Developing for Frontier using HIP

Balint Joo – OLCF
Oak Ridge Leadership Computing Facility
Crusher User Experience Talks (virtual)
Friday, Dec 9, 2022
joob AT ornl.gov
Outline

• My Focus: HIP and C++ using AMD/ROCm compilers
  – I have recently started using the HPE Wrappers too, mostly to help with profiling and potentially with MPI features like XPMEM.

• Crusher, the TDS System for Frontier
  – brief overview
  – modules
  – simple tests/interactive running

• Tools:
  – CMake and HIP Language mode
  – Rocgdb
  – Rocprof / OmniPerf
My Application: Lattice QCD using Chroma

- A layered C++ Code made up of 3 primary packages:
  - Chroma: physics written in QDP++ and interfaces
  - QUDA: A library for QCD on GPUs
    - K. Clark (NVIDIA) lead developer & maintainer
    - Community developed
    - Supplies our highly optimized Multigrid Solvers
  - QDP-JIT: An implementation of QDP++ using LLVM and JIT compilation for GPUs
    - F. Winter (JLab) lead developer
    - Uses expression templates and LLVM to give us performance portability through LLVM back-ends
- Our choice for Frontier was to use HIP for ROCm
  - most like CUDA which was already used in QUDA.
  - QDP-JIT was a specialized effort for Frank – getting to know the LLVM AMDGCN back end and the HIP and ROCm libraries.
QUDA Porting Story

- We decided not to just ‘hipify’ using the hipify tool
  - we’d have to re-do it every time the main code changed (often)
  - we need the code on other architectures too (e.g. Aurora)

- Vendors and QCD Developers came together
  - Kate Clark, Mathias Wagner from NVIDIA
  - Damon McDougall and Corbin Robeck from AMD
  - Patrick Steinbrecher from Intel
  - Myself from ORNL
  - James Osborn, Xiao-Yong Jin from ALCF
  - Alexei Strelchenko from Fermilab
  - Dean Howarth from LLNL (now LBNL)
QUDA Porting Cont’d

• Our portability working group met weekly (now biweekly)

• Code was refactored
  – Identified QUDA parallel patterns (1D, 2D, 3D kernels, reductions, etc.)
  – Abstracted programming model elements
    • kernel launches
    • data copies between host/device
    • GPU streams
    • shared memory

• Implemented Code in QUDA back-ends
  – HIP, SYCL/DPC++, OpenMP-offload
  – ISO Std. C++ parallelism is a future target
Some QUDA results: Wilson Dslash

QUDA Dslash Test, V=32^4 sites, single device, Wilson (scalar build)

- Half-18
- Half-12
- Half-8
- Single-18
- Single-12
- Single-8
- Double-18
- Double-12
- Double-8

GFLOPS reported

- MI100
- MI250X (1 GCD)
- MI250X (2 GCDs)
- V100
- A100

MI250X numbers from Crusher, using ROCm-5.1, MI100 Numbers from Spock (ROCm-4.5.2), V100 Numbers from Summit, A100 Numbers courtesy of E. Weinberg

Both Crusher and Spock are pre-production systems. All performance results are preliminary and may change with software updates or operational changes.
HMC Trajectory Scaling

- HMC Trajectories as part of the ECP FOM
  - Green Line uses the $64^3 \times 128$ config replicated twice in each direction (16x increase in volume)
  - Green line uses 12 steps per trajectory as opposed to original red-line which used 6.

- We can identify weak scaling counterpart points and see good weak scaling

"Stitched" Configuration courtesy of Boram Yoon (LANL)

NOT PHYSICAL
Both Crusher and Frontier are pre-production systems. All performance results are preliminary and may change with software updates or operational changes.
Crusher: The Test and Development System for Frontier

- The system has 192 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: 32+32 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](https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html)
Modules for ROCm/ HIP development: The Old Way

- To use hipcc and other rocm tools: `module load rocm/<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 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} -o app app.cpp ${MPI_LDFLAGS}`

- Cmake builds if not using Native HIP Language Support: 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 ‘rocsm’ 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.
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>
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>
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.

```
# HEX values in mask: each ‘digit’ is 4 bits here. ‘f’ means all 4 bits
# Masks below use all available cores in each L2 domain corresponding to a GCD
# NB: Masks need to be consistent with --cpus-per-task (-c)
# If we had e.g. --c 7, a mask of 0xff would not work. We would need e.g. 0x7f

MASK_0="0x00ff000000000000"  # Cores 48-55
MASK_1="0xff00000000000000"  # Cores 56-64
MASK_2="0x0000ff0000000000"  # Cores 16-23
MASK_3="0x000000ff00000000"  # Cores 24-31
MASK_4="0x00000000ff000000"  # Cores 0-8
MASK_5="0x0000000000ff0000"  # Cores 8-15
MASK_6="0x000000000000ff00"  # Cores 32-39
MASK_7="0x00000000000000ff"  # Cores 40-47

CPU_MASK="--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 8 ${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
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

```cpp
# Get ROCm CMake Helpers onto your CMake Module Path
if (NOT DEFINED ROCM_PATH )
  if (NOT DEFINED ENV{ROCM_PATH} )
    set(ROCM_PATH "/opt/rocm" CACHE PATH "RoCm path")
  else()
    set(ROCM_PATH $ENV{ROCM_PATH} CACHE PATH "RoCm path")
  endif()
else()
  set(CMAKE_MODULE_PATH "${ROCM_PATH}/lib/cmake" ${CMAKE_MODULE_PATH})
endif()

# Set GPU Targets and Find all the HIP modules
set(GPU_TARGETS "gfx906;gfx908" CACHE STRING "The GPU TARGETs")
find_package(HIP REQUIRED)
find_package(hipfft REQUIRED)
find_package(hiprand REQUIRED)
find_package(rocrand REQUIRED)
find_package(hipblas REQUIRED)
find_package(rocblas 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 CXX)
set_source_files_properties( ${MY_HIP_SRCS} PROPERTIES COMPILE_FLAGS "-x hip")

# Create a Library dependent on HIP
add_library( myLib ${MY_HIP_SRCS} )
target_include_directories(myLib PUBLIC ${ROCM_PATH}/hipfft/include)
target_link_libraries(myLib PUBLIC
  hip::hiprand roc::rocrand
  hip::hipfft
  roc::hipblas roc::rocblas
  hip::hipcub roc::rocprim_hip )
```
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

```cpp
# 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(rocblas 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::rocblas
   hip::hipcub roc::rocprim_hip )
```
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 ∈ [~0.87 – ~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.

Shared Mem (LDS)=5632   VGPRs=64   Fetch=22.8 MiB   Call Time=~370-390 µs

Scratch=276   SGPR=112   Write=3 MiB   Kernel Time=53-56 µs
Comments

• Name Mangling: \texttt{llvm-cxxfilt} (supplied with ROCM) is your friend

\texttt{bjoo@login1.spock test}$ \texttt{llvm-cxxfilt _ZN6Kokkos12Experimental4ImplL32hip_parallel_launch_local_memoryINS_4Impl11ParallelForINS3_16ViewValueFunctorINS0_3HIPEjLb1EEENS_11RangePolicyIJS6_NS_9IndexTypeI1EEEES6_\_EELj1024ELj1EEEvpkt}_

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 \lll call time here -> latency!!

• Actual BW \lll 26 MiB/55 us \lll 461 GiB/s (End – Begin)

• Observed BW \lll 26MiB/380 us \lll 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.
Chrome Trace

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

```assembly
.text
.globl

Z_6Kokkos12Experimental4Impl132hip_parallel_launch_local_memoryINS_4Impl11ParallelForIN2MG13DslashFunctorINS_7complexIfEES8_S8_Li1ELi0EEENS_11RangePolicyIJNS0_3HIPENS_12LaunchBoundsILj256ELj1EEEEEESB_EELj256ELj1EEEevPKT gsi -- Begin function
Z_6Kokkos12Experimental4Impl132hip_parallel_launch_local_memoryINS_4Impl11ParallelForIN2MG13DslashFunctorINS_7complexIfEES8_S8_Li1ELi0EEENS_11RangePolicyIJNS0_3HIPENS_12LaunchBoundsILj256ELj1EEEEEESB_EELj256ELj1EEEevPKT gsi
.p2align
8
.type

Z_6Kokkos12Experimental4Impl132hip_parallel_launch_local_memoryINS_4Impl11ParallelForIN2MG13DslashFunctorINS_7complexIfEES8_S8_Li1ELi0EEENS_11RangePolicyIJNS0_3HIPENS_12LaunchBoundsILj256ELj1EEEEEESB_EELj256ELj1EEEevPKT gsi, @function
Z_6Kokkos12Experimental4Impl132hip_parallel_launch_local_memoryINS_4Impl11ParallelForIN2MG13DslashFunctorINS_7complexIfEES8_S8_Li1ELi0EEENS_11RangePolicyIJNS0_3HIPENS_12LaunchBoundsILj256ELj1EEEEEESB_EELj256ELj1EEEevPKT gsi;
@ Z_6Kokkos12Experimental4Impl132hip_parallel_launch_local_memoryINS_4Impl11ParallelForIN2MG13DslashFunctorINS_7complexIfEES8_S8_Li1ELi0EEENS_11RangePolicyIJNS0_3HIPENS_12LaunchBoundsILj256ELj1EEEEEESB_EELj256ELj1EEEevPKT gsi
.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

---

**Instruction Mix**

<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>SPPRs</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>

*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
  – 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’
Summary

• We discussed
  – modules needed to get developing with hip on Crusher
  – running single device, interactive jobs, for debugging & profiling
  – how to bind processes (both 1-core and multi-core per process))
  – 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)

• Questions?

• And remember it is: not even tho’ I am a Wil Wheaton fan...
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.