



Getting feedback @ EuroHack

Mandes Schönherr,

Alistair Hart

mandes.schoenherr@cray.com,

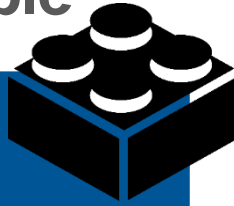
ahart@cray.com

(plus help from PGI and Nvidia folk)

What's going on?

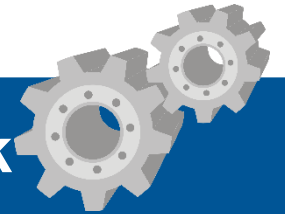
- You need feedback to see what is happening
- Two forms of feedback available

Compiler feedback



- describes what the compiler intends to do
- No performance overhead
- But you need to ask for it
- You should always ask for it

Runtime feedback



- describes what the runtime actually did
- performance overhead
- Use it when developing, not for performance testing or production
- CrayPAT, nvprof are the "Rolls-Royce" solutions
- There are also some less powerful methods, described here

Compiler feedback



Cray compiler:

- Compiler option: **-hlist=a**
- For every source file (foo.f, foo.c), get a new file when compile: foo.lst
- Lots of information about what compiler did (or didn't do)

PGI compiler:

- Compiler option: **-Minfo=accel**
- Information written to STDOUT when you compile

source with line numbers

```
...  
270. + 1 b-----< DO k = 2,kmax-1  
271. + 1 b b-----< DO j = 2,jmax-1  
272. 1 b b Vr3-----< DO i = 2,imax-1  
273. 1 b b Vr3          S0 = a(i,j,k,1)* ...  
...
```

loop handling
annotations

additional messages below

Quick runtime feedback



- A really quick way to see what is happening with your code as it runs on the GPU (or GPUs)

It's not scalable:

- There is a lot of information
 - Commentary: Event-by-event, ball-by-ball, blow-by-blow
- The longer your code runs, the more information there is
 - Multiplied by the number of MPI ranks
- It will slow down code execution
 - And probably skew the profile slightly

Various quick methods:

- Each enabled by environment variable at runtime (in jobscript)
 - No need to recompile
- Each gives text output
- **Don't use more than one at once!**

Nvidia Compute Profiler (PGI, Cray, CUDA)



- **export COMPUTE_PROFILE=1**

- **Gives timing information for each event**

- Data transfers
- Kernel executions
- Works for PGI and Cray compilers and CUDA
- Written to a new text file (you can specify the name using **COMPUTE_PROFILE_LOG**)
- Configurable using config file, specified in **COMPUTE_PROFILE_CONFIG**

```
$> cat cuda_profile_0.log
# NV_Warning: The legacy Command Line Profiler is deprecated and will be no longer available as of
the next major release of the CUDA toolkit. Please
use nvprof.
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla K40s
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR 145bd45225a4477a
method,gputime,cputime,occupancy
method=[ initmt_$ck_L208_17 ] gputime=[ 4392.704 ] cputime=[ 15.115 ] occupancy=[ 1.000 ]
method=[ initmt_$ck_L230_19 ] gputime=[ 3615.232 ] cputime=[ 11.049 ] occupancy=[ 1.000 ]
method=[ memcpyHtoD ] gputime=[ 1.568 ] cputime=[ 8.291 ]
method=[ jacobi_clone_5738_1_$ck_L157_3 ] gputime=[ 3848.672 ] cputime=[ 9.810 ] occupancy=[ 0.250 ]
method=[ memcpyDtoH ] gputime=[ 2.464 ] cputime=[ 19.330 ]
method=[ jacobi_clone_5738_1_$ck_L157_5 ] gputime=[ 438.240 ] cputime=[ 8.825 ] occupancy=[ 1.000 ]
method=[ jacobi_clone_5738_1_$ck_L157_3 ] gputime=[ 3779.968 ] cputime=[ 4.723 ] occupancy=[ 0.250 ]
method=[ memcpyDtoH ] gputime=[ 2.464 ] cputime=[ 17.377 ]
method=[ jacobi_clone_5738_1_$ck_L157_5 ] gputime=[ 436.544 ] cputime=[ 7.388 ] occupancy=[ 1.000 ]
method=[ jacobi_clone_5738_1_$ck_L157_3 ] gputime=[ 3886.144 ] cputime=[ 3.919 ] occupancy=[ 0.250 ]
```

- **Very useful to get some quick profiling**

- See how much time is spent in computation vs. data transfers

- **Tip from Nvidia:**

- To integrate Compute Profiler output with the application output:
 - **export COMPUTE_PROFILE_LOG=/dev/stdout**

```
$> cat compute_profile_config
# compute_profile_config
method
gputime
cputime
occupancy
memtransfersize
```


Nvidia nvprof (PGI, Cray, CUDA)



- **A better (newer) solution from Nvidia**
 - By default gives aggregated program view of performance
 - Can also give an event-by-event timeline
 - The data can also be loaded into the Nvidia nvvp GUI
- **There is a trick to do this in your jobscript**

```
$> cat job.batch

...
export PMI_NO_FORK=1
# aprun <aprun_options> <EXE> <EXE options> # without nvprof
srun <srun_options> -b nvprof <EXE> <EXE options>
```

- The resulting summary is printed at the end of the job

CRAY_ACC_DEBUG (just Cray)



- **export CRAY_ACC_DEBUG=1, 2 or 3**
 - Recommend level 2
- **Gives array movement information**
 - Name of the arrays
 - Number of bytes transferred
 - Written to STDERR
 - Just for the Cray compiler
 - Has an API to restrict when information is listed
 - Fortran example on next slide.
 - For C/C++ and more details, see: **man openacc**
- **Very useful to understand data movements**
 - What takes the time, debugging correctness errors

```
$> cat job.log
ACC: Initialize CUDA
ACC: Get Device 0
ACC: Create Context
ACC: Set Thread Context
ACC: Start transfer 7 items from himeno_F_v03.F90:116
ACC:     allocate 'a' (136855584 bytes)
ACC:     allocate 'b' (102641688 bytes)
ACC:     allocate 'bnd' (34213896 bytes)
ACC:     allocate 'c' (102641688 bytes)
ACC:     allocate 'p' (34213896 bytes)
ACC:     allocate 'wrk1' (34213896 bytes)
ACC:     allocate 'wrk2' (34213896 bytes)
ACC: End transfer (to acc 0 bytes, to host 0 bytes)
ACC: Start transfer 6 items from himeno_F_v03.F90:208
ACC:     present 'a' (136855584 bytes)
ACC:     present 'b' (102641688 bytes)
ACC:     present 'bnd' (34213896 bytes)
ACC:     present 'c' (102641688 bytes)
ACC:     present 'p' (34213896 bytes)
ACC:     present 'wrk1' (34213896 bytes)
ACC: End transfer (to acc 0 bytes, to host 0 bytes)
ACC: Execute kernel initmt_$ck_L208_17 blocks:129 threads:128
      async(auto) from himeno_F_v03.F90:208
...
```

CRAY_ACC_DEBUG API (Fortran example)



- Using the API to limit the runtime commentary to parts of interest

unset
commentary

set
commentary

```
! Execute code with CRAY_ACC_DEBUG=1, 2 or 3 in jobscript
PROGRAM main

USE openacc_lib                      ! exposes the API calls
INTEGER :: cray_acc_debug_orig      ! preserve original value

<start of executable code>
cray_acc_debug_orig = cray_acc_get_debug_global_level()
CALL cray_acc_set_debug_global_level(0)

<code without commentary>

CALL cray_acc_set_debug_global_level(cray_acc_debug_orig)
<code with commentary>
CALL cray_acc_set_debug_global_level(0)

<code without commentary>

END PROGRAM main
```


PGI_ACC_NOTIFY (just PGI)



- **export PGI_ACC_NOTIFY=1, 3, 7, 15, 31**
 - Recommend level 3
(kernel launches, data movement)
- **Gives array movement information**
 - Number of bytes transferred
 - Written to STDERR
 - Just for the PGI compiler
- **Very useful to understand data movements**
 - What takes the time, debugging correctness errors
- **export PGI_ACC_TIME=1**
 - gives a summarised output for whole program
 - probably shouldn't do this at the same time as PGI_ACC_NOTIFY

```
$> cat job.out
...
upload CUDA data file=/.../himeno_F_v03.F90 function=jacobi line=288
device=0 threadid=1 bytes=8
launch CUDA kernel file=/.../himeno_F_v03.F90 function=jacobi line=288
device=0 threadid=1 num_gangs=126 num_workers=1 vector_length=128
grid=126 block=128 shared memory=2048
launch CUDA kernel file=/.../himeno_F_v03.F90 function=jacobi line=288
device=0 threadid=1 num_gangs=1 num_workers=1 vector_length=256
grid=1 block=256 shared memory=2048
download CUDA data file=/.../himeno_F_v03.F90 function=jacobi line=288
device=0 threadid=1 bytes=8
...
```

```
$> cat job.out
...
Accelerator Kernel Timing data
/.../himeno_F_v03.F90
initmt NVIDIA devicenum=0
time(us): 9,421
208: data region reached 2 times
210: compute region reached 1 time
210: kernel launched 1 time
grid: [129] block: [128]
device time(us): total=4,886 max=4,886 min=4,886 avg=4,886
elapsed time(us): total=4,934 max=4,934 min=4,934 avg=4,934
...
```

MPI programs



```
$> cat job.batch
```

```
...
```

```
export PMI_NO_FORK=1
```

```
# aprun <aprun_options> <EXE> <EXE options> # without wrapper
```

```
srunk <srunk_options> bash wrapper.batch <EXE> <EXE options>
```

- The problem is that all the information from each rank comes out at once, and gets mixed up together
 - Better to separate the information to one file per rank
- The trick in the jobscript:
 - Use a **wrapper script to separate the output**
 - The profile method could also be selected

```
$> cat wrapper.batch
```

```
#!/bin/bash
```

```
# ONLY ACTIVATE ONE RUNTIME COLLECTION METHOD AT A TIME!!!
```

```
# A name for the files (replace F00 as appropriate)
```

```
jobstem=$(printf "F00.%03d" $ALPS_APP_PE)
```

```
# NVIDIA COMPUTE_PROFILER: set this 1 to activate
```

```
export COMPUTE_PROFILE=0
```

```
# Collect output in separate files, one per process
```

```
export COMPUTE_PROFILE_LOG=./${jobstem}.compprof
```

```
# Tune what is collected (optional)
```

```
export COMPUTE_PROFILE_CONFIG=compute_profile_config
```

```
# Collect CCE runtime information: set this 1,2,3 to activate
```

```
export CRAY_ACC_DEBUG=0
```

```
# Collect PGI runtime information: set this 1,3 etc. to activate
```

```
export PGI_ACC_NOTIFY=0
```

```
# Now execute binary with appropriate options
```

```
# Pipe STDERR to separate files
```

```
# (to catch CRAY_ACC_DEBUG, PGI_ACC_NOTIFY commentary)
```

```
exec $* 2> ${jobstem}.err
```

```
# EOF
```

COMPUTE | STORE | ANALYZE