Frontier Tips and Tricks

Balint Joo – OLCF

Oak Ridge Leadership Computing Facility
Frontier Training Workshop (virtual)

Friday, Feb 17, 2023

joob AT ornl.gov
Contents

• The joy of being the last but one talk, is that nearly everything has been said already 😊

• That having been said, here are the contents:
  – ROCm Building Tips
  – Interactive/Testing
  – SLURM Process binding and NIC Binding
  – Using the NVMEs
  – CMake Tips
  – Debugging
  – Profiling
  – Getting Help

• We may not get through it all, but you will be able to get the slides for reference.
Modules for ROCm/HIP development: The Old Way

- To use hipcc and other rocmt tools: module load rocmt/<version>

- To use a newer version of CMake: module load cmake

- To use a GPU aware MPI:
  - module load craype-accel-amd-gfx90a
  - export MPICH_GPU_SUPPORT_ENABLED=1

- To link against MPI:
  - module load cray-mpich
  - export MPICH_DIR=/opt/cray/pe/mpich/<version>/ofi/<PE>/<version>
  - export GTL_ROOT=/opt/cray/pe/mpich/<version>/gtl/lib
  - MPI_CFLAGS="${CRAY_XPMEM_INCLUDE_OPTS} -I${MPICH_DIR}/include"
  - MPI_LDFLAGS="${CRAY_XPMEM_POST_LINK_OPTS} -lxpmem -L${MPICH_DIR}/lib -lmpi -L$(GTL_ROOT) -lmpi_gtl_hsa"

- Command line:
  - hipcc ${MPI_CFLAGS} → app app.cpp ${MPI_LDFLAGS}

- Cmake builds (using HIPCC as C++ compiler): set CMake variables as (using -D on command line or in GUI)
  - CMAKE_CXX_COMPILER=hipcc and/or CMAKE_C_COMPILER=hipcc
  - CMAKE_CXX_FLAGS="${MPI_CFLAGS}" and/or CMAKE_C_FLAGS="${MPI_CFLAGS}"
  - CMAKE_EXE_LINKER_FLAGS="${MPI_LDFLAGS}"
  - if using shared libs CMAKE_SHARED_LINKER_FLAGS="${MPI_LDFLAGS}"
New Way: with Cray CC wrappers and AMD Compilers

• Main benefit: fewer explicit flags needed
• Can still use `hipcc` directly
• Incompatible with the ‘roc`m’ module (use one or the other)
  – `module unload PrgEnv-cray`
  – `module load PrgEnv-amd`
  – `module load amd/<ROCm version>`  # e.g. 4.5.2 or 5.1.0
    • this sets the ROCm settings too
• Now you can use ‘CC’ wrappers to compile
  – Handy if you want to use e.g. perftools / CrayPAT
  – No need for XPMEM flags (they are automatic)
  – May load stuff you don’t want: e.g. cray-libsci
    • you may need to explicitly unload this depending on your use-cases.
ROCm versions and MPI

- CC & HIPCC use LLVM underneath
  - Good idea to use MPI compiled with compatible version of LLVM as the CC/hipcc you are using.

- For ROCm-4.5.2 (oldest installed)
  - Use up to MPICH version 8.1.14
  - For hipcc builds MPI_LFDLAGS uses
    - /opt/cray/pe/mpich/8.1.14/ofi/amd/4.4

- For ROCm-5.x
  - Use MPICH version above 8.1.16 --- current is 8.1.23
  - For hipcc builds MPI_LFDLAGS uses
    - /opt/cray/pe/mpich/8.1.23/ofi/amd/5.0
Running Interactive Single node, Single Device jobs

- Frequently used when you may want to profile, check things, or profile and debug single device code

- Easiest without MPI for single device testing:

  ```
  # Sample session
  $ salloc -A <Account> -t hh:mm:ss -N 1 --exclusive
  # GPUs not visible yet at this point
  $
  $ srun -pty bash
  $
  # GPUs now visible. Run as if you were on a workstation
  $
  $ ./gpu_code <user_args>
  ```

- Login nodes also have a GPU for quick testing (no need to `salloc`)
  - Do not use the login node GPUs for heavy compute.
Frontier and Crusher Nodes

- Each node has
  - 1x 64 core AMD HPC Optimized EPYC CPU + 512 GB DDR4 memory
  - 4 x AMD Radeon Instinct MI250X GPUs (gfx90a)
    - Each GPU is made up of 2 Graphics Compute Dies (GCDs)
    - Each GCD has 64 GB HBM (1.6 TB/sec)
    - GPU-GPU: All-to-all Infinity Fabric Interconnect, Host-GPU: PCIe Gen4: 36+36 GB/sec
  - 2 x NVMe SSDs (1.9 TB each)

- Slingshot Interconnect: 25 + 25 GB/sec

- Crusher Documentation: https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html
Binding MPI ranks to GPUs, Cores & NUMA

- I really only need 1 thread per core so in my case I can use:

  
  `srun -n <#MPI> -N <#Nodes> --ntasks-per-node=8 --cpus-per-task=8 \  
  --cpu-bind=map_cpu:48,56,16,24,1,8,32,40 \  
  --mem-bind=map_mem:3,3,1,1,0,0,2,2 \  
  <Application> <args>`

  
  # 7 cpus-per-task for Low Noise Mode
  # Stay off Core 0 for Low Noise Mode
Binding MPI ranks to GPUs, Cores & NUMA (cont’d)

• If you want more threads per MPI use –cpu-bind=mask_cpu

• In the (128-bit) CPU mask, each bit corresponds to a core
  – e.g. core 0 ⇔ bit 0, core 1 ⇔ bit 1 and so forth
  – bits 0-63 are main CPU threads, 64-128 are hyper-threads

```
MASK_0="0x00fe000000000000"       # Cores 49-55
MASK_1="0xfe00000000000000"       # Cores 57-64
MASK_2="0x00000000fe000000"       # Cores 17-23
MASK_3="0x00000000fe000000"       # Cores 17-23
MASK_4="0x00000000fe000000"       # Cores 25-31
MASK_5="0x00000000fe000000"       # Cores 25-31
MASK_6="0x0000000000fe0000"       # Cores 33-39
MASK_7="0x0000000000fe0000"       # Cores 33-39

CPU_MASK= \n"--cpu-bind=mask_cpu:${MASK_0},${MASK_1},${MASK_2},${MASK_3},${MASK_4},${MASK_5},${MASK_6},${MASK_7}"

srun -N 1 -n 8 --ntasks-per-node=8 -c 7 ${CPU_MASK} --mem-bind=map_mem:3,3,1,1,0,0,2,2 \ <Application> <Arguments>
```

Use Tom’s hello_jobstep code to see the effect of your mappings
https://code.ornl.gov/olcf/hello_jobstep
Other useful options

• --gpu-bind=closest
  – if you only need each MPI rank to see only 1 GPU (all comms via MPI rather than via P2P). Then you don’t need a cpu-mask. Each process irrespective of which cores it lands on will be mapped correctly (keep –c 7 flag to give each process a full 7 core ‘width’ – together with the hidden core 0 it will fill an L2 region)

• -S 0
  – Make available all the cores (8 per L2 region)
    • NB: Low noise mode is still on. System functions are still on core 0
Comms optimization. (see also Tim Mattox’s talk!)

- **MPI Awareness**
  - module load craype-accel-gfx90a
  - export MPICH_GPU_SUPPORT_ENABLED=1

- **Place MPI buffers in GPU memory**
  - GPU has direct access to NIC

- **Make a process is bound to the right GPU and NIC**
  - MPICH_OFI_NIC_POLICY=NUMA.
    - map process to NIC nearest process’s NUMA domain
  - MPICH_OFI_NIC_POLICY=GPU
    - map process to NIC nearest process’s attached GPU
  - MPICH_OFI_NIC_POLICY=USER
    - Plus: MPICH_OFI_NIC_MAPPING=<nic>:<local process_ids>; <nic>:<local process ids>...
For Control Freaks only: Fully user bound

Assumption: App explicitly binds via API calls to GPU whose ID is its local MPI Rank

MASK_0="0x00fe000000000000" # Cores 49-55
MASK_1="0xfe00000000000000" # Cores 57-64
MASK_2="0x00000000fe000000" # Cores 17-23
MASK_3="0x000000000000fe00" # Cores 25-31
MASK_4="0x00000000000000fe" # Cores 1-7
MASK_5="0x0000000000000000fe00" # Cores 9-15
MASK_6="0x00000000fe000000000" # Cores 33-39
MASK_7="0x00000000000000fe00000000" # Cores 41-47

CPU_MASK=
"--cpu-bind=mask_cpu:${MASK_0},${MASK_1},
${MASK_2},${MASK_3},${MASK_4},${MASK_5},
${MASK_6},${MASK_7}"

export MPICH_OFI_NIC_POLICY=USER
export MPICH_OFI_NIC_MAPPING="0:0-1; 1:2-3; 2:4-5; 3:6-7"
srun -N 1 -n 8 --ntasks-per-node=8 -c 7 ${CPU_MASK} \
--mem-bind=map_mem:3,3,1,1,0,0,2,2 <Application> <Arguments>

Perspective: the above mapping gave me the same performance as if I had used MPI_OFI_NIC_POLICY=NUMA and not bothered with MPICH_OFI_NIC_MAPPING
Other useful tidbits

• Diagnostics:
  – export MPICH_ENV_DISPLAY=1
  – export MPICH_OFI_NIC_VERBOSE=1

• Synchronizing Collectives
  – Some codes occasionally assume that certain collectives are synchronizing, whereas optimizations may end up meaning they are actually not.
    • This can lead to e.g. hangs
  – One can disable collective optimizations (perform a barrier before the collective) either globally or for individual collectives:
    • export MPICH_COLL_SYNC=0  # Don’t sync before collectives (default)
    • export MPICH_COLL_SYNC=1  # Sync before every collective
    • export MPICH_COLL_SYNC= MPI_Bcast  # Sync before every broadcast
    • man MPI to see list of collectives that are appropriate here.
Use the NVMEs (Chris Zimmer’s talk)

- Each node has 2x 1.92TB NVME units
- To use in scripts: `#SBATCH –C nvme`
- On the command line: `salloc –C nvme`
- NVME directory: `/mnt/bb/${USER}`
- Important: This is a separate directory on each node
  - One node cannot read another node’s directory.
  - Use SLURM variables to avoid name collisions between processes on the same node
  - Make it look like a single FS with UnifyFS
- Directories go away after job ends
  - Make sure your launcher script saves anything you want before exiting.

launch.sh: Each process gets its own dir based on its local ID on the node

```bash
#!/bin/bash

U=${USER}
JOB=${SLURM_JOBID}
LOC=${SLURM_LOCALID}

# make a dir in NVME
DIR=/mnt/bb/${U}/${JOB}_${LOC}

if [ ! -d ${DIR} ];
  then
    mkdir -p ${DIR}
fi

#Run app with args
$* --my-dir=${DIR}
```
HIP and CMake v1

• 2 Ways to go:
  – use `hipcc` or `CC` as the CXX compiler and add extra flags for HIP
  – Use HIP Native Language support

• This version here uses ‘hipcc’ as CXX compiler

• Use find_package() for finding HIP libs
HIP and CMake v2

• Native HIP Language support:
  – mark files as being HIP using
    set_source_files_properties()

• Control compiler via
  – CMAKE_HIP_COMPILER
  – CMAKE_HIP_FLAGS
  – CMAKE_HIP_ARCHITECTURE

• Cannot use hipcc wrapper for CMAKE_HIP_COMPILER

• CMake will look for ROCm clang++ and add flags

• Doesn’t currently work with HIP on NVIDIA

• I still find setting the architecture confusing: GPU_TARGETS? HIP_ARCHITECTURES? etc.

• May take a while to stabilize

# Get ROCm CMake Helpers onto your CMake Module Path
enable_language(HIP)

if (NOT DEFINED ROCM_PATH )
  if (NOT DEFINED ENV{ROCM_PATH} )
    set(ROCM_PATH "/opt/rocm" CACHE PATH "ROCm path")
  else()
    set(ROCM_PATH $ENV{ROCM_PATH} CACHE PATH "ROCm path")
  endif()
endif()

set(CMAKE_MODULE_PATH "/${ROCM_PATH}/lib/cmake" 
  ${CMAKE_MODULE_PATH})

find_package(HIP REQUIRED)
find_package(hipfft REQUIRED)
find_package(hiprand REQUIRED)
find_package(rocrand REQUIRED)
find_package(hipblas REQUIRED)
find_package(rocsolver REQUIRED)
find_package(hipcub REQUIRED)
find_package(rocprim REQUIRED)

set( MY_HIP_SRCS my_hip_src1.cpp my_hip_src2.cpp my_hip_src3.cpp)

# Mark source files as HIP. I guess in the future just a
# LANGUAGE HIP property will suffice. For now do it via compile flags
set_source_files_properties( ${MY_HIP_SRCS} PROPERTIES LANGUAGE HIP)

# Create a Library dependent on HIP
add_library( myLib ${MY_HIP_SRCS} )
target_link_libraries(myLib PUBLIC
  hip::hiprand roc::rocrand
  hip::hipfft
  roc::hipblas roc::rocsolver
  hip::hipcub roc::rocprim_hip )
Debug Tips

• Run with `ulimit -c unlimited` - make sure crashes dump core
  - rocgdb <executable> -c <core file> -- may get you a useful backtrace

• In a hang:
  - squeue | grep <username> to get a list of hosts
  - ssh into any of the hosts
  - run top, or `ps` to find the PID of the executable
  - rocgdb -p <PID>
    • connect to running/hung process. Generate backtrace

• See Mark Stock’s excellent talk
GDB4HPC quick start

- module load gdb4hpc
- module load rocm
- module unload xalt  # xalt can inhibit launches
- salloc ....
- gdb4hpc
- dbg all> maint set unsafe on
  - Sometimes required if MPI Init etc. not found in the code
- launch $a\{N\} --launcher-args="--ntasks-per-node=1 -c 7 --gpus-per-node=8" application -- "application args"
  - N = number of processes
  - --launcher-args specifies ‘srun’ arguments
  - A bunch of startup messages will follow
- a\{0..1\}: Initial breakpoint, in main
  - # Set breakpoints etc and continue
- srun all> c
- See the Profiling and Debugging Tutorial by the HPE COE folks..
ROCProf – The AMD Profiler and Tracer

- rocprof measures a variety of counters and can trace execution

- There are ‘basic counters’ and ‘derived counters’
  - rocprof --list-basic
  - rocprof --list-derived

- Useful to know your code limiters to guide what to measure
  - e.g. Lattice QCD Wilson Dslash (my all time fave kernel / Nemesis )
  - Memory Bandwidth bound ( Flops/Byte $\in [~0.87 \text{ – } ~2.7 ~]$ )
  - High register usage: minimally around 70 registers needed/kernel
    - Spilling is a possibility
Measuring Memory Bandwidth

- Derived Counters: FetchSize, WriteSize
- In single device interactive job invoke as:

  rocprof -i ./mem_counters.txt --timestamp on -o ./dslash_test.csv
  ./dslash_test --dim 16 16 16 16 --niter 10

- 2 lines in input -> executable will run twice
- Profiling may affect performance
View CSV e.g. in Excel.

<table>
<thead>
<tr>
<th>Shared Mem (LDS)=5632</th>
<th>VGPRs=64</th>
<th>Fetch=22.8 MiB</th>
<th>Call Time=~370-390 µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scratch=276</td>
<td>SGPR=112</td>
<td>Write=3 MiB</td>
<td>Kernel Time=53-56 µs</td>
</tr>
</tbody>
</table>
Comments

• Name Mangling: `llvm-cxxfilt` (supplied with ROCM) is your friend

```bash
[bjoo@login1.spock test]$ llvm-cxxfilt _ZN6Kokkos12Experimental4ImplL32hip_parallel_launch_local_memoryINS_4Impl11ParallelForINS3_16ViewValueFunctorINS0_3HIPEjLb1EEENS_11RangePolicyIJS6_NS_9IndexTypeI1EEEES6_EELj1024ELj1EEEVPKT_
```

```cpp
void Kokkos::Experimental::Impl::hip_parallel_launch_local_memory<Kokkos::Impl::ParallelFor<Kokkos::Impl::ViewValueFunctor<Kokkos::Experimental::HIP, unsigned int, true>, Kokkos::RangePolicy<Kokkos::Experimental::HIP, Kokkos::IndexType<long> >, Kokkos::Experimental::HIP>, 1024u, 1u>(Kokkos::Impl::ParallelFor<Kokkos::Impl::ViewValueFunctor<Kokkos::Experimental::HIP, unsigned int, true>, Kokkos::RangePolicy<Kokkos::Experimental::HIP, Kokkos::IndexType<long> >, Kokkos::Experimental::HIP> const*)
```

• CompleteNs – DispatchNs ~ call time
• EndNs – BeginNs – kernel run time << call time here -> latency !!
• Actual BW ~ 26 MiB/55 us ~ 461 GiB/s (End – Begin)
• Observed BW ~ 26MiB/380 us ~ 66.8 GiB/s (CompNs-DispatchNs) ?
• Some ‘Scratch’ is used. Are we spilling registers?
Rocprof and Tracing

• To Trace HIP, HSA and GPU execution use
  
  rocprof --sys-trace
  
  ./dslash_test --dim 16 16 16 16 --niter 10

• Generates JSON file to use with ‘Chrome’ Trace viewer

• Default name: results.json

• You can view with a trace viewer.
  – Type ‘chrome://tracing’ in your chrome URL location
  – Or use your favorite Chrome-Trace compatible tracer tool
  – Getting used to navigating the traces in Chrome can take some time.
  – Also one can use the Perfetto UI trace viewer https://ui.perfetto.dev/
Last Kernel: DurationNs => 35840 ns
⇒ Bandwidth ~ 709 GiB/s
⇒ different profiling methods have different overheads...
Generating ISA files

• Compile with
  - `g --ggdb --save-temps`

• This will save LLVM bytecode, GPU assembly and object files:
  - `test_kokkos_perf`
  - `test_kokkos_perf-hip-amdgcn-amd-amdhsa-gfx90a.s` <- assembly
  - `test_kokkos_perf-hip-amdgcn-amd-amdhsa-gfx90a.o` <- object

• Assembly can be immediately looked at

• Dump object files with `llvm-objdump` e.g.:
  - `llvm-objdump --source --line-numbers ./test_kokkos_perf-hip-amdgcn-amd-amdhsa-gfx90a.o > ISA.dump`
Useful Info in Assembly files

• In the .s files look for function begin and end points:
  – .Lfunc_beginXXX – identify kernel
  – .Lfunc_end – useful into

```
.text
.globl
_ZNKokkos12Experimental4ImplL32hip_parallel_launch_local_memoryINS_4Impl11ParallelForIN2MG13DslashFunctorINS_7complexIfEES8_S8_Li1ELi0EEENS_11RangePolicyIJNS0_3HIPENS_12LaunchBoundsILj256ELj1EEEEEESB_EEIlj256ELj1EEEevPKT_; -- Begin function
_ZNKokkos12Experimental4ImplL32hip_parallel_launch_local_memoryINS_4Impl11ParallelForIN2MG13DslashFunctorINS_7complexIfEES8_S8_Li1ELi0EEENS_11RangePolicyIJNS0_3HIPENS_12LaunchBoundsILj256ELj1EEEEEESB_EEIlj256ELj1EEEevPKT_.p2align 8 .type
_ZNKokkos12Experimental4ImplL32hip_parallel_launch_local_memoryINS_4Impl11ParallelForIN2MG13DslashFunctorINS_7complexIfEES8_S8_Li1ELi0EEENS_11RangePolicyIJNS0_3HIPENS_12LaunchBoundsILj256ELj1EEEEEESB_EEIlj256ELj1EEEevPKT_,@function
_ZNKokkos12Experimental4ImplL32hip_parallel_launch_local_memoryINS_4Impl11ParallelForIN2MG13DslashFunctorINS_7complexIfEES8_S8_Li1ELi0EEENS_11RangePolicyIJNS0_3HIPENS_12LaunchBoundsILj256ELj1EEEEEESB_EEIlj256ELj1EEEevPKT_;
@_ZNKokkos12Experimental4ImplL32hip_parallel_launch_local_memoryINS_4Impl11ParallelForIN2MG13DslashFunctorINS_7complexIfEES8_S8_Li1ELi0EEENS_11RangePolicyIJNS0_3HIPENS_12LaunchBoundsILj256ELj1EEEEEESB_EEIlj256ELj1EEEevPKT_
.Lfunc_begin12:
```

Mangled name: use llvm-cxxfilt to unmangle

entry point
Useful Info in Assembly files

• In the .s files look for function begin and end points:
  – .Lfunc_beginXXX – identify kernel
  – .Lfunc_end – useful into

  .Lfunc_end12:
  ; -- End function
  ... — I REMOVED STUFF To save space....
  .section .AMDGPU.csdata
  ; Kernel info:
  ; codeLenInByte = 10640
  ; NumSgprs: 13
  ; NumVgprs: 108
  ; NumAgprs: 0
  ; TotalNumVgprs: 108  Useful info about GPR’s
  ; ScratchSize: 0  NumAgprs + Scratch Space = 0 means no spills.
  ; MemoryBound: 0
  ; ...

• .s files also give hints about spills. Search for “Folded Spill”
New tool: **OmniPerf**

- OmniPerf from AMD Research can provide
  - CLI collection similar in style to Nsight Compute
  - separate visualization
  - roofline analysis
  - text summaries (including from CLI)
  - One can also still look at CSV files in excel if one is feeling masochistic
  - See Alessandro’s Great Talk!

<table>
<thead>
<tr>
<th>Metric</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid Size</td>
<td>320.00</td>
<td>128.00</td>
<td>512.00</td>
<td>Work Items</td>
</tr>
<tr>
<td>Workgroup Size</td>
<td>192.00</td>
<td>128.00</td>
<td>256.00</td>
<td>Work Items</td>
</tr>
<tr>
<td>Total Wavefronts</td>
<td>87.00</td>
<td>2.00</td>
<td>172.00</td>
<td>Wavefronts</td>
</tr>
<tr>
<td>Saved Wavefronts</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>Wavefronts</td>
</tr>
<tr>
<td>Restored Wavefronts</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>Wavefronts</td>
</tr>
<tr>
<td>VGPRs</td>
<td>28.00</td>
<td>28.00</td>
<td>28.00</td>
<td>Registers</td>
</tr>
<tr>
<td>GPRs</td>
<td>80.00</td>
<td>80.00</td>
<td>80.00</td>
<td>Registers</td>
</tr>
<tr>
<td>LDS Allocation</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>Bytes</td>
</tr>
<tr>
<td>Scratch Allocation</td>
<td>496.00</td>
<td>496.00</td>
<td>496.00</td>
<td>Bytes</td>
</tr>
</tbody>
</table>

**Instruction Mix**

**Text summaries**

Profiled kernel is from the HemeLB code from UCL (P. Coveney, I. Zacharoidiou, et. al.)
New Tools: OmniPerf

• Still some multi-MPI issues... I had the following looking at a code I was working with:
  – Process 0 didn’t have any GPU kernels and this broke the counter collation
    • Solution I added a Dummy Kernel (thanks to Vassilios Mewes for the idea)
  – All the MPI tasks needed to run OmniPerf (for the replays to work)

• OmniPerf is mostly Python: [https://github.com/AMDResearch/omniperf](https://github.com/AMDResearch/omniperf)
  – uses e.g. pandas to process ROCprof JSON files
  – visualizations by running a local web server, or providing MongoDB database to a Grafana visualization service (needs setup)
  – One could install on local Linux system to process and visualize files obtained from Crusher
    • I had Python issues on my Mac – may be Mac specific. Your mileage may vary
    • Ended up using ‘local server approach’
Getting Help

• Submit a ticket to help@olcf.ornl.gov

• Consult the documentation at:
  – https://docs.olcf.ornl.gov/systems/frontier_user_guide.html
  – https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html

• Consider attending an “Office Hour”
  – Mondays at 2-3pm
  – Sign up at https://www.olcf.ornl.gov/crusher-office-hours/
Ticket Tips

• The most helpful tickets
  – Clearly state the problem, the key modules/env vars
  – Have a small reproducer (either attached or identified in the text)
  – detail any other investigation you may have undertaken before you got stuck

• Less helpful tickets
  – “Please help! My code stopped working.”

• The least helpful ticket:
  –
Summary

• We discussed
  – modules needed to get developing with HIP on Frontier & Crusher
  – running single device, interactive jobs, for debugging & profiling
  – how to bind processes (both 1-core and multi-core per process)
  – how to use NVME
  – how to set up CMake for building for HIP/ROCm
  – how to generate profiles and traces using the QUDA ‘dslash_test’ as an example (memory b/w bound kernel run in a latency bound region)
  – how to generate ISA, and look for kernel information
  – Looked at some new tools in the pipe (OmniPerf)
  – how to get help

• Questions?
Acknowledgements and Thanks!

• These tidbits here are a disordered collection of information I have gathered from our Frontier Center of Excellence colleagues at AMD especially: Nick Curtis, Damon McDougall and Corbin Robeck

• Our profiling examples used the QUDA Code available from https://github.com/lattice/quda.git which is maintained by Kate Clark and the QUDA community.

• Our ISA example use Kokkos Dslash which uses Kokkos. Big shout out to the Kokkos Team! Locally at ORNL the HIP porting is the hard work of Damien Lebrun-Grandie, Bruno Trucsin, Daniel Arndt and colleagues working closely with Nick Curtis at AMD (https://github.com/kokkos )

• OmniPerf plots were made using the HemeLB code from UCL courtesy of Peter Coveney and Ioannis Zacharoidiou
Funding Acknowledgement

• This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

• This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.