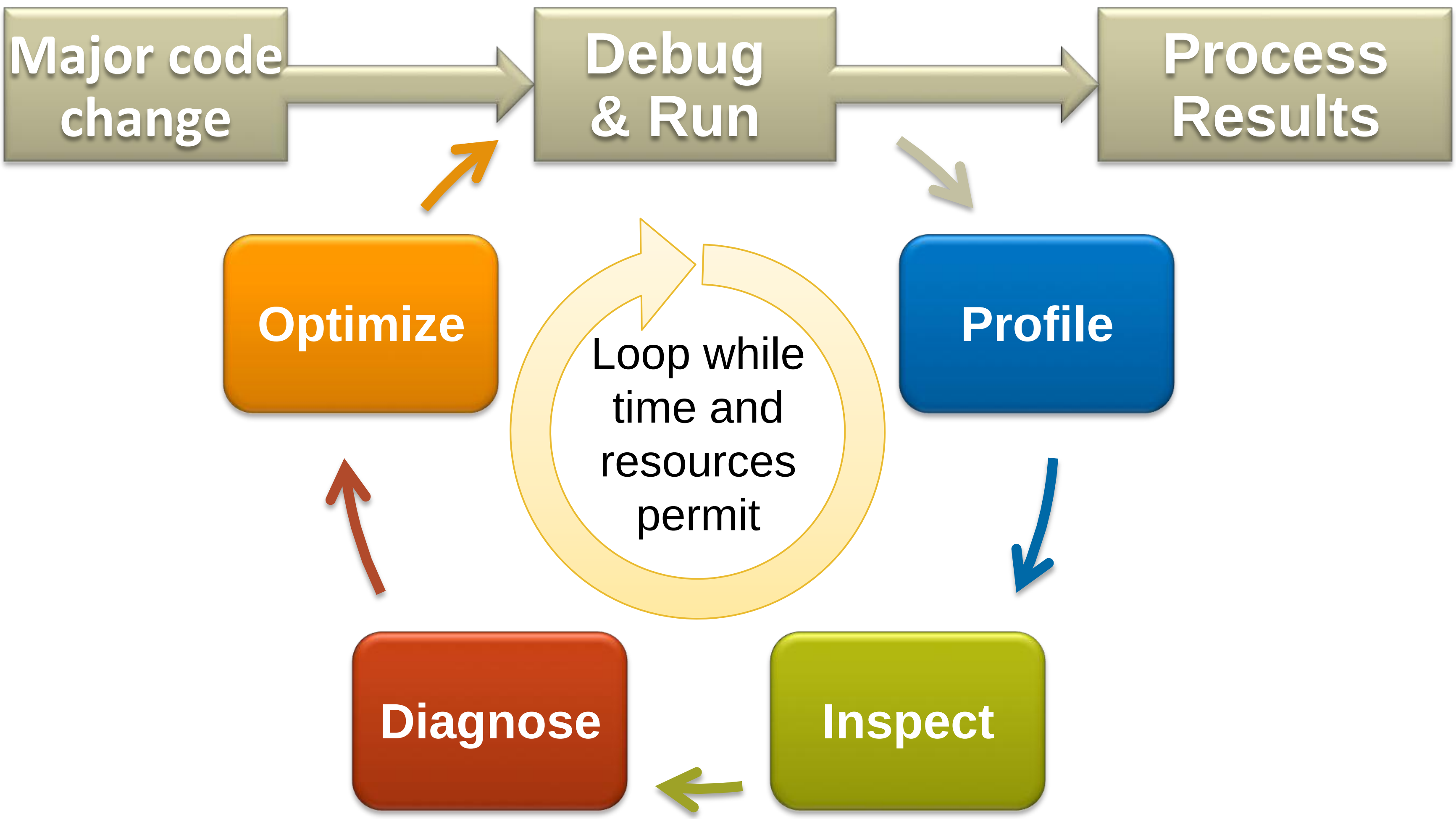
Performance Analysis with CrayPat

Part 1

Outline

- **Introduction to performance analysis with CrayPat**
  - Different approaches to profiling: Sampling vs. Tracing
  - How to recompile and run your code for CrayPat.
  - Combining Sampling and Tracing: Automatic Performance Analysis
- **Collecting Hardware Performance counters.**

The Optimization Cycle



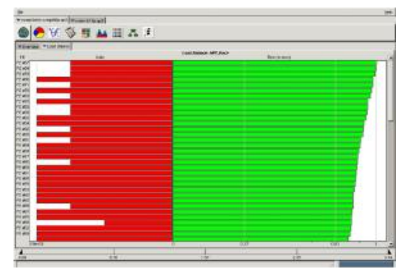
CrayPAT Overview

- **Assist the user with application performance analysis and optimization**
  - Provides concrete suggestions instead of just reporting data.
  - Work on user codes at realistic core counts with thousands of processes/threads
  - Integrate into large codes with millions of lines of code
- **Is a universal tool**
  - Basic functionality available to all compilers on the system
  - Additional functionality available for the Cray compiler (loop profiling)
- **Requires no source code or Makefile modification**
  - Automatic instrumentation at group (function) level such as mpi, io, …
  - Requires object files and archives for instrumentation and to be compiled with the wrapper scripts while the `perftools` module was loaded.
  - Able to generate instrumentation on optimized code.
  - Creates a new stand-alone instrumented program while preserving the original binary.
- **Is under continuous development – always improving!**

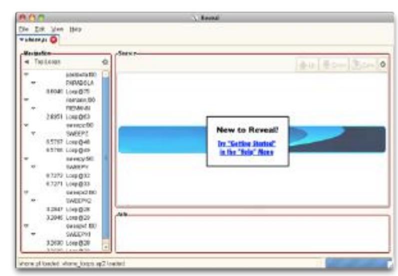
Components of CrayPat

- **Available through the `perftools` module:**
  - **pat_build** - Instruments the program to be analyzed (command line)
  - **pat_report** - Generates text reports from the performance data captured during program execution and exports data for use in other programs. (command line)

  - **Cray Apprentice2** - A graphical analysis tool that can be used to visualize and explore the performance data captured during program execution.



  - **Reveal** - A graphical source code analysis tool that can be used to correlate performance analysis data with annotated source code listings, to identify key opportunities for optimization.



  - **craypat-lite –** Light weight profiling tool.

Components of CrayPat (cont.)

- **grid_order** - Generates MPI rank order information that can be used with the MPICH_RANK_REORDER environment variable to override the default MPI rank placement scheme and specify a custom rank placement. (For more information, see the **intro_mpi(3)** man page.)

- **pat_help** - Help system, which contains extensive usage information and examples. This help system can be accessed by entering pat_help at the command line.

- The individual components of CrayPat are documented in the following man pages (info on hardware counters will follow):
  - **intro_craypat(1)**
  - **pat_build(1)**
  - **pat_report(1)**
  - **pat_help(1)**
  - **grid_order(1)**
  - **app2(1)**
  - **reveal(1)**

Sampling and Event Tracing

- **CrayPAT provides two fundamental ways of profiling:**

1. **Sampling**
   - By taking regular snapshots of the applications call stack we can create a statistical profile of where the application spends most time.
   - Snapshots can be taken at regular intervals in time or when some other external event occurs, like a hardware counter overflowing
2. **Event Tracing**
   - Alternatively we can record performance information every time a specific program event occurs, e.g. entering or exiting a function.
   - We can get accurate information about specific areas of the code every time the event occurs
   - Event tracing code can be added automatically or included manually through API calls.

- **Automatic Performance Analysis (APA) combines the two approaches.**
- **Loop profiling is a special flavor of event tracing.**

Sampling

Advantages
- Only need to instrument main routine
- Low Overhead – depends only on sampling frequency
- Smaller volumes of data produced

Disadvantages
- Only statistical averages available
- Limited information from performance counters

Event Tracing

Advantages
- More accurate and more detailed information
- Data collected from every traced function call not statistical averages

Disadvantages
- Increased overheads as number of function calls increases
- Huge volumes of data generated

The best approach is *guided tracing*.
e.g., Only tracing functions that are not small (i.e., very few lines of code) and contribute a lot to application's run time.
APA is an automated way to do this.

CrayPat

Full featured application profiling

Exercise 1: Generate a Sampling Profile

```
> module load perftools
```

- Makes the default version of CrayPAT available
- Subsequent compiler invocations will automatically insert necessary hooks for profiling (not always up-to-date with latest third-party compilers)
- Binaries are *not* automatically instrumented

```
> make clean; make
> pat_build -S himeno.exe
```

- Builds code with profiling hooks, then instruments the binary
- Result named himeno.exe+pat

```
> aprun -n 24 ./himeno.exe+pat  (within PBS script)
> pat_report -o myreport.txt himeno+pat+*  (when PBS job returns)
```

- Running the "+pat" binary creates a data file "*.xf" or a directory in run directory
- pat_report reads that data file and prints lots of human-readable performance data. Creates an *.ap2 file.

Table 2: Profile by Group, Function, and Line

```
 Samp% |    Samp | Imb. |   Imb. |Group
       |         | Samp | Samp%  | Function
       |         |      |        |  Source
       |         |      |        |   Line
       |         |      |        |    PE=HIDE

 100.0% | 2063.0 |   -- |     -- |Total
|--------------------------------------------------------------------------------
|  82.3% | 1698.0 |   -- |     -- |USER
||-------------------------------------------------------------------------------
||  77.2% | 1592.2 |   -- |     -- |jacobi
3|        |        |      |        | Himeno/test.samp/himeno.c
||||-----------------------------------------------------------------------------
4|||  61.1% | 1260.6 | 32.4 |   2.9% |line.243
4|||   7.2% |  147.8 | 19.2 |  13.2% |line.257
4|||   4.3% |   89.5 | 17.5 |  18.7% |line.258
4|||   4.2% |   86.5 |  8.5 |  10.2% |line.260
||||=============================================================================
||   5.1% |  105.8 |   -- |     -- |initmt
3|        |        |      |        | Himeno/test.samp/himeno.c
||===============================================================================
|  16.4% |  338.2 |   -- |     -- |ETC
||-------------------------------------------------------------------------------
||  13.8% |  284.8 |  5.2 |   2.1% |__cray_scopy_HSW
||   2.6% |   53.5 |  4.5 |   8.9% |__cray_sset_HSW
||===============================================================================
|   1.3% |   26.6 |   -- |     -- |MPI
|================================================================================
```

Top function

Communication not relevant. Threshold of 0.5% can be cancelled with –T option.

Exercise 2: Generate a Tracing Profile

```
> module load perftools
```

- Makes the default version of CrayPAT available.

```
> pat_build -u -g mpi himeno.exe
```

- If your application is already built with `perftools` loaded you do not have to rebuild when switching the experiment.
- Traces MPI functions calls and functions defined in the program source files

```
> aprun -n 24 ./himeno.exe+pat  (from within PBS script)
> pat_report -o myrep.txt himeno+pat+*
```

- Running the “+pat” binary creates a data file or directory
- pat_report reads that data file and prints lots of human-readable performance data. Creates an *.ap2 file.

Table 1: Profile by Function Group and Function

Time%	Time	Imb. Time	Imb. Time%	Calls	Group / Function / PE=HIDE
100.0%	20.643909	--	--	1149.0	Total
98.8%	20.395989	--	--	219.0	USER
91.1%	18.797060	0.115535	0.7%	2.0	jacobi
7.7%	1.597866	0.006647	0.5%	1.0	initmt
0.0%	0.000402	0.000167	33.5%	53.0	sendp3
1.2%	0.239306	--	--	871.0	MPI
0.7%	0.148981	0.094595	44.4%	159.0	MPI_Waitall
0.4%	0.085824	0.023669	24.7%	318.0	MPI_Isend
0.0%	0.004125	0.004316	58.4%	318.0	MPI_Irecv
0.0%	0.000298	0.000013	4.8%	55.0	MPI_Allreduce
0.0%	0.000033	0.000013	32.8%	1.0	MPI_Cart_create
0.0%	0.008614	--	--	59.0	MPI_SYNC
0.0%	0.006696	0.006627	99.0%	2.0	MPI_Barrier(sync)
0.0%	0.001802	0.001399	77.6%	55.0	MPI_Allreduce(sync)
0.0%	0.000061	0.000052	86.3%	1.0	MPI_Init(sync)
0.0%	0.000056	0.000051	91.7%	1.0	MPI_Finalize(sync)

User functions

Communication

Synchronisation

Options for Tracing

- **More information is given in the `pat_build` man page**
  - **-u** Create new trace intercept routines for those functions that are defined in the respective source file owned by the user.
  - **-w** Make tracing the default experiment and create new trace intercept routines for those functions for which no trace intercept routine already exists. If -t, -T, or the trace build directive are not specified, only those functions necessary to support the CrayPat runtime library are traced.
  - **-T tracefunc** Instrument program to trace the function references to tracefunc. This option applies to all user-defined entry points as well as to those that appear in the predefined function groups listed under the -g option. Use the nm or readelf command to determine function names to specify for tracing. If tracefunc begins with an exclamation point (!) character, references to tracefunc are not traced.
  - **-t tracefile** Instrument program to trace all function references listed in tracefile.
- **Only true function calls can be traced. Functions that are inlined by the compiler or that have local scope in a compilation unit cannot be traced.**

Options for Tracing

- More information is given in the `pat_build` man page
  - **-g tracegroup** Instrument the program to trace all function references belonging to the trace function group tracegroup. Only those functions actually executed by the program at runtime are traced. A selection of tracegroup values is:
    - **blas** Basic Linear Algebra subprograms
    - **netcdf** Network Common Data Form
    - **hdf5** HDF5 I/O library
    - **heap** dynamic heap
    - **io** includes stdio and sysio groups
    - **lapack** Linear Algebra Package
    - **mpi** MPI
    - **omp** OpenMP API
    - **sysio** I/O system calls
    - **syscall** system calls
- More information on the various tracegroup values is given in `$CRAYPAT_ROOT/share/traces` after loading the `perftools` module.

Files generated during regular Profiling

- **a.out+pat+PID-node[s|t].xf: raw data files**
  - Depending on the nature of the program and the environmental conditions in effect at the time of program execution, when executed, the instrumented executable generates one or more data files with the suffix .xf, where:
    - **a.out** is the name of the original program.
    - **PID** is the process ID assigned to the instrumented program at runtime.
    - **node** is the physical node ID upon which the rank zero process executed.
    - **s|t** is a one-letter code indicating the type of experiment performed, either **s** for sampling or **t** for tracing.
  - Use the `pat_report` command to view or dump the .xf file or export it to another file format for use with other applications, i.e. *.ap2 files.

- ***.ap2 files: self contained compressed performance files.**
  - Normally about 5 times smaller than the corresponding set of *.xf files.
  - Only one *.ap2 per experiment compared to potentially multiple *.xf files.
  - Contains the information needed from the application binary and can be reused, even if the application binary is no longer available or if it was rebuilt.
  - Is independent on the version used to generate the ap2 file while the xf files are very version dependent.
  - It is the only input format accepted by Cray Apprentice2 and Reveal.
  - => Can delete the .xf files after you have the ap2 file.

Using pat_report

- **Always need to run pat_report at least once to perform data conversion**
  - Combines information from xf output (optimized for writing to disk) and binary with raw performance data to produce ap2 file (optimized for visualization analysis and smaller than raw data)
  - Instrumented binary must still exist when data is converted!
  - Resulting ap2 file is the input for subsequent pat_report calls and Reveal or Apprentice[2]
  - xf files and instrumented binary files can be removed once ap2 file is generated.

- **Generates a text report of performance results**
  - Data laid out in tables
  - Many options for sorting, slicing or dicing data in the tables.

```
> pat_report -O <table option> *.ap2
> pat_report -O help (list of available profiles)
```

  - Volume and type of information depends upon sampling vs tracing.

Some useful predefined report types:

- **pat_report –O ca+src**
  - Show the callers (bottom-up view) leading to the routines that have a high use in the report and include source code line numbers for the calls and time-consuming statements.
- **pat_report –O load_balance**
  - Show load-balance statistics for the high-use routines in the program. Parallel processes with minimum, maximum and median times for routines will be displayed. Only available with tracing experiments.
- **pat_report –O mpi_callers**
  - Show MPI message statistics. Only available with tracing experiments.

Using pat_report

- The performance numbers reported are in general an average over all tasks (also explains non-integer values)

- Not always meaningful
  - Master-slave schemes
  - MPMD

```
Time% |      Time |     Imb. |  Imb. |  Calls |Group
      |           |     Time | Time% |        | Function
      |           |          |       |        |  PE=HIDE

100.0% | 20.643909 |       -- |    -- | 1149.0 |Total
|-----------------------------------------------------------
|  98.8% | 20.395989 |       -- |    -- |  219.0 |USER
||----------------------------------------------------------
||  91.1% | 18.797060 | 0.115535 |  0.7% |    2.0 |jacobi
||   7.7% |  1.597866 | 0.006647 |  0.5% |    1.0 |initmt
||   0.0% |  0.000402 | 0.000167 | 33.5% |   53.0 |sendp3
```

- To solve this you can filter the *.ap2 file

  `> pat_report -sfilter_input=‘condition’ …`

  - The ‘condition’ should be an expression involving 'pe' such as 'pe<1024' or 'pe%2==0'.
  - This option is also useful when the size of the full data file makes a report incorporating data from all PEs take too long or exceed the available memory

Combining Sampling and Tracing: APA

- **Motivation for Automatic Profiling Analysis:**
  - For programs that run for only a few seconds, there is no problem with using `pat_build` with the `-u` and `-g` mpi options to trace all user functions.
  - However with a large, long-running program such a trace will inject considerable overhead. It is better to limit tracing to those functions that consume the most time.
  - One can use a preliminary sampling experiment to determine and instrument those functions, referred to as automatic profiling analysis.

  - APA provides a simple procedure to instrument and collect performance data as a first step for novice and expert users.
  - Identifies top time consuming routines through sampling and provides instructions to trace only those routines.
  - Automatically creates instrumentation template customized to application for future in-depth measurement and analysis

Automatic Profiling Analysis (1/2)

```
> module load perftools
```

- Makes the default version of CrayPAT available.

```
> make clean; make
> pat_build himeno.exe
```

- The APA is the default experiment. No option needed.
- The pat_build generates a binary instrumented for sampling (different from the pure sampling shown before.)

```
> aprun –n 24 ./himeno.exe+pat” (from PBS job)
> pat_report –o myrep.txt himeno+pat+*
```

- Running the “+pat” binary creates a data file or directory.
- Applying pat_report to the *.xf generates an ***.apa** file in addition to the *.ap2 file.

Automatic Profiling Analysis (2/2)

```
> vi *.apa
```

- The *.apa file contains instructions for the next step, i.e. tracing. Modify it according to your needs.

```
> pat_build -O *.apa
```

- Generates an instrumented binary `himeno.exe+apa` for tracing according to the instructions in the *.apa file.

```
> aprun -n 24 ./himeno.exe+apa
> pat_report -o myrep.txt himeno+apa+*
```

- Running the "+apa" binary creates a new data file or directory.
- Applying pat_report to the *.xf generates a new*.ap2 file.

*.apa File after Sampling Experiment

```
# ----------------------------------------------------------------------
#       Collect the default PERFCTR group.
  -Drtenv=PAT_RT_PERFCTR=default
...
# ----------------------------------------------------------------------

#       Libraries to trace.
  -g mpi
# ----------------------------------------------------------------------

#       User-defined functions to trace, sorted by % of samples.

  -w  # Enable tracing of user-defined functions.
      # Note: -u should NOT be specified as an additional option.

# 77.44% 3751 bytes
         -T jacobi

# 5.04% 2467 bytes
         -T initmt
# ----------------------------------------------------------------------

-o himeno.exe+apa # New instrumented program.
```

Suggestion to collect Performance counters

Augment this list if needed, i.e. `-g mpi,io`

Add or remove functions as needed.

Create the binary for tracing

A Sequence of Commands

```
rns/crayPatExample> module load perftools     # Loaded the CrayPat module
rns/crayPatExample> ftn -o samp264 samp264.f  # compiled the code – simple application
rns/crayPatExample> pat_build samp264         # Created the experiment executable samp64+pat
rns/crayPatExample> vi samp264.pbs            # modify the job script to run samp64+pat
rns/crayPatExample> qsub samp264.pbs          # run the job
rns/crayPatExample> cat samp264.pbs.o1879623  # Made sure the job ran ☺
rns/crayPatExample> pat_report samp264+pat+15346-43sdt.xf > samp264+pat+15346.report
rns/crayPatExample> view samp264+pat+15346.report
rns/crayPatExample> pat_build -O samp264+pat+15346-43sdt.apa
rns/crayPatExample> ls -ltr
-rw-r--r-- 1 rns hwpt    5411 Sep 25 13:34 samp264.f
-rw-r--r-- 1 rns hwpt     306 Sep 25 13:34 samp264.pbs
-rwxr-xr-x 1 rns hwpt 2001625 Sep 25 13:35 samp264
-rwxr-xr-x 1 rns hwpt 3592502 Sep 25 13:35 samp264+pat
-rw------- 1 rns hwpt     459 Sep 25 13:36 samp264.pbs.o1879623
-rw-r----- 1 rns hwpt    7240 Sep 25 13:36 samp264+pat+15346-43sdt.xf
-rw-r--r-- 1 rns hwpt    5248 Sep 25 13:37 samp264+pat+15346.report
-rw-r--r-- 1 rns hwpt    1613 Sep 25 13:37 samp264+pat+15346-43sdt.apa
-rw-r--r-- 1 rns hwpt   36864 Sep 25 13:37 samp264+pat+15346-43sdt.ap2
-rwxr-xr-x 1 rns hwpt 3599971 Sep 25 13:53 samp264+apa

rns/crayPatExample> vi samp264.pbs            # modify the job script to run samp64+apa
rns/crayPatExample> qsub samp264.pbs          # run the job
rns/crayPatExample> pat_report samp264+apa+8557-142tdt.xf > samp264+apa+8557.report
rns/crayPatExample> view samp264+apa+8557.report
```

General Remarks

- **Always check that the instrumenting binary has not affected the run time notably compared to the original**
- **Collecting event traces on large numbers of frequently called functions, or setting the sampling interval very low can introduce a lot of overhead (check `trace-text-size` option to `pat_build`)**
- **MUST run on Lustre !**
- **The runtime analysis can be modified through the use of environment variables of the form `PAT_RT_*`**
  - Number of files used to store raw data:
    - 1 file created for program with 1 – 256 processes
    - √n files created for program with 257 – n processes
    - Ability to customize with PAT_RT_EXPFILE_MAX
  - Check the `PAT_LD_OBJECT_TMPDIR` variable if you cannot preserve the original build tree.

Hardware Performance Counters

- **CrayPat supports the use of hardware counters to collect hardware events**
  - Most counters accessed through the PAPI interface.
  - Predefined sets of hardware counters are specified that can be instrumented for performance analysis experiment.
  - Number of simultaneous counters limited by hardware.
- **CrayPat provides information at the function call level on hardware features like caches, vectorization and memory bandwidth. Very useful feature for understanding application performance bottlenecks.**
- **HWPC collection can slow down the execution notably.**
  - Should be used within a tracing experiment only for a small set of functions or ideally through an automatic performance analysis.

Hardware Counters Selection

- **HW counter collection enabled with `PAT_RT_PERFCTR` environment variable (not set by default)**

```
export PAT_RT_PERFCTR=<event list> | <group>
```

- Counter events are specified in a comma-separated list. Event names and groups from any and all components may be mixed as needed. To list the names of the individual events on your system, use the `papi_avail` and `papi_native_avail` commands which are explained in the `papi_counters` man page.
- Alternatively, counter group numbers can be used in addition to or in place of individual event names, to specify one or more predefined performance counter groups. The groups are given in the hwpc man page (contents in $CRAYPAT_ROOT/share/counters/)
- An overview of events is given in `pat_help->counters->haswell`
- Aries network performance counters is found in the `nwpc(5)` man page.
- Intel Running Average Power Limit and Cray Power Management in `rapl(5)`, and info on Performance API (PAPI) in `intro_papi(5)`.

Haswell HW counter groups (hwpc man page)

Table 5. Intel Haswell Event Sets

Group	Description
0	D1 with instruction counts
1	Summary with cache and TLB metrics (default)
2	D1, D2, and L3 metrics
3-5	Not used
6	Micro-op queue stalls
7	Back-end stalls
8	Instructions and branches
9	Instruction cache
10	Cache hierarchy
19	Prefetches
23	Summary with cache and TLB metrics (same as 1)

Intel Haswell Processor: Hardware performance counters do not support floating-point operations.

Most useful for measuring cache efficiency. List of events is given in $CRAYPAT_ROOT/share/counters

Example: HW counter data and derived metrics

```
==============================================================================
  USER / jacobi
------------------------------------------------------------------------------
  Time%                                                      91.0%
  Time                                                   18.783816 secs
  Imb. Time                                               0.131366 secs
  Imb. Time%                                                  0.8%
  Calls                                 0.106 /sec             2.0 calls
  CPU_CLK_THREAD_UNHALTED:REF_XCLK                      1874027894
  CPU_CLK_THREAD_UNHALTED:THREAD_P                     52330735798
  DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK                     15309079
  DTLB_STORE_MISSES:MISS_CAUSES_A_WALK                     9590363
  L1D:REPLACEMENT                                       2490612461
  L2_RQSTS:ALL_DEMAND_DATA_RD                           1255673984
  L2_RQSTS:DEMAND_DATA_RD_HIT                            495319777
  MEM_UOPS_RETIRED:ALL_LOADS                            7905309689
  User time (approx)                   18.783 secs     46977527366 cycles  100.0% Time
  CPU_CLK                               2.792GHz
  TLB utilization                      317.49 refs/miss      0.620 avg uses
  D1 cache hit,miss ratios              68.5% hits           31.5% misses
  D1 cache utilization (misses)          3.17 refs/miss      0.397 avg hits
  D2 cache hit,miss ratio               69.5% hits           30.5% misses
  D1+D2 cache hit,miss ratio            90.4% hits            9.6% misses
  D1+D2 cache utilization               10.40 refs/miss      1.300 avg hits
  D2 to D1 bandwidth                 4080.191MiB/sec   80363134952 bytes
  Average Time per Call                                   9.391908 secs
  CrayPat Overhead : Time                0.0%
```

Raw counters

derived

Example: Observations and suggestions

```
D1 + D2 cache utilization:

    7.7% of total execution time was spent in 1 functions with combined
    D1 and D2 cache hit ratios below the desirable minimum of 80.0%.
    Cache utilization might be improved by modifying the alignment or
    stride of references to data arrays in these functions.

     D1+D2 cache hit ratio  Time%  Function

      58.9%                  7.7%  initmt

TLB utilization:

    7.7% of total execution time was spent in 1 functions with fewer
    than the desirable minimum of 200 data references per TLB miss. TLB
    utilization might be improved by modifying the alignment or stride
    of references to data arrays in these functions.

       LS per TLB DM  Time%  Function

       5.21            7.7%  initmt
```

Compiler Feedback (CCE)

- With CCE use `-rm` for Fortran or `-hlist=a` for C/C++
- For each source file a corresponding `*.lst` file is created.

```
%%%    L o o p m a r k   L e g e n d    %%%

     Primary Loop Type        Modifiers
     ------- ---- ----        ---------
     A - Pattern matched      a - atomic memory operation
                              b - blocked
     C - Collapsed            c - conditional and/or
computed
     D - Deleted
     E - Cloned
     F - Flat - No calls      f - fused
     G - Accelerated          g - partitioned
     I - Inlined              i - interchanged
     M - Multithreaded        m - partitioned
                              n - non-blocking remote
transf.
                              p - partial
                              r - unrolled
                              s - shortloop
     V - Vectorized           w - unwound
```

```
  191.    C-------------<    for(i=0 ; i<MIMAX ; ++i)
  192.    C C-----------<      for(j=0 ; j<MJMAX ; ++j)
  193.    C C VCr2------<        for(k=0 ; k<MKMAX ; +
+k){
  194.    C C VCr2                 a[0][i][j][k]=0.0;
  195.    C C VCr2                 a[1][i][j][k]=0.0;
  196.    C C VCr2                 a[2][i][j][k]=0.0;
  197.    C C VCr2                 a[3][i][j][k]=0.0;
202.    C C VCr2                 c[1][i][j][k]=0.0;
  203.    C C VCr2                 c[2][i][j][k]=0.0;
  204.    C C VCr2 A---<>          p[i][j][k]=0.0;

CC-6005 CC: SCALAR File = himeno.c, Line = 193
  A loop was unrolled 2 times.

CC-6204 CC: VECTOR File = himeno.c, Line = 193
  A loop was vectorized.

CC-6231 CC: VECTOR File = himeno.c, Line = 204
  A statement was replaced by a library call.
```

Questions About the Data?

- Check the Notes before each table in the text report
- Check pat_help
- Check man pages

```
Notes for table 5:

  The Total value for Process HiMem (MBytes), Process Time is the avg
  for the PE values.

  The value shown for Process HiMem is calculated from information in
  the /proc/self/numa_maps files captured near the end of the program.
  It is the total size of all pages, including huge pages, that were
  actually mapped into physical memory from both private and shared
  memory segments.

  This table shows only the maximum, median, minimum PE entries,
    sorted by Process Time.
```

Questions About the Data? (2)

- > pat_help environment . . .

```
pat_help environment (.=quit ,=back ^=up /=top ~=search)
=> PAT_RT_SAMPLING_DATA

    Specifies additional data to collect during a sampling
    experiment. The valid values are shown below.

    The value may be followed by '@ratio' which indicates the
    frequency at which the data is sampled. By default the data is
    sampled once for every 100 sampled program counter addresses. For
    example, if 'ratio' is '1', the additional data requested would
    be collected each time the program counter is sampled.
    If the 'ratio' is '1000', the additional data requested would
    be collected once every 1000 program counter samples.

    Collecting additional data during sampling is only supported in
    full-trace mode (see PAT_RT_SUMMARY).

  Additional topics that may follow "PAT_RT_SAMPLING_DATA":

    cray_pm      perfctr
    cray_rapl    rusage
    heap         sheap
    memory
```

Questions About the Data? (3)

- > pat_help environment PAT_RT_SAMPLING_DATA memory

```
pat_help environment PAT_RT_SAMPLING_DATA
(.=quit ,=back ^=up /=top ~=search) => memory

    memory    collect data about the current state of memory

      himem        -   memory high water mark
      rss          -   resident set size
      peak         -   maximum virtual memory used
      priv         -   private resident memory
      shared       -   shared resident memory
      proportional -   proportional resident memory
```

pat_info Utility

- Can be used to generate a quick summary statement regarding the contents of a CrayPat .ap2 file or set of files without running pat_report or Apprentice2
- Useful if you have multiple .ap2 files in a directory or if you want to review what experiments you have already performed
- Works on a single .ap2 file or a directory of .ap2 files
- When invoked with no arguments, the command looks in the current directory for .ap2 files
- When invoked with a directory argument, information about all .ap2 files in that directory are displayed

Example pat_info Utility Output

```
# When given a single .ap2 file argument, it defaults to the long form
# (counter lists are added with the -c option):

kay-esl$ pat_info -c sweep3d.mpi+17552-12s.ap2
ap2:                      sweep3d.mpi+17552-12s.ap2
ap2_size:     289792
RTS:                      yes
Experiment:   samp_cs_time
PE:                       CRAY
NumPEs:                   96
NumThreads:   0
NumLeafNodes:928
OpenMP:                   yes
Original prog:            /lus/scratch/clark/sweep3d/sweep3d.mpi+orig
prog_size:    (not available)
NumHWPC:      16
  CYCLES_RTC
  L1D:REPLACEMENT
  L2_RQSTS:ALL_DEMAND_DATA_RD
  FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE
  FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE
  FP_COMP_OPS_EXE:X87
  FP_COMP_OPS_EXE:SSE_PACKED_SINGLE
  SIMD_FP_256:PACKED_SINGLE
  FP_COMP_OPS_EXE:SSE_FP_PACKED_DOUBLE
  SIMD_FP_256:PACKED_DOUBLE
  L2_RQSTS:DEMAND_DATA_RD_HIT
  CPU_CLK_UNHALTED:THREAD_P
  CPU_CLK_UNHALTED:REF_P
  MEM_UOPS_RETIRED:ALL_LOADS
  DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK
  DTLB_STORE_MISSES:MISS_CAUSES_A_WALK
NumCPMC:      2
  PM_ENERGY:NODE
  PM_ENERGY:ACC
```

CrayPat-lite

Light-weight application profiling

Good place to start!

CrayPat-lite Overview

- **Provide automatic application performance statistics at the end of a job. Focus is to offer a simplified interface to basic application performance information for users not familiar with the Cray performance tools and perhaps new to application performance analysis.**
- **The tool is enabled by loading a module and rebuild**

```
> module load perftools-lite
> make clean && make
```

- **Program is automatically relinked to add instrumentation in a.out (`pat_build` step done for the user)**
  - .o files are automatically preserved
  - No modifications are needed to a batch script to run instrumented binary, since original binary is replaced with instrumented version
  - `pat_report` is automatically run before job exits.
  - Performance statistics are issued to stdout
  - User can use “classic” CrayPat for more in-depth performance investigation

Steps to Using CrayPat-lite

Access light version of performance tools software

> module load perftools-lite

Build program

> make → a.out (instrumented program)

Run program (no modification to batch script)

aprun a.out → Condensed report to stdout
a.out*.rpt (same as stdout)
a.out*.ap2
MPICH_RANK_XXX files

Predefined Set of Performance Experiments

- **Set of predefined experiments, enabled with the `CRAYPAT_LITE` environment variable (before compilation)**
  - `sample_profile`
  - `event_profile`
- **The `sample_profile` is equivalent to**

```
> pat_build -O apa a.out
```

  - Includes collection of summary CPU performance counters around MAIN
  - Includes Imbalance information.
- **The `event_profile` is equivalent to**

```
> pat_build -u -gmpi a.out
```

  - Provides profile based on summarization of events.
  - Includes OpenMP if these models are used within program.
  - Collection of summary CPU performance counters
  - Filter to only trace functions above 1200 bytes
    - In most cases, omits tiny repetitive functions that can perturb results.

Performance Statistics Available

Job information

- Number of MPI ranks, …
- Wallclock
- Memory high water mark
- Performance counters (CPU only)

```
Number of PEs (MPI ranks):   64
Numbers of PEs per Node:     32 PEs on each of 2 Nodes
Numbers of Threads per PE:    1
Number of Cores per Socket:  16
Execution start time:  Fri Feb 15 14:42:24 2013

Wall Clock Time:  122.608994 secs
High Memory:  45.70 MBytes
```

Profile of top time consuming routines with load balance

```
Samp% |  Samp  | Imb.  | Imb.  |Group
      |        | Samp  | Samp% | Function
      |        |       |       | PE=HIDE

100.0% | 14272.5 |   -- |    -- |Total
|-------------------------------------------
| 46.0% | 6561.4 |   -- |    -- |USER
||------------------------------------------
||  5.9% |  847.6 | 155.4 | 15.7% |collocate_core_1_
||  4.9% |  700.3 | 125.7 | 15.5% |integrate_core_2_
||  3.8% |  544.0 | 124.0 | 18.9% |collocate_core_2_
||  3.7% |  523.1 |  73.9 | 12.6% |integrate_core_1_
||==========================================
| 29.7% | 4239.6 |   -- |    -- |MPI
||------------------------------------------
||  9.3% | 1328.3 | 198.7 | 13.2% |mpi_alltoallv
||  4.2% |  598.5 |  71.5 | 10.8% |mpi_waitall
||  2.9% |  413.8 | 107.2 | 20.9% |MPI_WAITANY
||  2.9% |  409.1 |  66.9 | 14.3% |MPI_Comm_create
||==========================================
```

```
Time% |     Time |    Imb. |  Imb. |     Calls |Group
      |          |    Time | Time% |           |Function
      |          |         |       |           | PE=HIDE

100.0% | 101.961423 |       -- |    -- | 5315211.9 |Total
|---------------------------------------------------------------
| 92.5% | 94.267451 |       -- |    -- | 5272245.9 |USER
||--------------------------------------------------------------
|| 75.8% | 77.248585 | 2.356249 |  3.0% |    1001.0 |LAMMPS_NS::PairLJCut::compute
||  6.5% |  6.644545 | 0.105246 |  1.6% |      51.0 |LAMMPS_NS::Neighbor::half_bin_newton
||  4.1% |  4.131842 | 0.634032 | 13.5% |       1.0 |LAMMPS_NS::Verlet::run
||  3.8% |  3.841349 | 1.241434 | 24.8% | 5262868.9 |LAMMPS_NS::Pair::ev_tally
||  1.3% |  1.288463 | 0.181268 | 12.5% |    1000.0 |LAMMPS_NS::FixNVE::final_integrate
||==============================================================
|  7.0% |  7.110931 |       -- |    -- |   42637.0 |MPI
||--------------------------------------------------------------
||  4.8% |  4.851309 | 3.371093 | 41.6% |   12267.0 |MPI_Send
||  1.5% |  1.536106 | 2.592504 | 63.8% |   12267.0 |MPI_Wait
||==============================================================
```

Observations and Instructions on how to get more info.

Profiling for the GPU

Profiling OpenACC codes

- CrayPAT tracing offers a powerful profiling for OpenACC codes. (Sampling does not collect GPU performance data)
- Load the GPU module and the performance tools

```
> module load craype-accel-nvidia35
> module load perftools
```

- Recompile your program

```
> ftn -c my_program.f
> ftn -o my_program my_program.o
```

- Instrument the application for tracing and execute

```
> pat_build -w my_program
> aprun -n pes my_program
```

- Generate a report out of the raw data file(s)

```
> pat_report -o report.txt my_program*.xf
```

Contents of report.txt (Table 1)

```
Table 1:  Profile by Function Group and Function

 Time%  |     Time  | Imb.  |  Imb.  | Calls  |Group
        |           | Time  | Time%  |        | Function

 100.0% | 16.409900 |    -- |     -- | 1252.0 |Total
|-------------------------------------------------------------------------
| 100.0% | 16.409731 |    -- |     -- |  851.0 |USER
||------------------------------------------------------------------------
||  51.2% |  8.403343 |    -- |     -- |    1.0 |mg_
||  34.3% |  5.622111 |    -- |     -- |  170.0 |resid_.ACC_COPY@li.615
||  11.8% |  1.936478 |    -- |     -- |  170.0 |resid_.ACC_COPY@li.639
||   2.7% |  0.440894 |    -- |     -- |  170.0 |resid_.ACC_SYNC_WAIT@li.639
||   0.0% |  0.005727 |    -- |     -- |  170.0 |resid_.ACC_KERNEL@li.615
||   0.0% |  0.001178 |    -- |     -- |  170.0 |resid_.ACC_REGION@li.615
||========================================================================
|   0.0% |  0.000170 |    -- |     -- |  401.0 |ETC
||------------------------------------------------------------------------
```

- Two ACC_COPY lines report data movements at either end of the OpenACC parallel region.
- ACC_KERNEL is essentially zero as it is launched asynchronously.
- GPU time is measured in the ACC_SYNC_WAIT
- ACC_REGION measures internal ops (set up pointers for transfer.)

Contents of report.txt (Table 2)

```
Table 2:  Time and Bytes Transferred for Accelerator Regions

  Host   | Host   |  Acc   | Acc Copy  | Acc Copy  | Events   |Calltree
 Time%   | Time   | Time   |        In |       Out |          |
         |        |        | (MBytes)  | (MBytes)  |          |

 100.0% | 8.007 | 7.962 |     12341 |      6171 |     850 |Total
|-------------------------------------------------------------------------------------
| 100.0% | 8.007 | 7.962 |     12341 |      6171 |     850 |mg_
||------------------------------------------------------------------------------------
||  50.0% | 4.005 | 3.969 |      6314 |      3157 |     735 |mg3p_
3|        |       |       |           |           |         | resid_
4|        |       |       |           |           |         |  resid_.ACC_REGION@li.615
|||||---------------------------------------------------------------------------------
5||||  36.2% | 2.898 | 2.877 |      6314 |        -- |     147 |resid_.ACC_COPY@li.615
5||||  10.8% | 0.867 | 0.860 |        -- |      3157 |     147 |resid_.ACC_COPY@li.639
5||||   2.9% | 0.235 |    -- |        -- |        -- |     147 |resid_.ACC_SYNC_WAIT@li.639
5||||   0.1% | 0.004 | 0.232 |        -- |        -- |     147 |resid_.ACC_KERNEL@li.615
5||||   0.0% | 0.001 |    -- |        -- |        -- |     147 |resid_.ACC_REGION@li.615(exclusive)
```

- The resid routine is called from several points – table shows one
- Table shows details on data transfers and timings for CPU and GPU.
- ACC_SYNC_WAIT time on CPU includes kernel time on GPU.
- For MPI programs the statistics are averaged over the PE.

Contents of report.txt (pat_build -u …)

```
Table 1:  Profile by Function Group and Function

 Time%  |      Time  | Imb.  |  Imb.  |    Calls  |Group
        |            | Time  | Time%  |           | Function

 100.0% | 16.452925 |    -- |     -- | 265303.0 |Total
|-----------------------------------------------------------------------------
| 100.0% | 16.452760 |    -- |     -- | 264902.0 |USER
||----------------------------------------------------------------------------
||  34.2% |  5.621172 |    -- |     -- |    170.0 |resid_.ACC_COPY@li.615
||  19.4% |  3.199216 |    -- |     -- |    168.0 |psinv_
||  11.8% |  1.940111 |    -- |     -- |    170.0 |resid_.ACC_COPY@li.639
||  10.7% |  1.764268 |    -- |     -- | 131072.0 |vranlc_
||   7.4% |  1.217534 |    -- |     -- |    147.0 |rprj3_
||   6.3% |  1.033920 |    -- |     -- |    147.0 |interp_
||   4.3% |  0.709337 |    -- |     -- |    151.0 |zero3_
||   2.7% |  0.441237 |    -- |     -- |    170.0 |resid_.ACC_SYNC_WAIT@li.639
||   1.5% |  0.240856 |    -- |     -- |      2.0 |zran3_
||   1.0% |  0.170554 |    -- |     -- |    487.0 |comm3_
|=============================================================================
```

- resid kernel no longer dominant in the whole picture but associated data transfers still very expensive.

Accelerator Table Column definitions

- **Host Time%**
  - percentage of wallclock time for events
- **Host Time**
  - wallclock time, in seconds, for the event
- **Acc Time**
  - amount of time the event executed on the accelerator
- **Acc Copy In**
  - amount of data copied to the accelerator
- **Acc Copy Out**
  - amount of data copied from the accelerator
- **Calls**
  - the number of time the event occurred

- **All of the above are summed for regions and functions**

Accelerator Table Column definitions

- Notes section at the beginning of the tables contains helpful information describing how the table was generated and suggestions on how to produce additional related tables.
- Data presented in default text report is organized as a calltree with functions/accelerated regions sorted in decreasing order by Host Time
- Called functions, regions and events are indented to the right
- Left-most column represents indentation in table
- By default, cells in accelerator tables that have no data are marked with '-'

Profiling with GPU Hardware Counters

- **CrayPAT supports a wide range of accelerator performance counter**

- **A predefined set of groups has been created for ease of use (combines events that can be counted together.)**

```
> module load perftools
> man accpc
> more $CRAYPAT_ROOT/share/CounterGroups.nvidia_k20x
```

- **Enable collection similarly to CPU counter collection**
  - PAT_RT_PERFCTR=group or events

- **Set the PAT_RT_ACCPC variable appropriately and run the instrumented (tracing) program.**

GPU Counter Statistics

Used PAT_RT_ACCPC=ipc_inst_rep_ovr to generate the following table after pat_report.

```
Table 3:  ACC Performance Counter Data

   Acc    | inst_executed    | inst_issued1    | inst_issued2    | thread_inst_executed     |  ipc   |  Acc    |Calltree
  Time%   |                  |                 |                 |                          |        | Util    |

  100.0% |      273341540 |     245524426 |      54206381 |           8261569560 | 0.003 | 45.5% |Total
 |-----------------------------------------------------------------------------------------------------------------------
 | 100.0% |      273341540 |     245524426 |      54206381 |           8261569560 | 0.003 | 45.5% |mg_
 ||----------------------------------------------------------------------------------------------------------------------
 ||  50.2% |      129718820 |     117634433 |      25615019 |           3940872320 | 0.003 | 22.9% |resid_
 3|        |                |               |               |                      |       |       | resid_.ACC_REGION@li.615
 ||||--------------------------------------------------------------------------------------------------------------------
 4|||  34.1% |              0 |             0 |             0 |                    0 |     0 | 15.5% |resid_.ACC_COPY@li.615
 4|||  13.5% |              0 |             0 |             0 |                    0 |     0 |  6.1% |resid_.ACC_COPY@li.639
 4|||   2.6% |      129718820 |     117634433 |      25615019 |           3940872320 | 0.061 |  1.2% |resid_.ACC_KERNEL@li.615
 ||||====================================================================================================================
 ||  49.8% |      143622720 |     127889993 |      28591362 |           4320697240 | 0.004 | 22.7% |mg3p_
 3|        |                |               |               |                      |       |       | resid_
 4|        |                |               |               |                      |       |       |  resid_.ACC_REGION@li.615
 |||||-------------------------------------------------------------------------------------------------------------------
 5||||  36.1% |              0 |             0 |             0 |                    0 |     0 | 16.4% |resid_.ACC_COPY@li.615
 5||||  10.8% |              0 |             0 |             0 |                    0 |     0 |  4.9% |resid_.ACC_COPY@li.639
 5||||   2.9% |      143622720 |     127889993 |      28591362 |           4320697240 | 0.060 |  1.3% |resid_.ACC_KERNEL@li.615
 |=======================================================================================================================
```

Loop mark and runtime commentary

- **Cray compiler commentary is available as annotated loop mark listing of the source file giving further information**
  - Loop transformation and optimization.
  - OpenACC transformations
- **Is requested by the compiler flag `-h list=a`**
  - Information is written to files whose name has the same stem as the source files and the extension .lst
  - Use `-h list=ad` for even lower level information, e.g., pattern matched routines or understand OpenACC synchronisation points.
- **Runtime commentary is obtained by setting the variable CRAY_ACC_DEBUG={1,2,3} before execution.**
  - Provides evidence that kernels have been executed on the GPU.
  - Information on data transfers between CPU and GPU.
  - Does not need to be set at compile time.

Example loop mark listing

```
   12.                  int i,j,k;
   13.
   14.                  #pragma acc kernels deviceptr(a, b) copyout(c[0:size*size])
   15.  + G----------<  {
   16.    G
   17.    G             // Initialize matrices.
   18.    G             #pragma acc loop independent
   19.    G gG-------<  for (i = 0; i < size; ++i) {
   20.    G gG                  #pragma acc loop independent
   21.    G gG g-----<          for (j = 0; j < size; ++j) {
   22.    G gG g                        a[i*size+j] = (float)i + j;
   23.    G gG g                        b[i*size+j] = (float)i - j;
   24.    G gG g                        c[i*size+j] = 0.0f;
   25.    G gG g----->          }
   26.    G gG------->  }
   27.    G
   …
 CC-6413 CC: ACCEL File = matrix-acc-alloc.c, Line = 15
  A data region was created at line 15 and ending at line 39.

CC-6419 CC: ACCEL File = matrix-acc-alloc.c, Line = 15
  Allocate memory for user shaped variable "c" on accelerator, copy back at line 39 (acc_copyout).

CC-6401 CC: ACCEL File = matrix-acc-alloc.c, Line = 19
  A loop was placed on the accelerator.

CC-6430 CC: ACCEL File = matrix-acc-alloc.c, Line = 19
  A loop was partitioned across the thread blocks.

CC-6430 CC: ACCEL File = matrix-acc-alloc.c, Line = 21
  A loop was partitioned across the 128 threads within a threadblock.
```

Region accelerated

Compiler information

Example runtime commentary

```
ACC: Start transfer 6 items from mg_v03.f:615
ACC:       allocate, copy to acc 'a' (32 bytes)
ACC:       allocate 'r' (137388096 bytes)
ACC:       allocate, copy to acc 'u' (137388096 bytes)
ACC:       allocate, copy to acc 'v' (137388096 bytes)
ACC:       allocate <internal> (530432 bytes)
ACC:       allocate <internal> (530432 bytes)
ACC: End transfer (to acc 274776224 bytes, to host 0 bytes)
ACC: Execute kernel resid_$ck_L615_1 blocks:256 threads:128 async(auto) from
mg_v03.f:615
ACC: Wait async(auto) from mg_v03.f:639
ACC: Start transfer 6 items from mg_v03.f:639
ACC:       free 'a' (32 bytes)
ACC:       copy to host, free 'r' (137388096 bytes)
ACC:       free 'u' (137388096 bytes)
ACC:       free 'v' (137388096 bytes)
ACC:       free <internal> (0 bytes)
ACC:       free <internal> (0 bytes)
```

Allocation and copy to the GPU.

Kernel execution.

Copy data back to Host.

Nvidia Performance Analysis Tools

Visualizing the GPU activity

Overview

- **The CCE interprets OpenACC directives to create GPU kernels written PTX.**
  - PTX is a low level, assembler like machine language used by Nvidia GPUs.
  - Allows us to use tools like Nvidia command line profiler to get information about the code (data transfer times and size, kernel execution times,..)

- **A description of Nvidia tools for profiling and debugging can be found at** docs.nvidia.com/cuda/#tools-manuals
  - CUDA-GDB & CUDA-MEMCHECK
  - Visual profiler nvpp
  - Command line profiler
  - nvprof

Using the command line and visual profiler (untested!)

- **Enable the profiler and run your application in PBS**

```
> module load craype-accel-nvidia35
> export COMPUTE_PROFILE=1
> export COMPUTE_PROFILE_CSV=1
> aprun <your_application> <arguments>
```

- **You should get a cuda_profile*.log file. (low information)**
- **To get more information create a <configfile> in the run directory and set**

```
> export COMPUTE_PROFILE_CONFIG=<configfile>
```

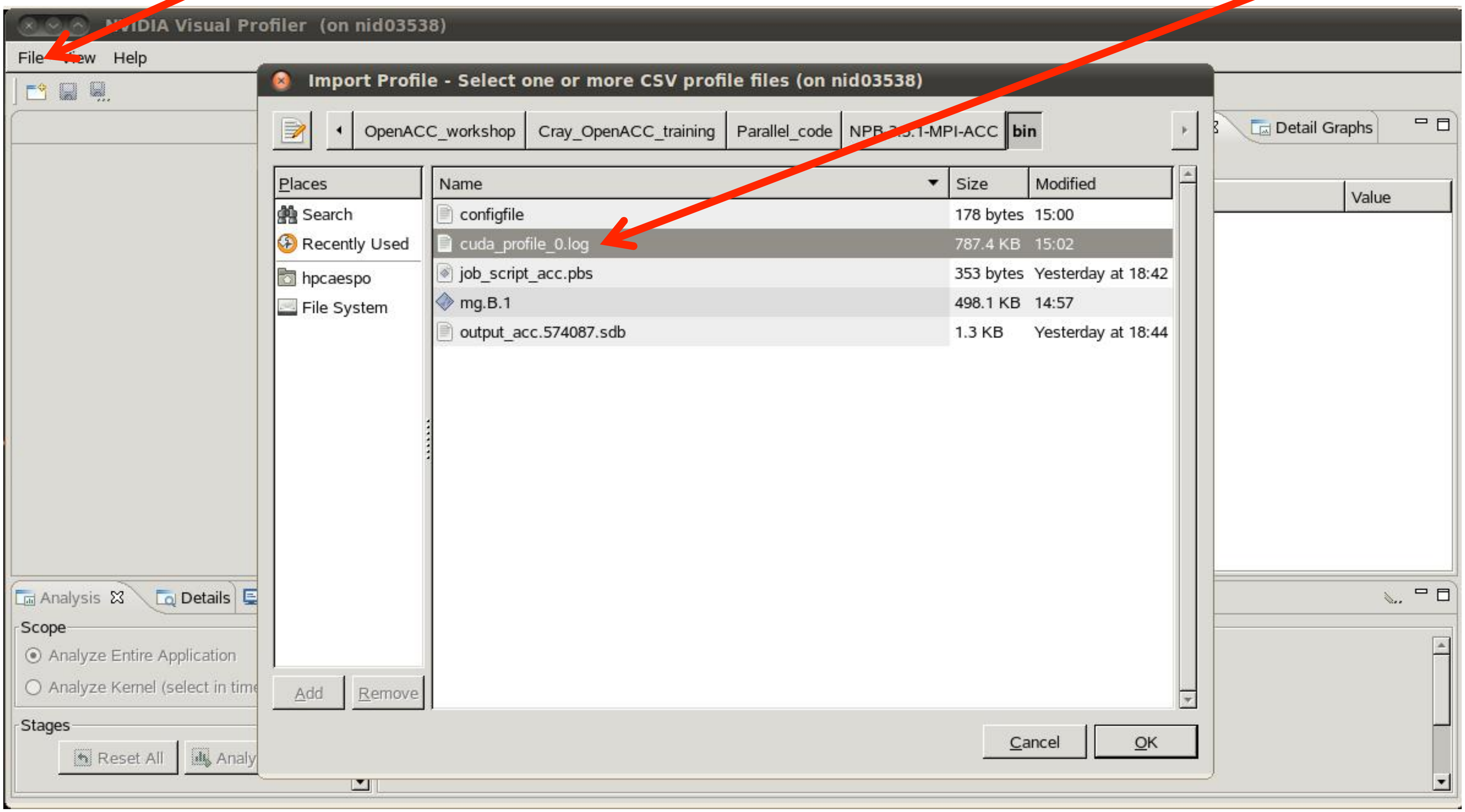
```
gpustarttimestamp
gridsize3d
threadblocksize
dynsmemperblock
stasmemperblock
regperthread
memtransfersize
memtransferdir
streamid
countermodeaggregate
active_warps
active_cycles
```

- **Rerun your application and launch the visual profiler once PBS session over:**

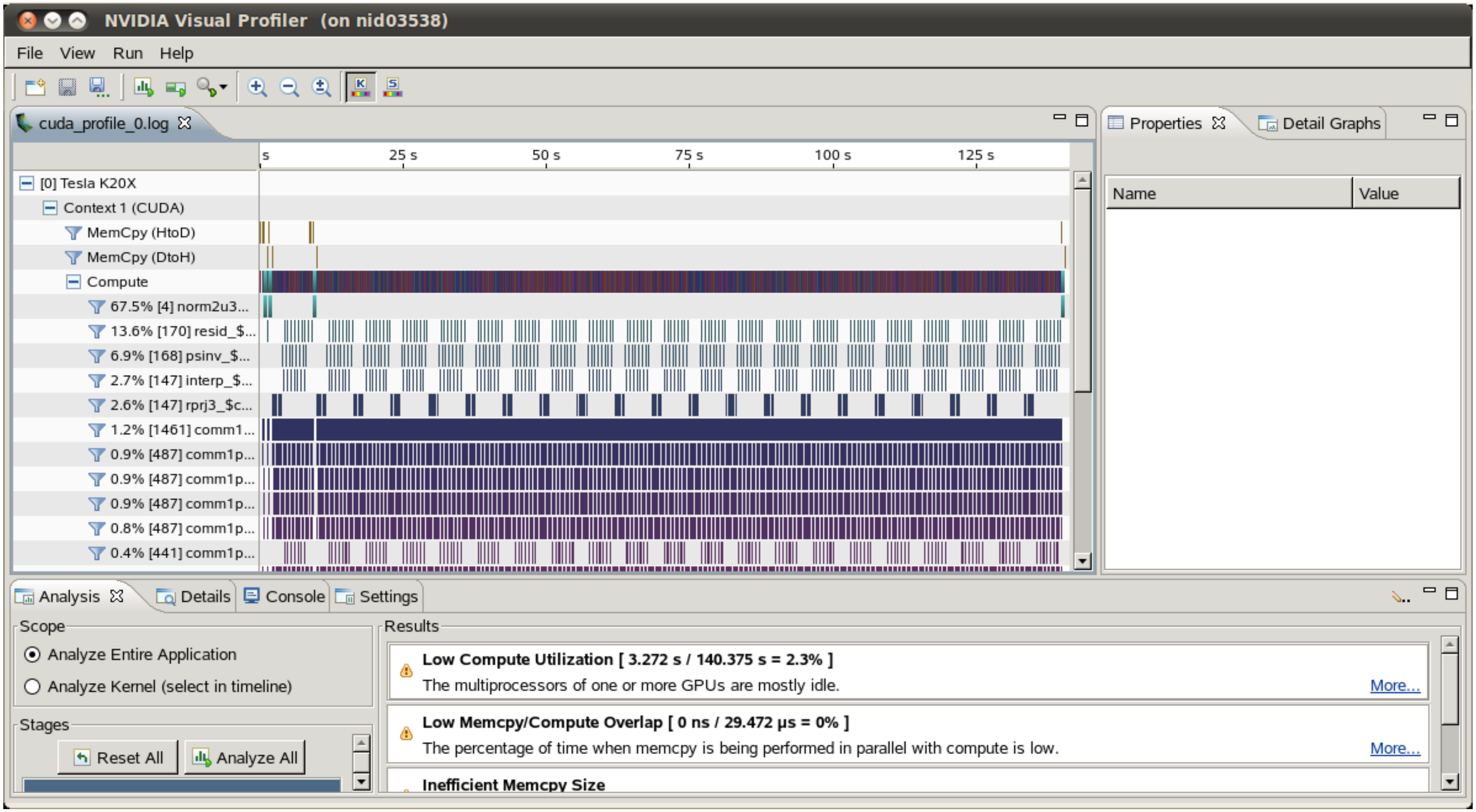
```
> aprun <your_application> <arguments>
> nvvp &
```

Using the command line and visual profiler

- Choose File -> Import CSV Profile … and select the *.log



Using the command line and visual profiler



Also useful to investigate (un)expected behaviour.

Profiling for the Intel Phi

CrayPat and Intel Xeon Phi

- **CrayPat supported for native (autonomous) mode only**
- **Supported functionality:**
  - Sampling of MPI and OpenMP jobs in autonomous mode.
  - Tracing of MPI and OpenMP jobs in autonomous mode.
    - Note that OpenMP timing information is associated with the calling function. The pat_region API can be used around OpenMP regions for localized timing information.
  - Cray Apprentice2 includes performance information for jobs that ran on the Xeon Phi.
  - A subset of the predefined trace groups is supported. The pat_build utility will issue a message if an unsupported groups is requested.
- **Unsupported functionality:**
  - Reveal
  - CrayPat-lite
  - static linking
  - PAPI: no performance counter support is available.
  - tracing statistics associated with an OpenMP region
  - Offload mode not supported in general. BUT, tracing and use of pat_region API calls around loops containing offload directives may return useful information. Sampling not currently supported in offload mode.

Using CrayPat for the Phi

- **Load environment as usual for native mode**
  - module swap PrgEnv-cray PrgEnv-intel
  - module unload cray-libsci atp craype-sandybridge craype-ivybridge
  - module load craype-intel-knc
- **Load instrumentation module**
  - module load perftools
- **Build executable with dynamic linking**
  - cc -lopenmp hello.c \
    -Wl,-rpath=$INTEL_PATH/compiler/lib/mic \
    -Wl,-rpath=/opt/cray/k1om/lib64
- **Use pat_build as normal to instrument program and pat_report or Apprentice2 to report resulting data**