Comparative Performance and Porting Effort of HIP and CUDA for an Implicit Monte Carlo Code

2020 Performance, Portability and Productivity in HPC

Coming to you pre-recorded from fabulous Los Alamos, New Mexico

Alex R. Long

9/1/2020

LA-UR-20-26679

Comparative ~~Performance and~~ Porting Effort of HIP and CUDA for an Implicit Monte Carlo Code

- First off, this talk does not address the performance of the HIP porting effort
- Monte Carlo transport is known to consume a large number of CPU cycles and I spend a lot of time thinking about this issue
- Our mandate is to be able to run on new machines, my experience thus far is that porting takes more time than optimizing so plan accordingly

The CUDA port of the Jayenne IMC library performs well without much optimization

- We ported our Monte Carlo transport code with CUDA
- The lion's share of the work was isolating code to run on an accelerator
  - Monte Carlo naturally has a lot of potential code paths
  - Pull difficult code paths out, isolate core work functions
  - After that, removing host code from device functions (std library) took some time
  - Initial performance gains came from using shared memory, eliminating separable compilation and using constant memory for some data
- I heard about HIP as a solution for CUDA codes on El Capitan and was excited to try to qualitatively measure the effort
  - Was I excited for petty reasons? The short answer, is yes, yes I was
  - I made a decision for performance first, could I have my cake and eat it too?

HIP has some attractive features for a “basic” CUDA code

- HIP generates code for AMD devices but is also a “thin layer” over CUDA for a “single-source” GPU solution
- No need to set experimental CUDA flags for constexpr or thrust with lambda functions
- Some features of Jayenne that make HIP a good fit (your mileage may vary)
  - Jayenne uses explicit memory management in CUDA (cudaMalloc, cudaFree) and not managed memory (under development in HIP)
  - No logic in Jayenne assumes a warp size (could be 64 in AMD devices)
  - No virtual functions inside the transport loop (not supported in HIP)

Some details of the port work: the easy parts

- Loaded the ROCM module, it put hipify-perl in my path and I ran
  - hipify-perl –i <file name>
- Kernel launch and adding include files was the only change I would consider more advanced than "find_replace"
  - My thrust include paths and functions remained unchanged
- As many codes do, I hide CUDA specific code behind Cmake configure time macros
- No macros had to change, even the ones for constant texture memory
- I changed "#ifdef __NVCC__" type checks to "__HIPCC__"
- No checks on __CUDA_ARCH__ in Jayenne so no ambiguity between device features

Some details of the port work: the hard parts that are related to how Jayenne works

- I saw a hipify-cmake script was also added to the path by ROCM
  - Main difference is changing the "find_package(CUDA)" command to HIP
  - Our CMake does not use the standard CMake "find_package(CUDA)" type approach, we instead enable CUDA as a language on a per project basis
- I only have three object files to make, let's do it by hand!
  - Remove gcc flags (sanitize)
  - make VERBOSE=1
  - Change compiler in make line to "hipcc"
- Our random number generator, Random123, code make heavy use of platform specific intrinsics
  - Are you shocked that this isn't very portable?
  - Ignoring instrinisic optimizations, even turning on "__device__" decorators caused a problem
  - No portability solution would have solved this problem for me!

Some details of the port work: the parts you expect when using new tools

- All complaints mentioned here are just to point out what does not “just work” in doing a HIP port
  - hipcc doesn’t accept −fopenmp flag in link phase (**fixed in latest**)
  - I ran into a bug in ROCprim with including some thrust functions (**fixed in latest)**
- ROCm uses the system gcc to build glibc, this caused problems in linking to my gcc code
  - This problem is fixed by specifying a gcc toolchain when using hipcc for compiling and linking
- Our code does not currently compile with clang10+ and ROCm 3.5 on our system is clang11

A comparison of porting efforts: C++ to CUDA and CUDA to HIP

- There is still work to do, as I mentioned, I was not able to run performance comparisons
- I successfully ran my simple GPU tests that moves a particle with the core loop
- A table with some rough numbers for the CUDA port based on the git commit history of Jayenne and the CDash dashboard

Port feature	Date
Start code reorganization	06/2017
Separate GPU functions	06/2019
First CUDA kernel call	11/2019
Runs tstRW_Transporter	03/2020
Passes all integration tests with GPU	08/2020

HIP port is here

Conclusions

- Of course, this comparison is not scientific in a number of ways
  - Measuring “effort” is inherently difficult
  - I’m not able to compare performance to CUDA yet
- That said, this port took a week of work
  - How long would it take to move to KOKKOS? I know where my loops are, I know where my memory needed on the device is, my code survives a pass with the NVCC compiler, maybe someone could tell me?
- The most valuable part of this work is knowing where the code is with respect to a HIP port
  - I can start having conversations about our CMake and how we expect HIP to work with it
  - Someone with more experience in compiler intrinsics can look at Random123
- It took me about a week of work to find out “where we are”