Lessons & Tips from OLCF’s Crusher Hackathons

Tom Papatheodore
HPC Engineer
System Acceptance & User Environment Group
Oak Ridge Leadership Computing Facility (OLCF)
Oak Ridge National Laboratory (ORNL)

December 1, 2022
Crusher Test and Development System (TDS)

- Crusher is OLCF's 192-node TDS with identical hardware as Frontier: [https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html](https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html)


- Exascale Computing Project (ECP): [https://www.exascaleproject.org/research/](https://www.exascaleproject.org/research/)

- Work with vendor partners to deliver documentation & training.
- Work with vendors to hold Crusher Office Hours (every Monday).
- Work with vendors and ECP to deliver Crusher Hackathons
Crusher Hackathons

• Goal: Help CAAR and ECP teams drive toward their development goals and existing milestones.
  - Achieving expected performance, scaling to multiple nodes, profiling their application, working through compiler issues, troubleshooting runtime errors, etc.

• 3 days of dedicated support from OLCF, HPE, AMD, and ECP staff.

• Held virtually using Zoom for team breakout rooms + Slack for overall persistent communication.

• Additional benefits:
  - Teams learn how to use vendor tools and give feedback
  - Large push from multiple application/software teams helps identify software, compiler, hardware bugs, and generally help harden the system
  - Opportunities for ECP ST and AD teams to integrate software, showcase helpful tools, etc.
  - Collect lessons learned to inform topics for special sessions, create additional documentation, and share information with future users and other centers with similar architectures (LUMI, Setonix, etc.).
Frontier Compute Node

[Diagram showing the structure of the Frontier Compute Node, including NVMe SSD, PCI SW, and connections to GPUs, NICs, and other components.]

KEY:
- 64 GB HBM 1.6 TB/s GPU
- 64 GB HBM 1.6 TB/s GPU Mi250X
- Infinity Fabric GPU-GPU (50+50 GB/s)
- PCIe Gen4 (8+8 GB/s)
- Ethernet (25+25 GB/s)
Frontier Overview

Extraordinary Engineering

Built by HPE

Olympus rack
- 128 AMD nodes
- 8,000 lbs
- Supports 400 KW

Powered by AMD

AMD node
- 1 AMD “Optimized 3rd Gen EPYC” CPU
- 4 AMD MI250X GPUs
- 512 GiB DDR4 memory on CPU
- 512 GiB HBM2e total per node
  (128 GiB HBM per GPU)
- Coherent memory across the node
- 4 TB NVM
- GPUs & CPU fully connected with AMD Infinity Fabric
- 4 Cassini NICs, 100 GB/s network BW

Compute blade
- 2 AMD nodes

System
- 2.0 EF Peak DP FLOPS
- 74 compute racks
- 29 MW Power Consumption
- 9,408 nodes
- 9.2 PiB memory (4.6 PiB HBM, 4.6 PiB DDR4)
- Cray Slingshot network with dragonfly topology
- 37 PB Node Local Storage
- 716 PB Center-wide storage
- 4,000 ft² footprint

All water cooled, even DIMMS and NICs
Lessons Learned
A Common Story...

In most cases, initial port was fairly straightforward, but some optimization was needed to realize expected performance.

- Port code from Summit (NVIDIA V100) to Crusher (AMD MI250X)
- Find initial performance on Crusher is similar or less performant than on Summit – what now?
- Many application teams found similar issues that needed to be addressed
  - Understand the performance expected for your application on Crusher relative to Summit
  - The usual suspects for optimization: occupancy, register usage, kernel launch parameters
  - System-specific considerations: Slurm bindings, hw atomics, managed memory

Addressing the subset of these issues relevant to a particular application has typically resulted in expected performance.
Expected Performance

Give participants performance expectations on Frontier relative to Summit

• Is your application GPU-bound or data-transfer-bound?
  • If GPU-bound, is it compute-bound or memory-bound?
    • Compute-bound: compute performance
    • Memory-bound: HBM, HBM bandwidth, L1 cache
    • This is where kernel profiling can be helpful
  • If data-transfer-bound, consider CPU-to-GPU bandwidth in table.

<table>
<thead>
<tr>
<th></th>
<th>V100</th>
<th>MI250X (GCD)</th>
<th>MI250X (GCD) / V100</th>
<th>Summit Node</th>
<th>Crusher Node</th>
<th>Crusher Node / Summit Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute performance</td>
<td>~7.8 TF</td>
<td>~26 TF</td>
<td>~3.3X</td>
<td>~47 TF</td>
<td>~208 TF</td>
<td>~4.4X</td>
</tr>
<tr>
<td>HBM</td>
<td>16 GB</td>
<td>64 GB</td>
<td>4X</td>
<td>96 GB</td>
<td>512 GB</td>
<td>~5.3X</td>
</tr>
<tr>
<td>HBM bandwidth</td>
<td>0.9 TB/s</td>
<td>1.6 TB/s</td>
<td>~1.8X</td>
<td>5.4 TB/s</td>
<td>12.8 TB/s</td>
<td>~2.4X</td>
</tr>
<tr>
<td>CPU-to-GPU bandwidth</td>
<td>50 GB/s</td>
<td>36 GB/s</td>
<td>~0.7X</td>
<td>300 GB/s</td>
<td>288 GB/s</td>
<td>~0.96X</td>
</tr>
<tr>
<td>L1 cache</td>
<td>Up to 128 KB</td>
<td>16 KB</td>
<td>0.125X – 0.5X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 cache</td>
<td>6 MB</td>
<td>8 MB</td>
<td>1.3X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network bandwidth</td>
<td></td>
<td></td>
<td></td>
<td>25 GB/s</td>
<td>100 GB/s</td>
<td>4X</td>
</tr>
</tbody>
</table>
Recall: MPI ranks should target the GPU associated with their L3 cache region

Always check process/thread/GPU bindings: https://code.ornl.gov/olcf/hello_jobstep

- **GPU_ID** – the node-level (or global) GPU ID read from ROCR_VISIBLE_DEVICES.
- **RT_GPU_ID** – the HIP runtime GPU ID (as reported from hipGetDevice).
- **Bus_ID** – the physical bus ID associated with the GPUs. Comparing the bus IDs is meant to definitively show that different GPUs are being used.
$ OMP_NUM_THREADS=1 srun -N1 -n8 --gpus-per-task=1 ./hello_jobstep | sort

Each rank has access to only 1 GPU, but **not ideal** mapping.

- MPI rank 0 maps to GPU 0
- MPI rank 1 maps to GPU 1
- Etc.
$OMP_NUM_THREADS=1 srun -N1 -n8 -c8 --gpus-per-task=1 --gpu-bind=closest ./hello_jobstep | sort

Each rank has access to only 1 GPU, and mapping is ideal:

- MPI rank 0 maps to GPU 4
- MPI rank 1 maps to GPU 5
- Etc.
$OMP_NUM_THREADS=1 srun -N1 -n8 --gpus-per-node=8 ./hello_jobstep | sort

MPI 000 - OMP 000 - HWT 004 - Node crusher132 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 001 - OMP 000 - HWT 008 - Node crusher132 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 002 - OMP 000 - HWT 017 - Node crusher132 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 003 - OMP 000 - HWT 024 - Node crusher132 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 004 - OMP 000 - HWT 032 - Node crusher132 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 005 - OMP 000 - HWT 040 - Node crusher132 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de
MPI 006 - OMP 000 - HWT 056 - Node crusher132 - RT_GPU_ID 0,1,2,3,4,5,6,7 - GPU_ID 0,1,2,3,4,5,6,7 - Bus_ID c1,c6,c9,ce,d1,d6,d9,de

All MPI ranks have access to all GPUs

- Ok, if application has built-in logic to map ranks to GPUs.
- Otherwise leads to all ranks targeting the same GPU 0.
$ OMP_NUM_THREADS=1 srun -N1 -n8 --gpus-per-node=8 --gpu-bind=closest ./hello_jobstep | sort

MPI 000 - OMP 000 - HWT 003 - Node crusher132 - RT_GPU_ID 0 - GPU_ID 4 - Bus_ID d1
MPI 001 - OMP 000 - HWT 009 - Node crusher132 - RT_GPU_ID 0 - GPU_ID 5 - Bus_ID d6
MPI 002 - OMP 000 - HWT 016 - Node crusher132 - RT_GPU_ID 0 - GPU_ID 2 - Bus_ID c9
MPI 003 - OMP 000 - HWT 024 - Node crusher132 - RT_GPU_ID 0 - GPU_ID 3 - Bus_ID ce
MPI 004 - OMP 000 - HWT 032 - Node crusher132 - RT_GPU_ID 0 - GPU_ID 6 - Bus_ID d9
MPI 005 - OMP 000 - HWT 042 - Node crusher132 - RT_GPU_ID 0 - GPU_ID 7 - Bus_ID de
MPI 006 - OMP 000 - HWT 052 - Node crusher132 - RT_GPU_ID 0 - GPU_ID 0 - Bus_ID c1
MPI 007 - OMP 000 - HWT 056 - Node crusher132 - RT_GPU_ID 0 - GPU_ID 1 - Bus_ID c6

Each rank has access to only 1 GPU, and mapping is ideal.

- MPI rank 0 maps to GPU 4
- MPI rank 1 maps to GPU 5
- Etc.
GPU HW Atomics

GPU HW-enabled atomics can be much faster than SW-enabled atomics.

But importantly, users must explicitly request HW atomics via the -munsafe-fp-atomics flag. Otherwise, atomic operations will be performed via software (i.e., CAS loops).

Whether or not this is safe to use depends on the "granularity" of memory allocated by the user, where “granularity” here refers to the coherency between the memory being used inside a kernel and the memory in the rest of the system.

- With coarse-grained memory, coherence with the rest of the system (CPU and GPUs) is obtained at synchronization points. (e.g., hipDeviceSynchronize, hipMemcpy)

- Fine-grained memory allows CPU and GPU (and multiple GPUs) to synchronize while the GPU kernel is running.

Only coarse-grained memory can use HW atomics. If HW atomics are requested on fine-grained memory, they will (silently) produce a no-op (i.e., give incorrect results).

For more information, see https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html#floating-point-fp-atomic-operations-and-coarse-fine-grained-memory-allocations
Managed Memory – XNACK

Enabled using either zero-copy memory access or page migration on page fault. To understand which will be used, users need to know about XNACK.

XNACK allows AMD GPUs to migrate memory pages (allocated in a unified virtual address space) between the CPU and GPU upon page-fault.

But importantly, support for XNACK must be enabled at both compile-time and run-time.

- A code that is compiled with xnack support and runs in an environment with XNACK enabled (i.e., \texttt{HSA\_XNACK=1}), will migrate pages between the CPU and GPU on page-fault.

- A code that is not compiled with xnack support and does not run in an environment with XNACK enabled, will still function as expected with managed memory, but the CPU will access the GPU’s memory in a zero-copy fashion (CPU reads directly from GPU memory), and vice versa.

XNACK is not enabled by default, so if a user doesn’t realize this, they might not understand why their application (using managed memory) is performing slowly.

For more information, see
https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html#enabling-gpu-page-migration
GPU Register Usage, GPU Profiling, and MI250X L1

Understanding GPU register pressure can help improve performance

• Register usage/spillage can be identified by looking at the ISA (assembly code) generated when using `--save-temps` flag. This typically requires some vendor support to grasp.

• For more info, please see the following slides and recordings from a recent AMD session:
  - https://vimeo.com/742349001

GPU profiling and roofline analysis is always a “hot topic” at these events

• For more information on using rocprof for this, please see: https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html#getting-started-with-the-rocmProfiler

• AMD’s Omniperf and Omnitrace can help simplify this:
  - https://amdresearch.github.io/omniperf/
  - https://amdresearch.github.io/omnitrace/

The L1 cache on AMD’s MI250X (one GCD) is 6X smaller than on NVIDIA’s V100

• L1-cache dependent codes can be bottlenecked by this on MI250X compared to V100.
GPU Initialization Costs and Warp Size

GPU runtime API initialization
- First GPU API call gets bloated by initialization
- Add e.g., `hipFree(0)` at start of program

GPU kernel initialization
- First call to GPU kernel gets bloated by initialization
- Add empty kernel call to start of program
- Add library initialization call at start of program (e.g., `rocblas_initialize()`)

These startup costs are typically very small relative to the overall runtime of a full application.

Warp size on NVIDIA GPUs is 32 but on AMD GPUs it’s 64
- If an application is tuned to a warp size of 32 and run on the MI250X, it will only be getting half the possible occupancy.
Helpful Tips
Slurm Tips – Flags and Completed Jobs

- `u` flag gives unbuffered output, which can be helpful when debugging.
- `l` flag prepends the task ID to lines of stdout.

To show completed jobs in a specific time period, specify a start (-S) and end (-E) time

- The default time window depends on other options (see `man sacct`)

```
$ sacct --user=tpapathe -S 2022-05-05T00:01 -E 2022-05-05T23:59 | awk '$0 !~ /

<table>
<thead>
<tr>
<th>JobID</th>
<th>JobName</th>
<th>Partition</th>
<th>Account</th>
<th>AllocCPUS</th>
<th>State</th>
<th>ExitCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>107814</td>
<td>interact+</td>
<td>batch</td>
<td>stf016</td>
<td>128</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107836</td>
<td>interact+</td>
<td>batch</td>
<td>stf016</td>
<td>128</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107849</td>
<td>hello_job+</td>
<td>batch</td>
<td>stf016</td>
<td>128</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107858</td>
<td>hello_job+</td>
<td>batch</td>
<td>stf016</td>
<td>256</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107866</td>
<td>hello_job+</td>
<td>batch</td>
<td>stf016</td>
<td>256</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107867</td>
<td>hello_job+</td>
<td>batch</td>
<td>stf016</td>
<td>256 CANCELLED+</td>
<td>0:0</td>
<td></td>
</tr>
<tr>
<td>107868</td>
<td>hello_job+</td>
<td>batch</td>
<td>stf016</td>
<td>256</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107965</td>
<td>MatrixTra+</td>
<td>batch</td>
<td>stf016</td>
<td>128</td>
<td>COMPLETED</td>
<td>0:0</td>
</tr>
<tr>
<td>107966</td>
<td>rocprof</td>
<td>batch</td>
<td>stf016</td>
<td>128 FAILED</td>
<td>6:0</td>
<td></td>
</tr>
<tr>
<td>107969</td>
<td>rocprof</td>
<td>batch</td>
<td>stf016</td>
<td>128 FAILED</td>
<td>6:0</td>
<td></td>
</tr>
</tbody>
</table>
```

Then you can drill into more details about each job with the `-j` flag and customized output.
Slurm Tips – Test CPU/GPU Binding Before Running

[crusher: ~]$ module load rocm

[crusher: ~]$ OMP_NUM_THREADS=1 srun -l -N1 -n8 -c8 --gpus-per-node=8 --gpu-bind=closest /bin/bash -c 'echo $(hostname) $(grep Cpus_allowed_list /proc/self/status) GPUS: $ROCR_VISIBLE_DEVICES' | sort

0: crusher076 Cpus_allowed_list: 0-7 GPUS: 4
1: crusher076 Cpus_allowed_list: 8-15 GPUS: 5
2: crusher076 Cpus_allowed_list: 16-23 GPUS: 2
3: crusher076 Cpus_allowed_list: 24-31 GPUS: 3
4: crusher076 Cpus_allowed_list: 32-39 GPUS: 6
5: crusher076 Cpus_allowed_list: 40-47 GPUS: 7
6: crusher076 Cpus_allowed_list: 48-55 GPUS: 0
7: crusher076 Cpus_allowed_list: 56-63 GPUS: 1

This command gives some broad details about the CPUs and GPUs your job step’s processes have access to.
$ cat submit.sh
#!/bin/bash

#SBATCH -A stf016
#SBATCH -J job-name
#SBATCH -o %x-%j.out
#SBATCH -t 5
#SBATCH -p batch
#SBATCH -N 1

scontrol show job ${SLURM_JOBID}
srun -n1 ./p2p/p2p --correct
sacct -j ${SLURM_JOBID} -o jobid%20,Start%20,elapsed%20

Also add...

module -t list
(lists currently-loaded environment modules)

ldd ./<your-executable>
(print shared object dependencies)

$ cat job-name-108121.out

JobId=108121 JobName=job-name

JobId=108121

UserId=tpapathe(5987) GroupId=tpapathe(8654) MCS_label=N/A
Priority=200 Nice=0 Account=stf016 QOS=normal

RunTime=00:00:00 TimeLimit=00:05:00 TimeMin=N/A
SubmitTime=2022-05-05T17:34:51 EligibleTime=2022-05-05T17:34:51
AccrueTime=2022-05-05T17:34:51
StartTime=2022-05-05T17:35:03 EndTime=2022-05-05T17:40:03 Deadline=N/A
SuspendTime=None SecsPreSuspend=0 LastSchedEval=2022-05-05T17:35:03 Scheduler=Backfill
Partition=batch AllocNode:Sid=login2:106343

BatchFlag=1 Reboot=0

ExitCode=0:0
RunTime=00:00:00
TimeLimit=00:05:00
TimeMin=N/A
SubmitTime=2022-05-05T17:34:51

...<APPLICATION OUTPUT>...
Submitting Helpful User Support Tickets

Typical questions we ask users for:

• Job ID
• List of modules loaded when compiling and running
• Batch script used to launch the job
• Job stdout and stderr
• Output from `ldd` run on executable

Notice these are all things that are included from the previous slide 😊

Other helpful things to include in your tickets

• Full errors that are generated
• Has the program run successfully before?
  - If so, did you change anything since then?
• Programming models used in your code

Submit support tickets to help@olcf.ornl.gov
### Debugging Issues on a Compute Node

Once on a compute node, you can use `rocm-smi`, `gdb`, `gstack`, etc.

Get process IDs from `ps -ef | grep <username>`, `top`, etc.

```bash
$ salloc -A stf016 -t 60 -N 1
salloc: Pending job allocation 108355
salloc: job 108355 queued and waiting for resources
salloc: job 108355 has been allocated resources
salloc: Granted job allocation 108355
$ make
Current system: crusher
hipcc --amdgpu-target=gfx90a -c p2p.cpp
hipcc --amdgpu-target=gfx90a p2p.o -o p2p
$ srun -n8 -c8 --gpus-per-node=8 --gpu-bind=closest ./p2p --correct
... ...

$ queue | grep tpapathe
108355 batch interact tpapathe R 13:58 1 crusher001
$ ssh crusher001
$ hostname
crusher001
$ rocm-smi --showpids
------------------------- ROCm System Management Interface -------------------------
------------------------ KFD Processes ------------------------
KFD process information:
PID PROCESS NAME GPU(s) VRAM USED SDMA USED CU OCCUPANCY
65592 p2p 8 2197880832 0 0

$ rocm-smi --showmemuse
---------------------------- ROCm System Management Interface ---------------------------
------------------- Current Memory Use -------------------
GPU[0] : GPU memory use (%) : 1
GPU[0] : Memory Activity : 363340274
GPU[1] : GPU memory use (%) : 2
GPU[1] : Memory Activity : 363699843
GPU[2] : GPU memory use (%) : 0
GPU[3] : GPU memory use (%) : 4
GPU[4] : GPU memory use (%) : 0
GPU[5] : GPU memory use (%) : 0
GPU[6] : GPU memory use (%) : 0
GPU[7] : GPU memory use (%) : 0
GPU[7] : Memory Activity : 361753816
```

Terminal 1

Terminal 2
rocprof Tips

Making sure you actually ran on a GPU:

$ srung ... rocprof --stats ... ./a.out

rocprof output with MPI:

Creates 1 output file per process

$ cat rocprof_winner.sh
#!/bin/bash
rocprof --stats --hip-trace -o my_output_${SLURM_PROCID}.csv "$@

$ srung ... ./rocprof_winner.sh ./a.out

Creates output file from specific process only

$ cat rocprof_winner_single.sh
#!/bin/bash
if [ ${SLURM_PROCID} -eq 0 ];then
  rocprof --stats --hip-trace -o my_output_${SLURM_PROCID}.csv "$@
else
  "$@
fi

$ srung ... ./rocprof_winner_single.sh ./a.out

Viewing trace files
- chrome://tracing
- https://ui.perfetto.dev/

rocprof output with MPI:

Making sure you actually ran on a GPU:

$ srung ... rocprof --stats ... ./a.out

rocprof output with MPI:

Creates 1 output file per process

$ cat rocprof_winner.sh
#!/bin/bash
rocprof --stats --hip-trace -o my_output_${SLURM_PROCID}.csv "$@

$ srung ... ./rocprof_winner.sh ./a.out

Creates output file from specific process only

$ cat rocprof_winner_single.sh
#!/bin/bash
if [ ${SLURM_PROCID} -eq 0 ];then
  rocprof --stats --hip-trace -o my_output_${SLURM_PROCID}.csv "$@
else
  "$@
fi

$ srung ... ./rocprof_winner_single.sh ./a.out

Viewing trace files
- chrome://tracing
- https://ui.perfetto.dev/
Another Way to Check if GPU is Being Used

Making sure you actually ran on a GPU (Cray OpenMP Offload only!)

$ CRAY_ACC_DEBUG=1 srun -A stf016 -t5 -N1 -n1 -c8 --gpus-per-node=8 --gpu-bind=closest ./vAdd_ompGPU

srun: job 108498 queued and waiting for resources
srun: job 108498 has been allocated resources

ACC: Transfer 3 items (to acc 805306368 bytes, to host 0 bytes) from vAdd_ompGPU.cpp:30
ACC: Execute kernel __omp_offloading_69_158925e9_main_130_cce$noloop$form from vAdd_ompGPU.cpp:30
ACC: Transfer 3 items (to acc 0 bytes, to host 268435456 bytes) from vAdd_ompGPU.cpp:30

Test passed.
Result = 1.0000000000000000
Tolerance = 0.0000000000000100
Array buffer size = 268435456
Elapsed time (s) = 0.915606
Some Final Thoughts
Helpful Resources

Crusher Quick-Start Guide:
https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html

- Known Issues: https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html#known-issues
- Profiling Applications: https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html#profiling-applications

CUDA APIs supported by HIP and HIP libraries:
https://github.com/ROCm-Developer-Tools/HIPIFY#cuda-apis

OLCF Training Archive: https://docs.olcf.ornl.gov/training/training_archive.html

```
$ man intro
intro_asm_intrin intro_craypat intro_intrinsics intro_openacc intro_pxf
intro_blacs intro_craype-api intro_io intro_openmp intro_quad
intro_blas1 intro_directives intro_irt intro_papi intro_quad_precision
intro_blas2 intro_dsmml intro_lapack intro_FAPI intro_scalapack
intro_blas3 intro_ffio intro_lapacke intro_perftools intro_specialibs
intro_cblas introhugepages intro_libm intro_pgas intro_timing
intro_coarray intro_ieee intro_libsci intro_pmi intro_upc
intro_conversion intro_intrin intro_mpi intro_pragmas
```
Summary

• Expected performance is application-specific.

• Additional optimizations likely needed to realize expected performance.

• Need to be aware of system-specific considerations.
  
  o Slurm bindings, unified memory, GPU HW atomics, L1 cache

• Profile applications/software to understand where performance stands relative to expected.

• GPU register usage might need to be investigated for optimal performance.

And, of course...

You’re running on some of the most advanced supercomputing hardware in the world!

...so have fun! 😊
Questions?

Summit here

Frontier here

papatheodore@ornl.gov
Extra Slides
### Clang, Clang Wrappers, and Cray’s Compiler Wrappers

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Programming Environment</th>
<th>Compiler Module</th>
<th>Language</th>
<th>Compiler Wrapper</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray</td>
<td>PrgEnv-cr Clay</td>
<td>cce</td>
<td>C</td>
<td>cc</td>
<td>craycc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C++</td>
<td>cc</td>
<td>craycc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fortran</td>
<td>fttn</td>
<td>crayftn</td>
</tr>
<tr>
<td>AMD</td>
<td>PrgEnv-amd</td>
<td>rocm</td>
<td>C</td>
<td>cc</td>
<td>amdclang</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C++</td>
<td>cc</td>
<td>amdclang++</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fortran</td>
<td>fttn</td>
<td>amdflang</td>
</tr>
<tr>
<td>GCC</td>
<td>PrgEnv-gnu</td>
<td>gcc</td>
<td>C</td>
<td>cc</td>
<td>${GCC_PATH}/bin/gcc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C++</td>
<td>cc</td>
<td>${GCC_PATH}/bin/g++</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fortran</td>
<td>fttn</td>
<td>${GCC_PATH}/bin/gfortran</td>
</tr>
</tbody>
</table>

**NOTE:** It is highly recommended to use the Cray compiler wrappers whenever possible, but if for some reason you don’t want to use the wrappers...

- Cray compilers, use **craycc, crayCC**, and **crayftn** (don’t use Cray’s bare Clang compilers)
- AMD compilers, use **amdclang, amdclang++**, and **amdflang** (don’t use AMD’s bare Clang compilers)