GPU Programming

A guide to using GPUs on the Cray XC40

GPU nodes on SERC system

- **Each XC GPU node has one Intel Xeon CPU and one Nvidia GPU**
  - CPU: Intel Ivybridge 12-core 2.4ghz
  - GPU: NVIDIA Tesla K40
    - 2880 cores, 12GB device memory
- **PBS Submission:**

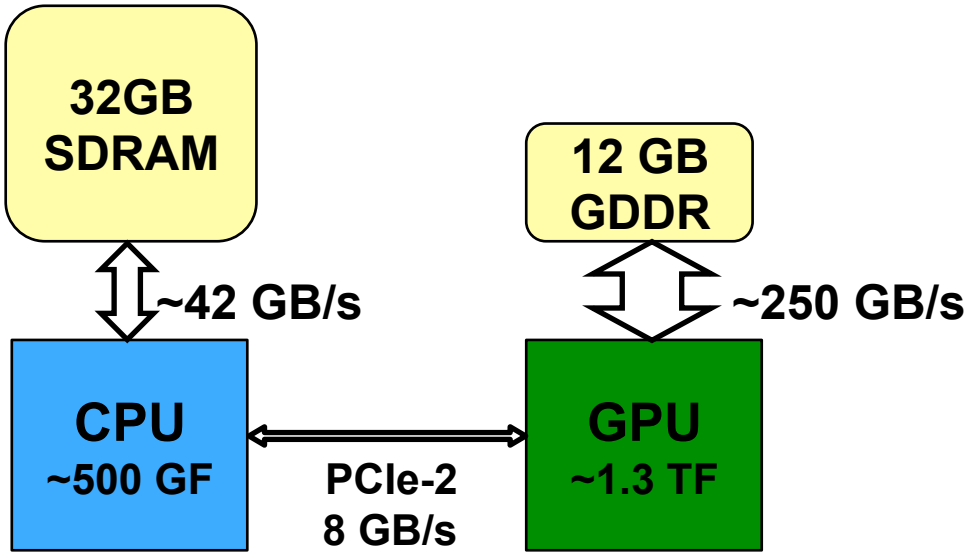
```
#PBS -l select=2:ncpus=1:accelerator=True:accelerator_model="Tesla_K40s"
#PBS -l accelerator_type="Tesla_K40s"
```

The New Generation of Supercomputers

- **Hybrid multicore has arrived and is here to stay**
  - Wide nodes are getting wider
  - Accelerators have leapt into the Top500

- **Programming accelerators efficiently is hard**
  - Three levels of parallelism required
    - MPI between nodes or sockets
    - Shared memory programming on the node
    - Vectorization for low level looping structures
  - Need a hybrid programming model to support these new systems
  - Need a high level programming environment
    - Compilers, tools, & libraries

Structural Issues with Accelerated Computing



- **Trick is to keep kernel data structures resident in GPU memory as much as possible**
  - **Avoid copying** between CPU and GPU
  - **Use asynchronous, non-blocking communication, multi-level overlapping**

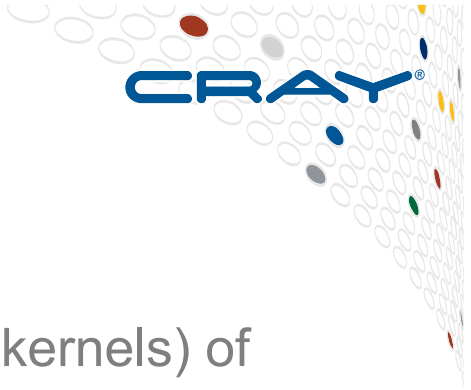
Program execution with a GPU

- **The main program runs on the host (CPU)**
  - Some of the code will also execute on the host
    - either serially or in parallel with threads (e.g., OpenMP)
  - This code could be:
    - calculations that you want to be done on the CPU, e.g.,
      - it is hard to parallelise for the GPU
      - there is not enough work to justify using the GPU
    - control statements for the GPU, e.g.,
      - memory management
      - synchronisation
    - communication calls, e.g. MPI
- **The main program can also**
  - launch kernels (tasks) on the device (GPU)
  - These are written specially for the GPU, e.g., with
    - CUDA
    - OpenACC

Cray Vision for Accelerated Computing

- NVIDIA CUDA: **If you work hard**, you can get good parallel performance
- **Most important hurdle for widespread adoption of accelerated computing in HPC is programming difficulty**
  - Need a single programming model that **is portable across machine types**
    - **Portable** expression of heterogeneity and multi-level parallelism
    - Programming model and optimization should not be significantly different for "accelerated" nodes and multi-core x86 processors
    - **Allow users to maintain a single code base**
- **Accelerated programming needs an ease of use tightly coupled high level programming environment with compilers, libraries, and tools that can hide the complexity of the system**
- **Ease of use is possible with**
  - Compiler making it **feasible for users** to write applications in **Fortran, C, and C++**
  - Tools to help users port and optimize for hybrid systems
  - Auto-tuned scientific libraries

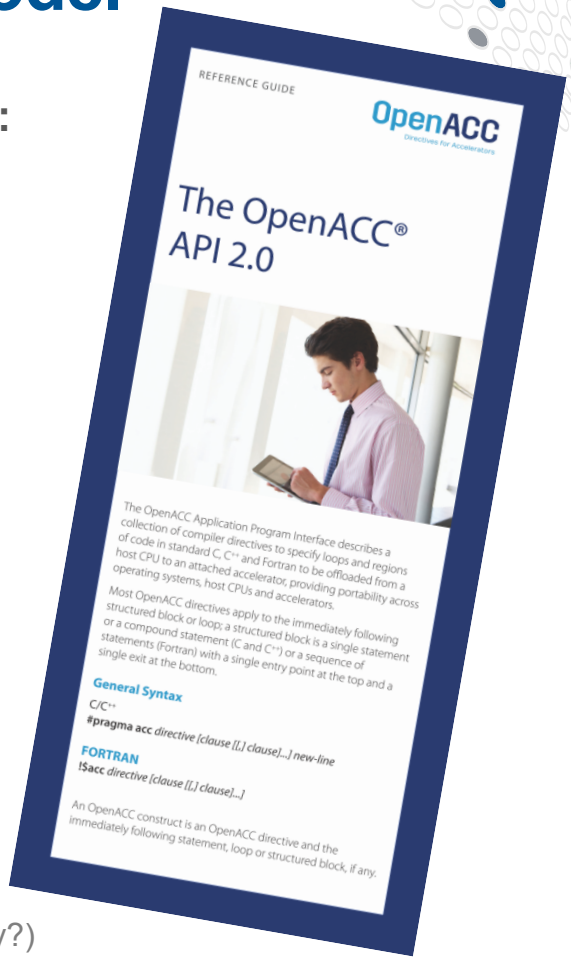
Potential programming approaches:



- **NVIDIA CUDA**
  - Offload-based programming model
  - Control code on host CPU launches parallel portions (kernels) of application on the attached GPU
- **OpenCL**
  - Open standard. Not officially supported by Cray but should work.
  - Programming model similar to CUDA
- **OpenACC**
  - High level programming model based on use of pragmas/directives (similar to OpenMP). Works with Fortran, C, C++
  - Main program executes on host, offload compute-intensive portions to GPU
- **Why directives?**
  - Address programming difficulty
  - Portability across platforms
  - Multi-language
  - Single code base
  - Multivendor support

OpenACC Accelerator Programming Model

- **Why a new model? There are already many ways to program:**
  - CUDA and OpenCL
    - All are quite low-level and closely coupled to the GPU
    - PGI CUDA Fortran: still CUDA just in a better base language
- **User needs to write specialized kernels:**
  - **Hard** to write and debug
  - **Hard** to optimize for specific GPU
  - **Hard** to update (porting/functionality)
- **OpenACC Directives provide high-level approach**
  - **Simple programming model for hybrid systems**
  - **Easier to maintain/port/extend code**
    - Non-executable statements (comments, pragmas)
    - The **same source** code can be compiled for multicore CPU
  - Based on Cray proposal to the OpenMP Accelerator Subcommittee
  - PGI accelerator directives, CAPS HMPP
    - First steps in the right direction – Needed standardization
  - **Possible performance sacrifice**
    - A small performance gap is acceptable (do you still hand-code in assembly?)
    - Goal is to provide at least 80% of the performance obtained with hand coded CUDA
- **Compiler support: all OpenACC Version 2.0 complete in 2014**
- **GCC support for OpenACC 2.0 planned for next major release**



Motivating Example: Reduction

- Sum elements of an array
- Original Fortran code
- **2.0 GFlops**

```
a=0.0

do i = 1,n
 a = a + b(i)
end do
```

The Reduction Code in Simple CUDA

```
__global__ void reduce0(int *g_idata, int *g_odata)
{
extern __shared__ int sdata[];

unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();

for(unsigned int s=1; s < blockDim.x; s *= 2) {
if ((tid % (2*s)) == 0) {
sdata[tid] += sdata[tid + s];
}
__syncthreads();
}

if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}


extern "C" void reduce0_cuda_(int *n, int *a, int *b)
{
int *b_d,red;
const int b_size = *n;


cudaMalloc((void **) &b_d , sizeof(int)*b_size);
cudaMemcpy(b_d, b, sizeof(int)*b_size,
cudaMemcpyHostToDevice);
```

```
dim3 dimBlock(128, 1, 1);
dim3 dimGrid(2048, 1, 1);
dim3 small_dimGrid(16, 1, 1);

int smemSize = 128 * sizeof(int);
int *buffer_d, *red_d;
int *small_buffer_d;

cudaMalloc((void **) &buffer_d , sizeof(int)*2048);
cudaMalloc((void **) &small_buffer_d ,
sizeof(int)*16);
cudaMalloc((void **) &red_d , sizeof(int));


reduce0<<< dimGrid, dimBlock, smemSize >>>(b_d,
buffer_d);

reduce0<<< small_dimGrid, dimBlock, smemSize
>>>(buffer_d, small_buffer_d);

reduce0<<< 1, 16, smemSize >>>(small_buffer_d,
red_d);

cudaMemcpy(&red, red_d, sizeof(int),
cudaMemcpyDeviceToHost);

*a = red;

cudaFree(buffer_d);
cudaFree(small_buffer_d);
cudaFree(b_d);
}
```

1.74 GFlops

The Reduction Code in Optimized CUDA

```
template<class T>
struct SharedMemory
{
    __device__ inline operator       T*()
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }

    __device__ inline operator const T*() const
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }
};

template <class T, unsigned int blockSize, bool nIsPow2>
__global__ void
reduce6(T *g_idata, T *g_odata, unsigned int n)
{
    T *sdata = SharedMemory<T>();

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;

    T mySum = 0;
    while (i < n)
    {
        mySum += g_idata[i];
        if (nIsPow2 || i + blockSize < n)
            mySum += g_idata[i+blockSize];
        i += gridSize;
    }
sdata[tid] = mySum;
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] = mySum = mySum
+ sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] = mySum = mySum
+ sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] = mySum = mySum
+ sdata[tid +  64]; } __syncthreads(); }
```

```
    if (tid < 32)
    {
        volatile T* smem = sdata;
        if (blockSize >=  64) { smem[tid] = mySum = mySum + smem[tid + 32];  }
        if (blockSize >=  32) { smem[tid] = mySum = mySum + smem[tid + 16];  }
        if (blockSize >=  16) { smem[tid] = mySum = mySum + smem[tid + 8];  }
        if (blockSize >=   8) { smem[tid] = mySum = mySum + smem[tid + 4];  }
        if (blockSize >=   4) { smem[tid] = mySum = mySum + smem[tid + 2];  }
        if (blockSize >=   2) { smem[tid] = mySum = mySum + smem[tid + 1];  }
    }

    if (tid == 0)
        g_odata[blockIdx.x] = sdata[0];
}
extern "C" void reduce6_cuda_(int *n, int *a, int *b)
{
  int *b_d;
  const int b_size = *n;

  cudaMalloc((void **) &b_d , sizeof(int)*b_size);
  cudaMemcpy(b_d, b, sizeof(int)*b_size, cudaMemcpyHostToDevice);

  dim3 dimBlock(128, 1, 1);
  dim3 dimGrid(128, 1, 1);
  dim3 small_dimGrid(1, 1, 1);
  int smemSize = 128 * sizeof(int);
  int *buffer_d;
  int small_buffer[4],*small_buffer_d;

  cudaMalloc((void **) &buffer_d , sizeof(int)*128);
  cudaMalloc((void **) &small_buffer_d , sizeof(int));
  reduce6<int,128,false><<< dimGrid, dimBlock, smemSize >>>(b_d,buffer_d, b_size);
  reduce6<int,128,false><<< small_dimGrid, dimBlock, smemSize
>>>(buffer_d, small_buffer_d,128);
  cudaMemcpy(small_buffer, small_buffer_d, sizeof(int),
cudaMemcpyDeviceToHost);

  *a = *small_buffer;

  cudaFree(buffer_d);
  cudaFree(small_buffer_d);
  cudaFree(b_d);
}
```

10.5 GFlops

The Reduction Code in OpenACC

- **Compiler does the work:**
  - Identifies parallel loops within the region
  - Determines the kernels needed
  - Splits the code into accelerator and host portions
  - Workshares loops running on accelerator
  - Data movement
    - Allocates/frees GPU memory at start/end of region
    - Moves data to/from GPU
- **8.32 GFlops**

```
!$acc data present(a,b)

a = 0.0

!$acc update device(a)

!$acc parallel

!$acc loop reduction(+:a)

do i = 1,n
  a = a + b(i)
end do

!$acc end parallel
!$acc end data
```

Reduction Code Summary

Summary of code complexity and performance

Programming Language / Model	Unit of computation	Lines of code	Performance in Gflops (higher is better)	Performance normalized to X86 core
Fortran	Single x86 core	4	2.0 Gflops	1.0
Simple CUDA	GPU	30	1.74 Gflops	0.87
Optimized CUDA	GPU	69	10.5 Gflops	5.25
OpenACC	GPU	9	8.32 Gflops	4.16

GPU Programming Environments

- **Not all compiler modules will work for all the different programming approaches. Generally, swap to the most up to date version in each case (should be default)**
  - e.g., "module avail cce" then "module swap cce cce/<whatever>"
- **For any GPU programming (CUDA, OpenCL, OpenACC...)**
  - make sure module craype-ivybridge is loaded since CPU is Ivybridge
  - make sure you always: "module load craype-accel-nvidia35"
    - it is not loaded by default
- **The craype-accel-nvidia35 module loads the following additional modules:**
  - cray-libsci_acc
  - cudatoolkit
- **Load craype-accel-nvidia* only if you are developing code that will be executed on GPU nodes.**
  - Loading the accelerator module enables dynamic linking by default and loads the libsci_acc module, which causes increased overhead if the resulting code is executed on non-GPU nodes.

Compiling CUDA

- **CUDA-C: programming model developed by NVIDIA**
  - Consists of both library calls and language extensions
  - Only NVIDIA's compiler nvcc understands the language extensions
  - Lots of tutorials and examples online
  - Requires explicitly rewriting important parts of code to
    - Manage gpu memory, copy data between host and gpu, execute on gpu
- **Compilation:**
  - module load craype-accel-nvidia35 craype-ivybridge
  - Main CPU code compiled with PrgEnv "cc" wrapper
    - either PrgEnv-gnu for gcc; or PrgEnv-cray for craycc
  - GPU CUDA-C kernels must be compiled with nvcc
    - nvcc -O3 -arch=sm_35
  - PrgEnv "cc" wrapper used for linking
    - Only GPU flag needed: `-lcudart`
      - i.e. no CUDA `-L` flags needed (added in cc wrapper)
- **nvcc does not know about MPI headers**
  - Simplest solution: isolate CUDA C and MPI codes into separate files
  - More complicated solution: explicitly include the MPI include directory in the nvcc compile
- **Building a .cu file enables C++ name mangling**
  - C codes will need to be built with the CC compiler or...
  - Add extern "C" to continue using cc compiler

Compiling OpenCL

- **OpenCL – set of libraries and C language extensions maintained by Kronos group and supported by multiple vendors**
  - Functionally similar to low-level CUDA driver API
  - Requires explicitly rewriting important parts of code as with CUDA
- **Compilation:**
  - module load craype-accel-nvidia35
  - Main CPU code compiled with PrgEnv "cc" wrapper
    - either PrgEnv-gnu for gcc; or PrgEnv-cray for craycc
  - GPU OpenCL kernels compiled with nvcc
  - PrgEnv "cc" wrapper used for linking
    - Only GPU flag needed: `-lOpenCL`
- **Alternatively:**
  - Use PrgEnv-gnu for all compilation
    - still need `-lOpenCL` at linktime

Compiling OpenACC

- **OpenACC: a collection of compiler directives**
  - Specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator,
  - provides portability across operating systems, host CPUs and accelerators.
  - Supported compilers: Cray and PGI
- **Compilation (Cray):**
  - module load craype-accel-nvidia35
  - module load craype-ivybridge
  - module load PrgEnv-cray
  - Compile with cc or ftn
  - Helpful flags: -hacc –rm (Fortran) or –h pragma=acc –hmsgs (C)
  - See manpage for –hacc_model options
- **Compilation (PGI):**
  - module unload cray-libsci_acc (not supported for PGI compiler)
  - Use -acc -ta=nvidia flags
  - Compiler feedback is very useful: -Minfo=accel
  - Runtime feedback is also useful: export ACC_NOTIFY=1

module load cray-libsci_acc

- **Provides GPU-accelerated scientific libraries**
  - e.g. BLAS
- **Libraries can be used:**
  - For GPU-resident data (without data copies)
  - For CPU-resident data
    - pure accelerated: all data copied to/from GPU
    - hybrid (some data processed on CPU, some on GPU [with copying])
  - Optional smart interface will select best method at runtime
- **For more information: man intro libsci_acc**

- **Also: `man intro_openacc, man openacc.examples`**

Usage Tips

- **Sharing the GPU in an XC node**
  - Nvidia GPUs default to dedicated mode where each GPU is mapped to one and only one process or one MPI rank per compute node.
  - Override default by setting CRAY_CUDA_MPS=1 (aka “proxy mode”)
  - If use this, make sure APRUN_XFER_LIMITS is disabled (unset)
    - NVIDIA driver will multiplex CUDA kernels from different processes
    - Might allow for more efficient loading and utilization of GPU
    - Keep in mind memory limitations of hardware
  - In proxy mode, more likely to see errors like CUDA_ERROR_OUT_OF_MEMORY
  - For debugging set CRAY_CUDA_MPS=0
  - Performance-related environment variables:
    - MPICH_RDMA_ENABLED_CUDA
    - MPICH_G2G_PIPELINE

OpenACC Debugging

- **The intro_openacc man page contains a huge amount of detailed information in its discussion section – do read!**

- **CRAY_ACC_DEBUG**
  - When the runtime environment variable CRAY_ACC_DEBUG is set to 1, 2,or 3, CCE writes runtime commentary of accelerator activity to STDERR for debugging purposes
  - Every accelerator action on every PE generates output prefixed with "ACC:". This may produce a large volume of output and it may be difficult to associate messages with certain routines and/or certain PEs.
  - There is also a set of API calls that programmer can use to enable or disable output at certain points of the code

- **Can also use Cray Performance Tools (CrayPat)**
  - More information in later session

Example of CRAY_ACC_DEBUG output

```
ACC: Initialize CUDA
ACC: Get Device 0
ACC: Create Context
ACC: Set Thread Context
ACC: Start transfer 2 items from saxpy.c:17
ACC: allocate, copy to acc 'x' (4194304 bytes)
ACC: allocate, copy to acc 'y' (4194304 bytes)
ACC: End transfer (to acc 8388608 bytes, to host 0 bytes)
ACC: Execute kernel saxpy$ck_L17_1 blocks:8192 threads:128
async(auto) from saxpy.c:17
ACC: Wait async(auto) from saxpy.c:18
ACC: Start transfer 2 items from saxpy.c:18
ACC: free 'x' (4194304 bytes)
ACC: copy to host, free 'y' (4194304 bytes)
ACC: End transfer (to acc 0 bytes, to host 4194304 bytes)
```

NVIDIA Debugging Tools

- **Command-line profiler via environment variables (also works for OpenACC)**
  - MPI or serial
  - Set COMPUTE_PROFILE=1 before running to enable
  - Writes log file showing kernel activity etc.
  - export COMPUTE_PROFILE_LOG=”myfilename” to change log name
  - For multiple processes, add'%p' in the COMPUTE_PROFILE_LOG name. This will generate separate profiler output files for each process - with '%p' substituted by the process id.
    - e.g. export COMPUTE_PROFILE_LOG="cuda_log_%p"
  - COMPUTE_PROFILE_CSV: set to 1 to enable a comma separated version of the log output.
  - COMPUTE_PROFILE_CONFIG: used to specify a config file for selecting profiling options and performance counters.

- **nvprof**
  - Serial code tuning
  - export COMPUTE_PROFILE=0
  - aprun -n 1 nvprof laplace2d_accpgi [I have not tested this!!]

Example of COMPUTE_PROFILE=1 output

export COMPUTE_PROFILE=1
> aprun -n 1 ./laplace2d_acc

```
main()
Jacobi relaxation Calculation: 4096 x 4096 mesh
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000403
  700, 0.000345
  800, 0.000302
  900, 0.000269
 total: 3.945941 s
Application 140290 resources: utime ~4s, stime ~1s
```

> ls –l

```
-rw------- 1 arnoldg bw_staff  236416 Mar 12 13:39 cuda_profile_0.log
```

> more cuda_profile_0.log

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla K20X
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff69047ada518
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 53270.656 ] cputime=[ 53558.000 ]
method=[ memcpyHtoD ] gputime=[ 1.600 ] cputime=[ 37.000 ]
method=[ laplace$ck_L64_3 ] gputime=[1899.712 ] cputime=[ 26.0 ] occupancy=[ 0.75 ]
method=[ memcpyDtoH ] gputime=[ 3.104 ] cputime=[ 49.000 ]
method=[ laplace$ck_L75_5 ] gputime=[1757.760 ] cputime=[ 10.0 ] occupancy=[ 1.00 ]
method=[ laplace$ck_L64_3 ] gputime=[1905.536 ] cputime=[ 8.0 ] occupancy=[ 0.75 ]
...
```

Example of nvprof output

export COMPUTE_PROFILE=0 # or unset

> export LD_LIBRARY_PATH=$CRAY_CUDATOOLKIT_DIR/lib64:$LD_LIBRARY_PATH
> cd $PBS_O_WORKDIR
> aprun -n 1 nvprof laplace2d_accpgi

```
======== NVPROF is profiling laplace2d_accpgi...
======== Command: laplace2d_accpgi
main()
Jacobi relaxation Calculation: 4096 x 4096 mesh
    0, 0.250000
  100, 0.002397
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
  600, 0.000403
  700, 0.000345
  800, 0.000302
  900, 0.000269
 total: 6.712810 s
======== Warning: Application returned non-zero code 19
======== Profiling result:
 Time(%)      Time   Calls       Avg       Min       Max  Name
   65.24      3.48s    1000    3.48ms    3.47ms    3.49ms  laplace_66_gpu
   31.11      1.66s    1000    1.66ms    1.66ms    1.66ms  laplace_77_gpu
    2.41  128.73ms    1000  128.73us  127.68us  130.33us  laplace_70_gpu_red
    0.72   38.63ms    1001   38.59us    2.53us   36.03ms  [CUDA memcpy DtoH]
    0.51   27.25ms    1128   24.16us    3.74us  182.66us  [CUDA memcpy HtoD]
Application 83077 resources: utime ~5s, stime ~3s
```