Porting Codes to LUMI
Cray User Group
May 5th, 2021

George S. Markomanolis
Lead HPC Scientist, CSC – IT Center for Science Ltd.
Outline

• LUMI
• Approaches to port codes on LUMI
• Benchmarking
• Profiling
• Tuning
LUMI, the Queen of the North

LUMI is a Tier-α GPU-accelerated supercomputer that enables the convergence of high-performance computing, artificial intelligence, and high-performance data analytics.

- Supplementary CPU partition
- ~200,000 AMD EPYC CPU cores

Possibility for combining different resources within a single run. HPE Slingshot technology.

30 PB encrypted object storage (Ceph) for storing, sharing and staging data

Tier-α GPU partition: over 550 Pflop/s powered by AMD Instinct GPUs

Interactive partition with 32 TB of memory and graphics GPUs for data analytics and visualization

7 PB Flash-based storage layer with extreme I/O bandwidth of 2 TB/s and IOPS capability. Cray ClusterStor E1000.

80 PB parallel file system

www.lumi-supercomputer.eu  #lumisupercomputer #lumieurohpc
AMD GPUs (MI100 example)

LUMI will have a different GPU
Differences between HIP and CUDA

- AMD GCN hardware wavefronts size is 64 (like warp for CUDA), some terminology is different
- Some CUDA library functions do not have AMD equivalents
- Shared memory and registers per thread can differ between AMD and NVIDIA hardware
- ROCM 4.1 just released and improves some functionalities (Warp-Level primitives was not supported by HIP but maybe this is improved)
Porting Codes to LUMI

1. **Parallel code with GPU**
   - **Does it have OpenMP?**
     - Yes: Use OpenMP Offloading for GPU
     - No: Try new libraries and re-write parts of the code

2. **Advanced programmer with knowledge on GPUs and enough available resources and time**
   - **Is performance good?**
     - Yes: Enjoy!
     - No: Profile, tune OpenMP calls and data transfers

3. **Profile, identify kernels, port them in HIP and tune**
   - **Is the code in C/C++?**
     - Yes: Fix code, if any, that was not converted to HIP (for C/C++), profile and tune, use hip libraries where possible
     - No: Profile Fortran, port kernels to HIP and tune

4. **CUDA**
   - **Is it C/C++ code?**
     - Yes: Use hipify tools
     - No: Use hipfort and prepare the kernels

5. **OpenACC**
   - **Is it Fortran code?**
     - Yes: Profile and port the OpenACC calls to OpenMP Offloading to GPU
     - No: Profi and try new libraries and re-write parts of the code

6. **Apexak, SYCL, Kokkos, Raja (not all programming languages supported)**
   - Do you want to try new libraries and re-write parts of the code?

* Need to check which programming languages will be supported
Porting Codes to LUMI (experimental)

Porting codes to LUMI

- Parallel code with GPU
  - Does it have OpenMP?
    - Yes: Use OpenMP tools or port through profiling and identify important loops to OpenMP.
    - No: Use novel tools or port through profiling and identify important loops to OpenMP.
  - Performance good?
    - Yes: Enjoy!
    - No: Fix code, if any, that was not converted to HIP (for C++), profile and tune, and use hip libraries where possible.

- Parallel code with GPU
  - OpenACC
    - Is it C/C++ code?
      - Yes: Use hipify tools and prepare the kernels.
      - No: Use OpenACC tools.
    - Is it Fortran code?
      - Yes: Use hipify tools and prepare the kernels.
      - No: Use OpenACC tools.
  - CUDA
    - Is it C/C++ code?
      - Yes: Use hipify tools and prepare the kernels.
      - No: Use CUDA tools.
    - Is it Fortran code?
      - Yes: Use hipify tools and prepare the kernels.
      - No: Use CUDA tools.
  - Alpaka, SYCL, Kokkos, Raja
    - Not all programming languages supported

Do you want to try new libraries and re-write parts of the code?

Advanced programmer with knowledge on GPUs and enough available resources and time

Enjoy!

* Need to check which programming languages will be supported
OpenMP Offload

• There are many tutorials about OpenMP Offloading

• Some basic OpenMP useful constructs:
  o #pragma omp target enter/exit data map
  o #pragma omp target teams distribute parallel for simd
  o thread_limit(X) num_teams(Y)

• There are a lot of tutorials about OpenMP Offloading

• OpenMP 5.0, what is new: https://www.openmp.org/spec-html/5.0/openmpse71.html

• OpenMP 5.1, what is new: https://www.openmp.org/wp-content/uploads/OpenMP-API-Additional-Definitions-2-0.pdf

• OpenMP 5.0 tutorial: https://ecpannualmeeting.com/assets/overview/sessions/ff2020%20ECP-Tutorial-with-ECP-template.pdf
BabelStream (default settings)
Improving performance on BabelStream for MI100

• Original call:

```
#pragma omp target teams distribute parallel for simd
```

• Optimized call

```
#pragma omp target teams distribute parallel for simd thread_limit(256) num_teams(240)
```

• For the dot case we used 720 teams
BabelStream, tune AOMP
Introduction to HIP

• HIP: Heterogeneous Interface for Portability is developed by AMD to program on AMD GPUs
• It is a C++ runtime API and it supports both AMD and NVIDIA platforms
• HIP is similar to CUDA and there is no performance overhead on NVIDIA GPUs
• Many well-known libraries have been ported on HIP
• New projects or porting from CUDA, could be developed directly in HIP

https://github.com/ROCm-Developer-Tools/HIP
Hipify

• Hipify tools convert automatically CUDA codes
• It is possible that not all the code is converted, the remaining needs the implementation of the developer
• Hipify-perl: text-based search and replace
• Hipify-clang: source-to-source translator that uses clang compiler
Hipify-perl

• It can scan directories and converts CUDA codes with replacement of the cuda to hip (sed –e ’s/cuda/hip/g’)

$ hipify-perl --inplace filename

It modifies the filename input inplace, replacing input with hipified output, save backup in .prehip file.

$ hipconvertinplace-perl.sh directory

It converts all the related files that are located inside the directory
Hipify-perl (cont).

1) $ ls src/

   Makefile.am  matMulAB.c  matMulAB.h  matMul.c

2) $ hipconvertinplace-perl.sh src

3) $ ls src/

   Makefile.am  matMulAB.c  matMulAB.c.prehip  matMulAB.h  matMulAB.h.prehip  matMul.c  matMul.c.prehip

No compilation took place, just conversion.
• The hipify-perl will return a report for each file, and it looks like this:

info: TOTAL-converted 53 CUDA->HIP refs ( error:0 init:0 version:0 device:1 ... library:16
... numeric_literal:12 define:0 extern_shared:0 kernel_launch:0 )
warn:0 LOC:888
kernels (0 total):
hipFree 18
HIPBLAS_STATUS_SUCCESS 6
hipSuccess 4
hipMalloc 3
HIPBLAS_OP_N 2
hipDeviceSynchronize 1
hip_runtime 1
## Differences between CUDA and HIP API

<table>
<thead>
<tr>
<th>CUDA</th>
<th>HIP</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &quot;cuda.h&quot;</code></td>
<td><code>#include &quot;hip/hip_runtime.h&quot;</code></td>
</tr>
<tr>
<td><code>cudaMalloc(&amp;d_x, N*sizeof(double));</code></td>
<td><code>hipMalloc(&amp;d_x, N*sizeof(double));</code></td>
</tr>
<tr>
<td><code>cudaMemcpy(d_x,x,N*sizeof(double),</code></td>
<td><code>hipMemcpy(d_x,x,N*sizeof(double),</code></td>
</tr>
<tr>
<td><code>    cudaMemcpyHostToDevice);</code></td>
<td><code>    hipMemcpyHostToDevice);</code></td>
</tr>
<tr>
<td><code>cudaDeviceSynchronize();</code></td>
<td><code>hipDeviceSynchronize();</code></td>
</tr>
</tbody>
</table>
Launching kernel with CUDA and HIP

**CUDA**

```c
kernel_name <<<gridsize, blocksize, shared_mem_size, stream>>>(arg0, arg1, ...);
```

**HIP**

```c
hipLaunchKernelGGL(kernel_name, gridsize, blocksize, shared_mem_size, stream, arg0, arg1, ...);
```
## Libraries (not exhaustive)

<table>
<thead>
<tr>
<th>NVIDIA</th>
<th>HIP</th>
<th>ROCm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuBLAS</td>
<td>hipBLAS</td>
<td>rocBLAS</td>
<td>Basic Linear Algebra Subroutines</td>
</tr>
<tr>
<td>cuRAND</td>
<td>hipRAND</td>
<td>rocRAND</td>
<td>Random Number Generator Library</td>
</tr>
<tr>
<td>cuFFT</td>
<td>hipFFT</td>
<td>rocFFT</td>
<td>Fast Fourier Transform Library</td>
</tr>
<tr>
<td>cuSPARSE</td>
<td>hipSPARSE</td>
<td>rocSPARSE</td>
<td>Sparse BLAS + SPMV</td>
</tr>
<tr>
<td>NCCL</td>
<td>RCCL</td>
<td></td>
<td>Communications Primitives Library based on the MPI equivalents</td>
</tr>
<tr>
<td>CUB</td>
<td>hipCUB</td>
<td>rocPRIM</td>
<td>Low Level Optimized Parallel Primitives</td>
</tr>
</tbody>
</table>
Benchmark MatMul cuBLAS, hipBLAS

• Use the benchmark [https://github.com/pc2/OMP-Offloading](https://github.com/pc2/OMP-Offloading)

• Matrix multiplication of 2048 x 2048, single precision

• All the CUDA calls were converted and it was linked with hipBlas

• CUDA (V100)

matMulAB (10) : 1011.2 GFLOPS  12430.1 GFLOPS

• HIP (MI100)

matMulAB (10) : 2327.6 GFLOPS  22216.7 GFLOPS

• MI100 achieves close to the theoretical peak for single precision
N-BODY SIMULATION

- N-Body Simulation (https://github.com/themathgeek13/N-Body-Simulations-CUDA) AllPairs_N2
- 171 CUDA calls converted to HIP without issues, close to 1000 lines of code
- 32768 number of small particles, 2000 time steps

CUDA execution time on V100 : 68.5 seconds
HIP execution time on MI100: 95.57 seconds, 39.5% worse performance

- Tune the number of threads per block to 256 instead of 1024, then:

HIP execution time on Mi100: 54.32 seconds, 26.1% better performance than V100
Fortran

• First Scenario: Fortran + CUDA C/C++
  o Assuming there is no CUDA code in the Fortran files.
  o Hipify CUDA
  o Compile and link with hipcc

• Second Scenario: CUDA Fortran
  o There is no HIP equivalent
  o HIP functions are callable from C, using `extern C`
  o See hipfort
Hipfort

• The approach to port Fortran codes on AMD GPUs is different, the hipify tool does not support it.
• We need to use hipfort, a Fortran interface library for GPU kernel *
• Steps:
  1) We write the kernels in a new C++ file
  2) Wrap the kernel launch in a C function
  3) Use Fortran 2003 C binding to call the function
  4) Things could change in the future
• Use OpenMP offload to GPUs

* https://github.com/ROCmSoftwarePlatform/hipfort
Fortran CUDA example

- Saxpy example
- Fortran CUDA, 29 lines of code
- Ported to HIP manually, two files of 52 lines, with more than 20 new lines.
- Quite a lot of changes for such a small code.
- Should we try to use OpenMP offload before we try to HIP the code?
- Need to adjust Makefile to compile the multiple files
- Example of Fortran with HIP: https://github.com/cschpc/lumi/tree/main/hipfort
BabelStream on MI100 (HIP vs AOMP)
Megahip

- [https://github.com/zjin-lcf/oneAPI-DirectProgramming](https://github.com/zjin-lcf/oneAPI-DirectProgramming)
- 115 Applications/Examples with CUDA, SYCL, OpenMP offload and HIP
- Testing hipify tool, create a megahip script to convert all the CUDA examples to HIP
- `.megahip.sh`

3287 CUDA calls were converted to HIP

115 applications totally 45692 lines of code, there are warnings for 4 of them, there are totally 24 warnings that something was wrong, check warnings.txt
Application Success 96.5217%
Conversion Success 99.2699%
OpenACC

- GNU will provide OpenACC (Mentor Graphics contract, now called Siemens EDA)
- HPE will use the provided GNU compiler for OpenACC support
- HPE will support for OpenACC v2.0 for Fortran. This is quite old OpenACC version.
- Clacc from ORNL: https://github.com/llvm-doe-org/llvm-project/tree/clacc/master
  OpenACC from LLVM only for C (Fortran and C++ in the future)
  - Translate OpenACC to OpenMP Offload
Clacc

- It supports C programming language, Fortran is on the way, C++ not started(??) yet

$ \text{clang -fopenacc-print=omp -fopenacc-structured-ref-count-omp=no-hold -fopenacc-present-omp=no-present} \text{ jacobi.c}$

Original code:
```c
#pragma acc parallel loop reduction(max:lnorm) private(i,j) present(newarr, oldarr) collapse(2)
for (i = 1; i < nx + 1; i++) {
    for (j = 1; j < ny + 1; j++) {

New code:
```c
#pragma omp target teams map(alloc: newarr,oldarr) map(tofrom: lnorm) shared(newarr,oldarr)
firstprivate(nx,ny,factor) reduction(max: lnorm) \n#pragma omp distribute private(i,j) collapse(2)
for (i = 1; i < nx + 1; i++) {
    for (j = 1; j < ny + 1; j++) {
```
SYCL Implementations in Development

SYCL implementations are available from an increasing number of vendors, including adding support for diverse acceleration API back-ends in addition to OpenCL.

hipSYCL and SYCL 2020: https://github.com/hipSYCL/featuresupport
SAXPY SYCL

cyl::queue q(sycl::default_selector{});

const float A(aval);
sycl::buffer<float,1> d_X { h_X.data(), sycl::range<1>(h_X.size()) };
sycl::buffer<float,1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) };
sycl::buffer<float,1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) };
q.submit([&](sycl::handler& h) {
    auto X = d_X.template get_access<sycl::access::mode::read>(h);
    auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
    auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);
    h.parallel_for<class nstream>(sycl::range<1>{length}, [=] (sycl::id<1> it) {
        const int i = it[0];
        Z[i] = A * X[i] + Y[i];
    });
});
q.wait();
Results of BabelStream on Mi100 (AOMP vs HIP vs hipSYCL)
Profiling/Debugging

- AMD provides APIs for profiling and debugging
- Some simple environment variables such as `AMD_LOG_LEVEL=4` will provide some information.
- More information about a hipMemcpy error:

  ```c
  hipError_t err = hipMemcpy(c,c_d,nBytes,hipMemcpyDeviceToHost);
  printf("\%s ", hipGetErrorString(err));
  ```
- ROCprofiler, ROCgdb
- Some profiling tools work with AMD GPUs
## TAU profiling

<table>
<thead>
<tr>
<th>Name</th>
<th>Exclusive TAU GPU...</th>
<th>Inclusive TAU GPU...</th>
<th>Calls</th>
<th>Child Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>.TAU application</td>
<td>0.31</td>
<td>0.633</td>
<td>1</td>
<td>501</td>
</tr>
<tr>
<td>void add_kernel&lt;double&gt;(double const*, double const*, double*) [clone .kd]</td>
<td>0.66</td>
<td>0.68</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>void copy_kernel&lt;double&gt;(double const*, double*) [clone .kd]</td>
<td>0.052</td>
<td>0.052</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>void dot_kernel&lt;double&gt;(double const*, double const*, double*, int) [clone .kd]</td>
<td>0.059</td>
<td>0.059</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>void init_kernel&lt;double&gt;(double*, double*, double*, double, double, double) [clone .kn]</td>
<td>0.001</td>
<td>0.001</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>void mul_kernel&lt;double&gt;(double*, double const*) [clone .kd]</td>
<td>0.052</td>
<td>0.052</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>void triad_kernel&lt;double&gt;(double*, double const*, double const*) [clone .kd]</td>
<td>0.68</td>
<td>0.68</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

```bash
tau_exec -T rocm,serial -rocm /hip-stream
```
Rocprof

- Statistics for kernels and names (see the created csv file):

```
rocprof --stats ./hip-stream
```

- Create a metrics.txt file with content (choose metrics):

```
pmc: GPUBusy Wavefronts VALUInsts SALUInsts SFetchInsts MemUnitStalled VALUUtilization
VALUBusy SALUBusy WriteUnitStalled
range: 0:100
gpu: 0
kernel: add_kernel copy_kernel triad_kernel dot_kernel mul_kernel
```

```
rocprof -i metrics.txt ./hip-stream
```
Tuning

- Multiple wavefronts per compute unit (CU) is important to hide latency and instruction throughput
- Tune number of threads per block, number of teams for OpenMP offloading etc.
- Memory coalescing increases bandwidth
- Unrolling loops allow compiler to prefetch data
- Small kernels can cause latency overhead, adjust the workload
- Use of Local Data Share (LDS) memory
Conclusion/Future work

• A code written in C/C++ and MPI+OpenMP is a bit easier to be ported to OpenMP offload compared to other approaches.

• The hipSYCL could be a good option considering that the code is in C++. Good support from hipSYCL.

• There can be challenges, depending on the code and what GPU functionalities are integrated to an application

• It will be required to tune the code for high occupancy

• Profiling should be used to identify bottlenecks

• Track historical performance among new compilers

• GCC for OpenACC and OpenMP Offloading for AMD GPUs

• Tracking how profiling tools work on AMD GPUs

• We have trained more than 80 people on HIP porting: [http://github.com/csc-training/hip](http://github.com/csc-training/hip)
Acknowledgements

• My colleagues from CSC
• Nicholas Malaya from AMD for the many conversations and emails that we have exchanged
• Michael Klemm from AMD for the OpenMP discussions
• Many people from AMD discussing about issues and future
• HPE for the conversations and presentations
Questions?

Georgios.Markomanolis@csc.fi