.

5.66.2.68 __device__ float __ull2float_rd (unsigned long long int *x*)

Convert the unsigned integer value x to a single-precision floating point value in round-down (to negative infinity) mode.

Returns:

Returns converted value.

5.66.2.69 __device__ float __ull2float_rn (unsigned long long int *x*)

Convert the unsigned integer value x to a single-precision floating point value in round-to-nearest-even mode.

Returns:

Returns converted value.

5.66.2.70 __device__ float __ull2float_ru (unsigned long long int *x*)

Convert the unsigned integer value x to a single-precision floating point value in round-up (to positive infinity) mode.

Returns:

Returns converted value.

5.66.2.71 __device__ float __ull2float_rz (unsigned long long int *x*)

Convert the unsigned integer value x to a single-precision floating point value in round-towards-zero mode.

Returns:

Returns converted value.

Chapter 6

Data Structure Documentation

6.1 CUDA_ARRAY3D_DESCRIPTOR_st Struct Reference

Data Fields

- size_t Depth
- unsigned int Flags
- CUarray_format Format
- size_t Height
- unsigned int NumChannels
- size_t Width

6.1.1 Detailed Description

3D array descriptor

6.1.2 Field Documentation

6.1.2.1 size_t CUDA_ARRAY3D_DESCRIPTOR_st::Depth

Depth of 3D array

6.1.2.2 unsigned int CUDA_ARRAY3D_DESCRIPTOR_st::Flags

Flags

6.1.2.3 CUarray_format CUDA_ARRAY3D_DESCRIPTOR_st::Format

Array format

6.1.2.4 size_t CUDA_ARRAY3D_DESCRIPTOR_st::Height

Height of 3D array

6.1.2.5 unsigned int CUDA_ARRAY3D_DESCRIPTOR_st::NumChannels

Channels per array element

6.1.2.6 size_t CUDA_ARRAY3D_DESCRIPTOR_st::Width

Width of 3D array

6.2 CUDA_ARRAY_DESCRIPTOR_st Struct Reference

Data Fields

- CUarray_format Format
- size_t Height
- unsigned int NumChannels
- size_t Width

6.2.1 Detailed Description

Array descriptor

6.2.2 Field Documentation

6.2.2.1 CUarray_format CUDA_ARRAY_DESCRIPTOR_st::Format

Array format

6.2.2.2 size_t CUDA_ARRAY_DESCRIPTOR_st::Height

Height of array

6.2.2.3 unsigned int CUDA_ARRAY_DESCRIPTOR_st::NumChannels

Channels per array element

6.2.2.4 size_t CUDA_ARRAY_DESCRIPTOR_st::Width

Width of array

6.3 CUDA_MEMCPY2D_st Struct Reference

Data Fields

- CUarray dstArray
- CUdeviceptr dstDevice
- void * dstHost
- CUmemorytype dstMemoryType
- size_t dstPitch
- size_t dstXInBytes
- size_t dstY
- size_t Height
- CUarray srcArray
- CUdeviceptr srcDevice
- const void * srcHost
- CUmemorytype srcMemoryType
- size_t srcPitch
- size_t srcXInBytes
- size_t srcY
- size_t WidthInBytes

6.3.1 Detailed Description

2D memory copy parameters

6.3.2 Field Documentation

6.3.2.1 CUarray CUDA_MEMCPY2D_st::dstArray

Destination array reference

6.3.2.2 CUdeviceptr CUDA_MEMCPY2D_st::dstDevice

Destination device pointer

6.3.2.3 void* CUDA_MEMCPY2D_st::dstHost

Destination host pointer

6.3.2.4 CUmemorytype CUDA_MEMCPY2D_st::dstMemoryType

Destination memory type (host, device, array)

6.3.2.5 size_t CUDA_MEMCPY2D_st::dstPitch

Destination pitch (ignored when dst is array)

6.3.2.6 size_t CUDA_MEMCPY2D_st::dstXInBytes

Destination X in bytes

6.3.2.7 size_t CUDA_MEMCPY2D_st::dstY

Destination Y

6.3.2.8 size_t CUDA_MEMCPY2D_st::Height

Height of 2D memory copy

6.3.2.9 CUarray CUDA_MEMCPY2D_st::srcArray

Source array reference

6.3.2.10 CUdeviceptr CUDA_MEMCPY2D_st::srcDevice

Source device pointer

6.3.2.11 const void* CUDA_MEMCPY2D_st::srcHost

Source host pointer

6.3.2.12 CUmemorytype CUDA_MEMCPY2D_st::srcMemoryType

Source memory type (host, device, array)

6.3.2.13 size_t CUDA_MEMCPY2D_st::srcPitch

Source pitch (ignored when src is array)

6.3.2.14 size_t CUDA_MEMCPY2D_st::srcXInBytes

Source X in bytes

6.3.2.15 size_t CUDA_MEMCPY2D_st::srcY

Source Y

6.3.2.16 size_t CUDA_MEMCPY2D_st::WidthInBytes

Width of 2D memory copy in bytes

6.4 CUDA_MEMCPY3D_PEER_st Struct Reference

Data Fields

- size_t Depth
- CUarray dstArray
- CUcontext dstContext
- CUdeviceptr dstDevice
- size_t dstHeight
- void * dstHost
- size_t dstLOD
- CUmemorytype dstMemoryType
- size_t dstPitch
- size_t dstXInBytes
- size_t dstY
- size_t dstZ
- size_t Height
- CUarray srcArray
- CUcontext srcContext
- CUdeviceptr srcDevice
- size_t srcHeight
- const void * srcHost
- size_t srcLOD
- CUmemorytype srcMemoryType
- size_t srcPitch
- size_t srcXInBytes
- size_t srcY
- size_t srcZ
- size_t WidthInBytes

6.4.1 Detailed Description

3D memory cross-context copy parameters

6.4.2 Field Documentation

6.4.2.1 size_t CUDA_MEMCPY3D_PEER_st::Depth

Depth of 3D memory copy

6.4.2.2 CUarray CUDA_MEMCPY3D_PEER_st::dstArray

Destination array reference

6.4.2.3 CUcontext CUDA_MEMCPY3D_PEER_st::dstContext

Destination context (ignored with dstMemoryType is CU_MEMORYTYPE_ARRAY)

6.4.2.4 CUdeviceptr CUDA_MEMCPY3D_PEER_st::dstDevice

Destination device pointer

6.4.2.5 size_t CUDA_MEMCPY3D_PEER_st::dstHeight

Destination height (ignored when dst is array; may be 0 if Depth==1)

6.4.2.6 void* CUDA_MEMCPY3D_PEER_st::dstHost

Destination host pointer

6.4.2.7 size_t CUDA_MEMCPY3D_PEER_st::dstLOD

Destination LOD

6.4.2.8 CUmemorytype CUDA_MEMCPY3D_PEER_st::dstMemoryType

Destination memory type (host, device, array)

6.4.2.9 size_t CUDA_MEMCPY3D_PEER_st::dstPitch

Destination pitch (ignored when dst is array)

6.4.2.10 size_t CUDA_MEMCPY3D_PEER_st::dstXInBytes

Destination X in bytes

6.4.2.11 size_t CUDA_MEMCPY3D_PEER_st::dstY

Destination Y

6.4.2.12 size_t CUDA_MEMCPY3D_PEER_st::dstZ

Destination Z

6.4.2.13 size_t CUDA_MEMCPY3D_PEER_st::Height

Height of 3D memory copy

6.4.2.14 CUarray CUDA_MEMCPY3D_PEER_st::srcArray

Source array reference

6.4.2.15 CUcontext CUDA_MEMCPY3D_PEER_st::srcContext

Source context (ignored with srcMemoryType is CU_MEMORYTYPE_ARRAY)

6.4.2.16 CUdeviceptr CUDA_MEMCPY3D_PEER_st::srcDevice

Source device pointer

6.4.2.17 size_t CUDA_MEMCPY3D_PEER_st::srcHeight

Source height (ignored when src is array; may be 0 if Depth==1)

6.4.2.18 const void* CUDA_MEMCPY3D_PEER_st::srcHost

Source host pointer

6.4.2.19 size_t CUDA_MEMCPY3D_PEER_st::srcLOD

Source LOD

6.4.2.20 CUmemorytype CUDA_MEMCPY3D_PEER_st::srcMemoryType

Source memory type (host, device, array)

6.4.2.21 size_t CUDA_MEMCPY3D_PEER_st::srcPitch

Source pitch (ignored when src is array)

6.4.2.22 size_t CUDA_MEMCPY3D_PEER_st::srcXInBytes

Source X in bytes

6.4.2.23 size_t CUDA_MEMCPY3D_PEER_st::srcY

Source Y

6.4.2.24 size_t CUDA_MEMCPY3D_PEER_st::srcZ

Source Z

6.4.2.25 size_t CUDA_MEMCPY3D_PEER_st::WidthInBytes

Width of 3D memory copy in bytes

6.5 CUDA_MEMCPY3D_st Struct Reference

Data Fields

- size_t Depth
- CUarray dstArray
- CUdeviceptr dstDevice
- size_t dstHeight
- void * dstHost
- size_t dstLOD
- CUmemorytype dstMemoryType
- size_t dstPitch
- size_t dstXInBytes
- size_t dstY
- size_t dstZ
- size_t Height
- void * reserved0
- void * reserved1
- CUarray srcArray
- CUdeviceptr srcDevice
- size_t srcHeight
- const void * srcHost
- size_t srcLOD
- CUmemorytype srcMemoryType
- size_t srcPitch
- size_t srcXInBytes
- size_t srcY
- size_t srcZ
- size_t WidthInBytes

6.5.1 Detailed Description

3D memory copy parameters

6.5.2 Field Documentation

6.5.2.1 size_t CUDA_MEMCPY3D_st::Depth

Depth of 3D memory copy

6.5.2.2 CUarray CUDA_MEMCPY3D_st::dstArray

Destination array reference

6.5.2.3 CUdeviceptr CUDA_MEMCPY3D_st::dstDevice

Destination device pointer

6.5.2.4 size_t CUDA_MEMCPY3D_st::dstHeight

Destination height (ignored when dst is array; may be 0 if Depth==1)

6.5.2.5 void* CUDA_MEMCPY3D_st::dstHost

Destination host pointer

6.5.2.6 size_t CUDA_MEMCPY3D_st::dstLOD

Destination LOD

6.5.2.7 CUmemorytype CUDA_MEMCPY3D_st::dstMemoryType

Destination memory type (host, device, array)

6.5.2.8 size_t CUDA_MEMCPY3D_st::dstPitch

Destination pitch (ignored when dst is array)

6.5.2.9 size_t CUDA_MEMCPY3D_st::dstXInBytes

Destination X in bytes

6.5.2.10 size_t CUDA_MEMCPY3D_st::dstY

Destination Y

6.5.2.11 size_t CUDA_MEMCPY3D_st::dstZ

Destination Z

6.5.2.12 size_t CUDA_MEMCPY3D_st::Height

Height of 3D memory copy

6.5.2.13 void* CUDA_MEMCPY3D_st::reserved0

Must be NULL

6.5.2.14 void* CUDA_MEMCPY3D_st::reserved1

Must be NULL

6.5.2.15 CUarray CUDA_MEMCPY3D_st::srcArray

Source array reference

6.5.2.16 CUdeviceptr CUDA_MEMCPY3D_st::srcDevice

Source device pointer

6.5.2.17 size_t CUDA_MEMCPY3D_st::srcHeight

Source height (ignored when src is array; may be 0 if Depth==1)

6.5.2.18 const void* CUDA_MEMCPY3D_st::srcHost

Source host pointer

6.5.2.19 size_t CUDA_MEMCPY3D_st::srcLOD

Source LOD

6.5.2.20 CUmemorytype CUDA_MEMCPY3D_st::srcMemoryType

Source memory type (host, device, array)

6.5.2.21 size_t CUDA_MEMCPY3D_st::srcPitch

Source pitch (ignored when src is array)

6.5.2.22 size_t CUDA_MEMCPY3D_st::srcXInBytes

Source X in bytes

6.5.2.23 size_t CUDA_MEMCPY3D_st::srcY

Source Y

6.5.2.24 size_t CUDA_MEMCPY3D_st::srcZ

Source Z

6.5.2.25 size_t CUDA_MEMCPY3D_st::WidthInBytes

Width of 3D memory copy in bytes

6.6 CUDA_POINTER_ATTRIBUTE_P2P_TOKENS_st Struct Reference

6.6.1 Detailed Description

GPU Direct v3 tokens

6.7 CUDA_RESOURCE_DESC_st Struct Reference

Data Fields

- unsigned int flags
- CUresourcetype resType
- CUdeviceptr devPtr
- CUarray_format format
- CUarray hArray
- size_t height
- CUmipmappedArray hMipmappedArray
- unsigned int numChannels
- size_t pitchInBytes
- size_t sizeInBytes
- size_t width

6.7.1 Detailed Description

CUDA Resource descriptor

6.7.2 Field Documentation

6.7.2.1 CUdeviceptr CUDA_RESOURCE_DESC_st::devPtr

Device pointer

6.7.2.2 unsigned int CUDA_RESOURCE_DESC_st::flags

Flags (must be zero)

6.7.2.3 CUarray_format CUDA_RESOURCE_DESC_st::format

Array format

6.7.2.4 CUarray CUDA_RESOURCE_DESC_st::hArray

CUDA array

6.7.2.5 size_t CUDA_RESOURCE_DESC_st::height

Height of the array in elements

6.7.2.6 CUmipmappedArray CUDA_RESOURCE_DESC_st::hMipmappedArray

CUDA mipmapped array

6.7.2.7 unsigned int CUDA_RESOURCE_DESC_st::numChannels

Channels per array element

6.7.2.8 size_t CUDA_RESOURCE_DESC_st::pitchInBytes

Pitch between two rows in bytes

6.7.2.9 CUresourcetype CUDA_RESOURCE_DESC_st::resType

Resource type

6.7.2.10 size_t CUDA_RESOURCE_DESC_st::sizeInBytes

Size in bytes

6.7.2.11 size_t CUDA_RESOURCE_DESC_st::width

Width of the array in elements

6.8 CUDA_RESOURCE_VIEW_DESC_st Struct Reference

Data Fields

- size_t depth
- unsigned int firstLayer
- unsigned int firstMipmapLevel
- CUresourceViewFormat format
- size_t height
- unsigned int lastLayer
- unsigned int lastMipmapLevel
- size_t width

6.8.1 Detailed Description

Resource view descriptor

6.8.2 Field Documentation

6.8.2.1 size_t CUDA_RESOURCE_VIEW_DESC_st::depth

Depth of the resource view

6.8.2.2 unsigned int CUDA_RESOURCE_VIEW_DESC_st::firstLayer

First layer index

6.8.2.3 unsigned int CUDA_RESOURCE_VIEW_DESC_st::firstMipmapLevel

First defined mipmap level

6.8.2.4 CUresourceViewFormat CUDA_RESOURCE_VIEW_DESC_st::format

Resource view format

6.8.2.5 size_t CUDA_RESOURCE_VIEW_DESC_st::height

Height of the resource view

6.8.2.6 unsigned int CUDA_RESOURCE_VIEW_DESC_st::lastLayer

Last layer index

6.8.2.7 unsigned int CUDA_RESOURCE_VIEW_DESC_st::lastMipmapLevel

Last defined mipmap level

6.8.2.8 size_t CUDA_RESOURCE_VIEW_DESC_st::width

Width of the resource view

6.9 CUDA_TEXTURE_DESC_st Struct Reference

Data Fields

- CUaddress_mode addressMode [3]
- CUfilter_mode filterMode
- unsigned int flags
- unsigned int maxAnisotropy
- float maxMipmapLevelClamp
- float minMipmapLevelClamp
- CUfilter_mode mipmapFilterMode
- float mipmapLevelBias

6.9.1 Detailed Description

Texture descriptor

6.9.2 Field Documentation

6.9.2.1 CUaddress_mode CUDA_TEXTURE_DESC_st::addressMode[3]

Address modes

6.9.2.2 CUfilter_mode CUDA_TEXTURE_DESC_st::filterMode

Filter mode

6.9.2.3 unsigned int CUDA_TEXTURE_DESC_st::flags

Flags

6.9.2.4 unsigned int CUDA_TEXTURE_DESC_st::maxAnisotropy

Maximum anistropy ratio

6.9.2.5 float CUDA_TEXTURE_DESC_st::maxMipmapLevelClamp

Mipmap maximum level clamp

6.9.2.6 float CUDA_TEXTURE_DESC_st::minMipmapLevelClamp

Mipmap minimum level clamp

6.9.2.7 CUfilter_mode CUDA_TEXTURE_DESC_st::mipmapFilterMode

Mipmap filter mode

6.9.2.8 float CUDA_TEXTURE_DESC_st::mipmapLevelBias

Mipmap level bias

6.10 cudaChannelFormatDesc Struct Reference

Data Fields

- enum cudaChannelFormatKind f
- int w
- int x
- int y
- int z

6.10.1 Detailed Description

CUDA Channel format descriptor

6.10.2 Field Documentation

6.10.2.1 enum cudaChannelFormatKind cudaChannelFormatDesc::f

Channel format kind

6.10.2.2 int cudaChannelFormatDesc::w

w

6.10.2.3 int cudaChannelFormatDesc::x

x

6.10.2.4 int cudaChannelFormatDesc::y

y

6.10.2.5 int cudaChannelFormatDesc::z

z

6.11 cudaDeviceProp Struct Reference

Data Fields

- int asyncEngineCount
- int canMapHostMemory
- int clockRate
- int computeMode
- int concurrentKernels
- int deviceOverlap
- int ECCEnabled
- int integrated
- int kernelExecTimeoutEnabled
- int l2CacheSize
- int major
- int maxGridSize [3]
- int maxSurface1D
- int maxSurface1DLayered [2]
- int maxSurface2D [2]
- int maxSurface2DLayered [3]
- int maxSurface3D [3]
- int maxSurfaceCubemap
- int maxSurfaceCubemapLayered [2]
- int maxTexture1D
- int maxTexture1DLayered [2]
- int maxTexture1DLinear
- int maxTexture1DMipmap
- int maxTexture2D [2]
- int maxTexture2DGather [2]
- int maxTexture2DLayered [3]
- int maxTexture2DLinear [3]
- int maxTexture2DMipmap [2]
- int maxTexture3D [3]
- int maxTextureCubemap
- int maxTextureCubemapLayered [2]
- int maxThreadsDim [3]
- int maxThreadsPerBlock
- int maxThreadsPerMultiProcessor
- int memoryBusWidth
- int memoryClockRate
- size_t memPitch
- int minor
- int multiProcessorCount
- char name [256]
- int pciBusID
- int pciDeviceID
- int pciDomainID
- int regsPerBlock
- size_t sharedMemPerBlock
- size_t surfaceAlignment

- int tccDriver
- size_t textureAlignment
- size_t texturePitchAlignment
- size_t totalConstMem
- size_t totalGlobalMem
- int unifiedAddressing
- int warpSize

6.11.1 Detailed Description

CUDA device properties

6.11.2 Field Documentation

6.11.2.1 int cudaDeviceProp::asyncEngineCount

Number of asynchronous engines

6.11.2.2 int cudaDeviceProp::canMapHostMemory

Device can map host memory with cudaHostAlloc/cudaHostGetDevicePointer

6.11.2.3 int cudaDeviceProp::clockRate

Clock frequency in kilohertz

6.11.2.4 int cudaDeviceProp::computeMode

Compute mode (See cudaComputeMode)

6.11.2.5 int cudaDeviceProp::concurrentKernels

Device can possibly execute multiple kernels concurrently

6.11.2.6 int cudaDeviceProp::deviceOverlap

Device can concurrently copy memory and execute a kernel. Deprecated. Use instead asyncEngineCount.

6.11.2.7 int cudaDeviceProp::ECCEnabled

Device has ECC support enabled

6.11.2.8 int cudaDeviceProp::integrated

Device is integrated as opposed to discrete

6.11.2.9 int cudaDeviceProp::kernelExecTimeoutEnabled

Specified whether there is a run time limit on kernels

6.11.2.10 int cudaDeviceProp::l2CacheSize

Size of L2 cache in bytes

6.11.2.11 int cudaDeviceProp::major

Major compute capability

6.11.2.12 int cudaDeviceProp::maxGridSize[3]

Maximum size of each dimension of a grid

6.11.2.13 int cudaDeviceProp::maxSurface1D

Maximum 1D surface size

6.11.2.14 int cudaDeviceProp::maxSurface1DLayered[2]

Maximum 1D layered surface dimensions

6.11.2.15 int cudaDeviceProp::maxSurface2D[2]

Maximum 2D surface dimensions

6.11.2.16 int cudaDeviceProp::maxSurface2DLayered[3]

Maximum 2D layered surface dimensions

6.11.2.17 int cudaDeviceProp::maxSurface3D[3]

Maximum 3D surface dimensions

6.11.2.18 int cudaDeviceProp::maxSurfaceCubemap

Maximum Cubemap surface dimensions

6.11.2.19 int cudaDeviceProp::maxSurfaceCubemapLayered[2]

Maximum Cubemap layered surface dimensions

6.11.2.20 int cudaDeviceProp::maxTexture1D

Maximum 1D texture size

6.11.2.21 int cudaDeviceProp::maxTexture1DLayered[2]

Maximum 1D layered texture dimensions

6.11.2.22 int cudaDeviceProp::maxTexture1DLinear

Maximum size for 1D textures bound to linear memory

6.11.2.23 int cudaDeviceProp::maxTexture1DMipmap

Maximum 1D mipmapped texture size

6.11.2.24 int cudaDeviceProp::maxTexture2D[2]

Maximum 2D texture dimensions

6.11.2.25 int cudaDeviceProp::maxTexture2DGather[2]

Maximum 2D texture dimensions if texture gather operations have to be performed

6.11.2.26 int cudaDeviceProp::maxTexture2DLayered[3]

Maximum 2D layered texture dimensions

6.11.2.27 int cudaDeviceProp::maxTexture2DLinear[3]

Maximum dimensions (width, height, pitch) for 2D textures bound to pitched memory

6.11.2.28 int cudaDeviceProp::maxTexture2DMipmap[2]

Maximum 2D mipmapped texture dimensions

6.11.2.29 int cudaDeviceProp::maxTexture3D[3]

Maximum 3D texture dimensions

6.11.2.30 int cudaDeviceProp::maxTextureCubemap

Maximum Cubemap texture dimensions

6.11.2.31 int cudaDeviceProp::maxTextureCubemapLayered[2]

Maximum Cubemap layered texture dimensions

6.11.2.32 int cudaDeviceProp::maxThreadsDim[3]

Maximum size of each dimension of a block

6.11.2.33 int cudaDeviceProp::maxThreadsPerBlock

Maximum number of threads per block

6.11.2.34 int cudaDeviceProp::maxThreadsPerMultiProcessor

Maximum resident threads per multiprocessor

6.11.2.35 int cudaDeviceProp::memoryBusWidth

Global memory bus width in bits

6.11.2.36 int cudaDeviceProp::memoryClockRate

Peak memory clock frequency in kilohertz

6.11.2.37 size_t cudaDeviceProp::memPitch

Maximum pitch in bytes allowed by memory copies

6.11.2.38 int cudaDeviceProp::minor

Minor compute capability

6.11.2.39 int cudaDeviceProp::multiProcessorCount

Number of multiprocessors on device

6.11.2.40 char cudaDeviceProp::name[256]

ASCII string identifying device

6.11.2.41 int cudaDeviceProp::pciBusID

PCI bus ID of the device

6.11.2.42 int cudaDeviceProp::pciDeviceID

PCI device ID of the device

6.11.2.43 int cudaDeviceProp::pciDomainID

PCI domain ID of the device

6.11.2.44 int cudaDeviceProp::regsPerBlock

32-bit registers available per block

6.11.2.45 size_t cudaDeviceProp::sharedMemPerBlock

Shared memory available per block in bytes

6.11.2.46 size_t cudaDeviceProp::surfaceAlignment

Alignment requirements for surfaces

6.11.2.47 int cudaDeviceProp::tccDriver

1 if device is a Tesla device using TCC driver, 0 otherwise

6.11.2.48 size_t cudaDeviceProp::textureAlignment

Alignment requirement for textures

6.11.2.49 size_t cudaDeviceProp::texturePitchAlignment

Pitch alignment requirement for texture references bound to pitched memory

6.11.2.50 size_t cudaDeviceProp::totalConstMem

Constant memory available on device in bytes

6.11.2.51 size_t cudaDeviceProp::totalGlobalMem

Global memory available on device in bytes

6.11.2.52 int cudaDeviceProp::unifiedAddressing

Device shares a unified address space with the host

6.11.2.53 int cudaDeviceProp::warpSize

Warp size in threads

6.12 cudaExtent Struct Reference

Data Fields

- size_t depth
- size_t height
- size_t width

6.12.1 Detailed Description

CUDA extent

See also:

make_cudaExtent

6.12.2 Field Documentation

6.12.2.1 size_t cudaExtent::depth

Depth in elements

6.12.2.2 size_t cudaExtent::height

Height in elements

6.12.2.3 size_t cudaExtent::width

Width in elements when referring to array memory, in bytes when referring to linear memory

6.13 cudaFuncAttributes Struct Reference

Data Fields

- int binaryVersion
- size_t constSizeBytes
- size_t localSizeBytes
- int maxThreadsPerBlock
- int numRegs
- int ptxVersion
- size_t sharedSizeBytes

6.13.1 Detailed Description

CUDA function attributes

6.13.2 Field Documentation

6.13.2.1 int cudaFuncAttributes::binaryVersion

The binary architecture version for which the function was compiled. This value is the major binary version * 10 + the minor binary version, so a binary version 1.3 function would return the value 13.

6.13.2.2 size_t cudaFuncAttributes::constSizeBytes

The size in bytes of user-allocated constant memory required by this function.

6.13.2.3 size_t cudaFuncAttributes::localSizeBytes

The size in bytes of local memory used by each thread of this function.

6.13.2.4 int cudaFuncAttributes::maxThreadsPerBlock

The maximum number of threads per block, beyond which a launch of the function would fail. This number depends on both the function and the device on which the function is currently loaded.

6.13.2.5 int cudaFuncAttributes::numRegs

The number of registers used by each thread of this function.

6.13.2.6 int cudaFuncAttributes::ptxVersion

The PTX virtual architecture version for which the function was compiled. This value is the major PTX version * 10 + the minor PTX version, so a PTX version 1.3 function would return the value 13.

6.13.2.7 size_t cudaFuncAttributes::sharedSizeBytes

The size in bytes of statically-allocated shared memory per block required by this function. This does not include dynamically-allocated shared memory requested by the user at runtime.

6.14 cudaIpcEventHandle_st Struct Reference

6.14.1 Detailed Description

CUDA IPC event handle

6.15 cudaIpcMemHandle_st Struct Reference

6.15.1 Detailed Description

CUDA IPC memory handle

6.16 cudaMemcpy3DParms Struct Reference

Data Fields

- cudaArray_t dstArray
- struct cudaPos dstPos
- struct cudaPitchedPtr dstPtr
- struct cudaExtent extent
- enum cudaMemcpyKind kind
- cudaArray_t srcArray
- struct cudaPos srcPos
- struct cudaPitchedPtr srcPtr

6.16.1 Detailed Description

CUDA 3D memory copying parameters

6.16.2 Field Documentation

6.16.2.1 cudaArray_t cudaMemcpy3DParms::dstArray

Destination memory address

6.16.2.2 struct cudaPos cudaMemcpy3DParms::dstPos [read]

Destination position offset

6.16.2.3 struct cudaPitchedPtr cudaMemcpy3DParms::dstPtr [read]

Pitched destination memory address

6.16.2.4 struct cudaExtent cudaMemcpy3DParms::extent [read]

Requested memory copy size

6.16.2.5 enum cudaMemcpyKind cudaMemcpy3DParms::kind

Type of transfer

6.16.2.6 cudaArray_t cudaMemcpy3DParms::srcArray

Source memory address

6.16.2.7 struct cudaPos cudaMemcpy3DParms::srcPos [read]

Source position offset

6.16.2.8 struct cudaPitchedPtr cudaMemcpy3DParms::srcPtr [read]

Pitched source memory address

6.17 cudaMemcpy3DPeerParms Struct Reference

Data Fields

- cudaArray_t dstArray
- int dstDevice
- struct cudaPos dstPos
- struct cudaPitchedPtr dstPtr
- struct cudaExtent extent
- cudaArray_t srcArray
- int srcDevice
- struct cudaPos srcPos
- struct cudaPitchedPtr srcPtr

6.17.1 Detailed Description

CUDA 3D cross-device memory copying parameters

6.17.2 Field Documentation

6.17.2.1 cudaArray_t cudaMemcpy3DPeerParms::dstArray

Destination memory address

6.17.2.2 int cudaMemcpy3DPeerParms::dstDevice

Destination device

6.17.2.3 struct cudaPos cudaMemcpy3DPeerParms::dstPos [read]

Destination position offset

6.17.2.4 struct cudaPitchedPtr cudaMemcpy3DPeerParms::dstPtr [read]

Pitched destination memory address

6.17.2.5 struct cudaExtent cudaMemcpy3DPeerParms::extent [read]

Requested memory copy size

6.17.2.6 cudaArray_t cudaMemcpy3DPeerParms::srcArray

Source memory address

6.17.2.7 int cudaMemcpy3DPeerParms::srcDevice

Source device

6.17.2.8 struct cudaPos cudaMemcpy3DPeerParms::srcPos [read]

Source position offset

6.17.2.9 struct cudaPitchedPtr cudaMemcpy3DPeerParms::srcPtr [read]

Pitched source memory address

6.18 cudaPitchedPtr Struct Reference

Data Fields

- size_t pitch
- void * ptr
- size_t xsize
- size_t ysize

6.18.1 Detailed Description

CUDA Pitched memory pointer

See also:

make_cudaPitchedPtr

6.18.2 Field Documentation

6.18.2.1 size_t cudaPitchedPtr::pitch

Pitch of allocated memory in bytes

6.18.2.2 void* cudaPitchedPtr::ptr

Pointer to allocated memory

6.18.2.3 size_t cudaPitchedPtr::xsize

Logical width of allocation in elements

6.18.2.4 size_t cudaPitchedPtr::ysize

Logical height of allocation in elements

6.19 cudaPointerAttributes Struct Reference

Data Fields

- int device
- void * devicePointer
- void * hostPointer
- enum cudaMemoryType memoryType

6.19.1 Detailed Description

CUDA pointer attributes

6.19.2 Field Documentation

6.19.2.1 int cudaPointerAttributes::device

The device against which the memory was allocated or registered. If the memory type is cudaMemoryTypeDevice then this identifies the device on which the memory referred physically resides. If the memory type is cudaMemoryTypeHost then this identifies the device which was current when the memory was allocated or registered (and if that device is deinitialized then this allocation will vanish with that device's state).

6.19.2.2 void* cudaPointerAttributes::devicePointer

The address which may be dereferenced on the current device to access the memory or NULL if no such address exists.

6.19.2.3 void* cudaPointerAttributes::hostPointer

The address which may be dereferenced on the host to access the memory or NULL if no such address exists.

6.19.2.4 enum cudaMemoryType cudaPointerAttributes::memoryType

The physical location of the memory, cudaMemoryTypeHost or cudaMemoryTypeDevice.

6.20 cudaPos Struct Reference

Data Fields

- size_t x
- size_t y
- size_t z

6.20.1 Detailed Description

CUDA 3D position

See also:

make_cudaPos

6.20.2 Field Documentation

6.20.2.1 size_t cudaPos::x

x

6.20.2.2 size_t cudaPos::y

y

6.20.2.3 size_t cudaPos::z

z

6.21 cudaResourceDesc Struct Reference

Data Fields

- enum cudaResourceType resType
- cudaArray_t array
- struct cudaChannelFormatDesc desc
- void * devPtr
- size_t height
- cudaMipmappedArray_t mipmap
- size_t pitchInBytes
- size_t sizeInBytes
- size_t width

6.21.1 Detailed Description

CUDA resource descriptor

6.21.2 Field Documentation

6.21.2.1 cudaArray_t cudaResourceDesc::array

CUDA array

6.21.2.2 struct cudaChannelFormatDesc cudaResourceDesc::desc [read]

Channel descriptor

6.21.2.3 void* cudaResourceDesc::devPtr

Device pointer

6.21.2.4 size_t cudaResourceDesc::height

Height of the array in elements

6.21.2.5 cudaMipmappedArray_t cudaResourceDesc::mipmap

CUDA mipmapped array

6.21.2.6 size_t cudaResourceDesc::pitchInBytes

Pitch between two rows in bytes

6.21.2.7 enum cudaResourceType cudaResourceDesc::resType

Resource type

6.21.2.8 size_t cudaResourceDesc::sizeInBytes

Size in bytes

6.21.2.9 size_t cudaResourceDesc::width

Width of the array in elements

6.22 cudaResourceViewDesc Struct Reference

Data Fields

- size_t depth
- unsigned int firstLayer
- unsigned int firstMipmapLevel
- enum cudaResourceViewFormat format
- size_t height
- unsigned int lastLayer
- unsigned int lastMipmapLevel
- size_t width

6.22.1 Detailed Description

CUDA resource view descriptor

6.22.2 Field Documentation

6.22.2.1 size_t cudaResourceViewDesc::depth

Depth of the resource view

6.22.2.2 unsigned int cudaResourceViewDesc::firstLayer

First layer index

6.22.2.3 unsigned int cudaResourceViewDesc::firstMipmapLevel

First defined mipmap level

6.22.2.4 enum cudaResourceViewFormat cudaResourceViewDesc::format

Resource view format

6.22.2.5 size_t cudaResourceViewDesc::height

Height of the resource view

6.22.2.6 unsigned int cudaResourceViewDesc::lastLayer

Last layer index

6.22.2.7 unsigned int cudaResourceViewDesc::lastMipmapLevel

Last defined mipmap level

6.22.2.8 size_t cudaResourceViewDesc::width

Width of the resource view

6.23 cudaTextureDesc Struct Reference

Data Fields

- enum cudaTextureAddressMode addressMode [3]
- enum cudaTextureFilterMode filterMode
- unsigned int maxAnisotropy
- float maxMipmapLevelClamp
- float minMipmapLevelClamp
- enum cudaTextureFilterMode mipmapFilterMode
- float mipmapLevelBias
- int normalizedCoords
- enum cudaTextureReadMode readMode
- int sRGB

6.23.1 Detailed Description

CUDA texture descriptor

6.23.2 Field Documentation

6.23.2.1 enum cudaTextureAddressMode cudaTextureDesc::addressMode[3]

Texture address mode for up to 3 dimensions

6.23.2.2 enum cudaTextureFilterMode cudaTextureDesc::filterMode

Texture filter mode

6.23.2.3 unsigned int cudaTextureDesc::maxAnisotropy

Limit to the anisotropy ratio

6.23.2.4 float cudaTextureDesc::maxMipmapLevelClamp

Upper end of the mipmap level range to clamp access to

6.23.2.5 float cudaTextureDesc::minMipmapLevelClamp

Lower end of the mipmap level range to clamp access to

6.23.2.6 enum cudaTextureFilterMode cudaTextureDesc::mipmapFilterMode

Mipmap filter mode

6.23.2.7 float cudaTextureDesc::mipmapLevelBias

Offset applied to the supplied mipmap level

6.23.2.8 int cudaTextureDesc::normalizedCoords

Indicates whether texture reads are normalized or not

6.23.2.9 enum cudaTextureReadMode cudaTextureDesc::readMode

Texture read mode

6.23.2.10 int cudaTextureDesc::sRGB

Perform sRGB->linear conversion during texture read

6.24 CUdevprop_st Struct Reference

Data Fields

- int clockRate
- int maxGridSize [3]
- int maxThreadsDim [3]
- int maxThreadsPerBlock
- int memPitch
- int regsPerBlock
- int sharedMemPerBlock
- int SIMDWidth
- int textureAlign
- int totalConstantMemory

6.24.1 Detailed Description

Legacy device properties

6.24.2 Field Documentation

6.24.2.1 int CUdevprop_st::clockRate

Clock frequency in kilohertz

6.24.2.2 int CUdevprop_st::maxGridSize[3]

Maximum size of each dimension of a grid

6.24.2.3 int CUdevprop_st::maxThreadsDim[3]

Maximum size of each dimension of a block

6.24.2.4 int CUdevprop_st::maxThreadsPerBlock

Maximum number of threads per block

6.24.2.5 int CUdevprop_st::memPitch

Maximum pitch in bytes allowed by memory copies

6.24.2.6 int CUdevprop_st::regsPerBlock

32-bit registers available per block

6.24.2.7 int CUdevprop_st::sharedMemPerBlock

Shared memory available per block in bytes

6.24.2.8 int CUdevprop_st::SIMDWidth

Warp size in threads

6.24.2.9 int CUdevprop_st::textureAlign

Alignment requirement for textures

6.24.2.10 int CUdevprop_st::totalConstantMemory

Constant memory available on device in bytes

6.25 CUipcEventHandle_st Struct Reference

6.25.1 Detailed Description

CUDA IPC event handle

6.26 CUipcMemHandle_st Struct Reference

6.26.1 Detailed Description

CUDA IPC mem handle

6.27 surfaceReference Struct Reference

Data Fields

- struct cudaChannelFormatDesc channelDesc

6.27.1 Detailed Description

CUDA Surface reference

6.27.2 Field Documentation

6.27.2.1 struct cudaChannelFormatDesc surfaceReference::channelDesc [read]

Channel descriptor for surface reference

6.28 textureReference Struct Reference

Data Fields

- enum cudaTextureAddressMode addressMode [3]
- struct cudaChannelFormatDesc channelDesc
- enum cudaTextureFilterMode filterMode
- unsigned int maxAnisotropy
- float maxMipmapLevelClamp
- float minMipmapLevelClamp
- enum cudaTextureFilterMode mipmapFilterMode
- float mipmapLevelBias
- int normalized
- int sRGB

6.28.1 Detailed Description

CUDA texture reference

6.28.2 Field Documentation

6.28.2.1 enum cudaTextureAddressMode textureReference::addressMode[3]

Texture address mode for up to 3 dimensions

6.28.2.2 struct cudaChannelFormatDesc textureReference::channelDesc [read]

Channel descriptor for the texture reference

6.28.2.3 enum cudaTextureFilterMode textureReference::filterMode

Texture filter mode

6.28.2.4 unsigned int textureReference::maxAnisotropy

Limit to the anisotropy ratio

6.28.2.5 float textureReference::maxMipmapLevelClamp

Upper end of the mipmap level range to clamp access to

6.28.2.6 float textureReference::minMipmapLevelClamp

Lower end of the mipmap level range to clamp access to

6.28.2.7 enum cudaTextureFilterMode textureReference::mipmapFilterMode

Mipmap filter mode

6.28.2.8 float textureReference::mipmapLevelBias

Offset applied to the supplied mipmap level

6.28.2.9 int textureReference::normalized

Indicates whether texture reads are normalized or not

6.28.2.10 int textureReference::sRGB

Perform sRGB->linear conversion during texture read

Index

Notice

Trademarks

Copyright