Getting started with AMD GPUs
FOSDEM’21 HPC, Big Data and Data Science devroom
February 7th, 2021

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

• Motivation
• LUMI
• ROCm
• Introduction and porting codes to HIP
• Benchmarking
• Fortran and HIP
• Tuning
Disclaimer

• AMD ecosystem is under heavy development, many things can change without notice
• All the experiments took place on NVIDIA V100 GPU (Puhti cluster at CSC)
• Trying to use the latest versions of ROCm
• Some results are really fresh and investigating the outcome
LUMI, the Queen of the North

LUMI is a Tier-0 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-0 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
Motivation/Challenges

• LUMI will have AMD GPUs
• Need to learn how to program and port codes on AMD ecosystem
• Provide training to LUMI users
• Investigate in the future about possible problems
• Not yet access to AMD GPUs
AMD GPUs (MI100 example)

LUMI will have a different GPU

AMD MI100
Differences between HIP and CUDA

• AMD GCN hardware wavefronts size is 64 (like warp for CUDA)
• Some CUDA library functions do not have AMD equivalents
• Shared memory and registers per thread can differ between AMD and NVIDIA hardware
ROCm

• Open Software Platform for GPU-accelerated Computing by AMD
**ROCM installation**

- Many components need to be installed
- ROCm-cmake
- ROCT Thunk Interface
- HSA Runtime API and runtime for ROCm
- ROCM LLVM / Clang
- Rocminfo (only for AMD HW)
- ROCm-Device-Libs
- ROCm-CompilerSupport
- ROCclr - Radeon Open Compute Common Language Runtime
- HIP


Repo: [https://github.com/cschpc/lumi](https://github.com/cschpc/lumi)
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
## 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), cudaMemcpyHostToDevice);</code></td>
<td><code>hipMemcpy(d_x,x,N*sizeof(double), 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

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

HIP

hipLaunchKernelGGL(kernel_name, gridsize, blocksize, shared_mem_size, stream, arg0, arg1, ...);
**HIP API**

- **Device Management**
  - `hipSetDevice()`, `hipGetDevice()`, `hipGetDeviceProperties()`, `hipDeviceSynchronize()`

- **Memory Management**
  - `hipMalloc()`, `hipMemcpy()`, `hipMemcpyAsync()`, `hipFree()`, `hipHostMalloc`

- **Streams**
  - `hipStreamCreate()`, `hipStreamSynchronize()`, `hipStreamDestroy()`

- **Events**
  - `hipEventCreate()`, `hipEventRecord()`, `hipStreamWaitEvent()`, `hipEventElapsedTime()`

- **Device Kernels**
  - `__global__`, `__device__`, `hipLaunchKernelGGL()`

- **Device code**
  - `threadIdx`, `blockIdx`, `blockDim`, `__shared__`
  - Hundreds math functions covering entire CUDA math library

- **Error handling**
  - `hipGetLastError()`, `hipGetErrorString()`

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.
Hipify-perl (cont).

- 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
Hipify-perl (cont).

**CUDA**

```c
#include <cuda_runtime.h>
#include "cublas_v2.h"

if (cudaSuccess != cudaMalloc((void **) &a_dev,
sizeof(*a) * n * n) ||
cudaSuccess != cudaMalloc((void **) &b_dev,
sizeof(*b) * n * n) ||
cudaSuccess != cudaMalloc((void **) &c_dev,
sizeof(*c) * n * n)) {
    printf("error: memory allocation (CUDA)\n");
cudaFree(a_dev); cudaFree(b_dev);
cudaFree(c_dev);
    cudaDestroy(handle);
    exit(EXIT_FAILURE);
}
```

**HIP**

```c
#include <hip/hip_runtime.h>
#include "hipblas.h"

if (hipSuccess != hipMalloc((void **) &a_dev,
sizeof(*a) * n * n) ||
    hipSuccess != hipMalloc((void **) &b_dev,
sizeof(*b) * n * n) ||
    hipSuccess != hipMalloc((void **) &c_dev,
sizeof(*c) * n * n)) {
    printf("error: memory allocation (CUDA)\n");
    hipFree(a_dev); hipFree(b_dev); hipFree(c_dev);
    hipblasDestroy(handle);
    exit(EXIT_FAILURE);
}
```
Compilation

1) Compilation with \texttt{CC=hipcc}

matMulAB.c:21:10: fatal error: hipblas.h: No such file or directory 21 | #include "hipblas.h"

2) Install HipBLAS library *

3) Compile again and the binary is ready. When the HIP is on NVIDIA hardware, the .cpp file should be compiled with the option “hipcc -x cu …”.

- The hipcc is using nvcc on NVIDIA GPUs and hcc for AMD GPUs

* https://github.com/cschpc/lumi/blob/main/hip/hipblas.md
Hipify-clang

• Build from source

• Some times needs to include manually the headers -I/...

   $ hipify-clang --print-stats -o matMul.o matMul.c

   [HIPIFY] info: file 'matMul.c' statistics:
   CONVERTED refs count: 0
   UNCONVERTED refs count: 0
   CONVERSION %: 0
   REPLACED bytes: 0
   TOTAL bytes: 4662
   CHANGED lines of code: 1
   TOTAL lines of code: 155
   CODE CHANGED (in bytes) %: 0
   CODE CHANGED (in lines) %: 1
   TIME ELAPSED s: 22.94
Benchmark MatMul OpenMP ofload

• Use the benchmark https://github.com/pc2/OMP-Offloading for testing purposes, matrix multiplication of 2048 x 2048

• All the CUDA calls were converted and it was linked with hipBlas among also OpenMP offload

• CUDA

  matMulAB (11) : 1001.2 GFLOPS 11990.1 GFLOPS maxabserr = 0.0

• HIP

  matMulAB (11) : 978.8 GFLOPS 12302.4 GFLOPS maxabserr = 0.0

• For the most executions, HIP version was equal or a bit better than CUDA version, for total execution, there is ~2.23% overhead for HIP using NVIDIA GPUs
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

• HIP calls: hipMemcpy, hipMalloc, hipMemcpyHostToDevice, hipMemcpyDeviceToHost, hipLaunchKernelGGL, hipDeviceSynchronize, hip_runtime, hipSuccess, hipGetErrorString, hipGetLastError, hipError_t, HIP_DYNAMIC_SHARED

• 32768 number of small particles, 2000 time steps

• CUDA execution time: 68.5 seconds

• HIP execution time: 70.1 seconds, ~2.33% overhead
**Fortran**

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

- **Second Scenario: CUDA Fortran**
  - There is no HIP equivalent
  - HIP functions are callable from C, using `extern C`
  - 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
- The HIP version is up to 30% faster, seems to be a comparison between nvcc and pgf90, still checking to verify some results
- Example of Fortran with HIP: https://github.com/cschpc/lumi/tree/main/hipfort
Fortran CUDA example (cont.)

Original Fortran CUDA

```fortran
module mathops
contains
  attributes(global) subroutine saxpy(x, y, a)
    implicit none
    real, value :: y(:), a
    integer :: i, n
    n = size(x)
    i = blockDim%x + (blockIdx%x - 1) + threadIdx.x
    if (i < n) y(i) = y(i) + a*x(i)
  end subroutine saxpy
end module mathops

program testSaxpy
  use mathops
  use cudafor
  implicit none
  integer, parameter :: N = 1000000000
  real :: x(N), y(N), a
  real, device :: d_x(N), d_y(N)
  type(dim3) :: grid, tBlock
  tBlock = dim3(256,1,1)
  grid = dim3(ceiling(real(N)/tBlock%x),1,1)
  x = 1.0; y = 2.0; a = 2.0
  y_d = y
  call saxpy<<grid, tBlock>>>(x_d, y_d, a)
  y = y_d
  write(*,*) 'Max error: ', maxval(abs(y-4.0))
end program testSaxpy
```

New Fortran 2003 with HIP

```fortran
program testSaxpy
  use hipfort
  use hipcheck
  implicit none
  interface
    subroutine launchy(x, y, a)
      use hip_kernels
      implicit none
      type(hip_ptr) :: y, a
      integer, value :: x, n
      n = size(x)
      i = blockDim.x + (blockIdx.x - 1) + threadIdx.x
      if (i < n) y(i) = y(i) + a*x(i)
    end subroutine launchy
  end interface
  implicit none
  integer, parameter :: N = 1000000000
  real, device :: d_x(N), d_y(N)
  type(dim3) :: d_grid, d_tBlock
  d_grid = dim3(ceiling(real(N)/d_tBlock%x),1,1)
  d_tBlock = dim3(256,1,1)
  call hipLaunchKernelGGL(saxpy, grid, tBlock, 0, 0, d_x, d_y, d_a, N)
  write(*,*) 'Max error: ', maxval(abs(y-4.0))
end program testSaxpy
```

C++ with HIP and extern C

```c++
#include <hip_runtime.h>
#include <cstddef>

__global__ void saxpy(float *y, float *x, float a, int n)
{
    size_t i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) y[i] = y[i] + a*x[i];
}

extern "C"
{
    grid launch(float **dout, float **da, float db, int N)
    {
        dim3 d_tBlock(256,1,1);
        dim3 d_grid(call(float)N/d_tBlock.x,1,1);
        hipLaunchKernelGGL(saxpy, grid, d_tBlock, 0, 0, *dout, *da, db, N);
    }
}
```
AMD OpenMP (AOMP)

• We have tested the LLVM provided OpenMP offload and gets improved by the time
• AOMP is under heavy development and we started testing it.
• AOMP has still some performance issues according to some public results but we expect to be also improved significantly by the time LUMI is delivered
• https://github.com/ROCm-Developer-Tools/aomp
OpenMP or HIP?

• Some users will be questioning about the approach

• OpenMP can provide a quick porting but it is expected with HIP to have better performance as we avoid some layers like that.

• For complicated codes and programming languages as Fortran, probably OpenMP could provide a benefit. Always profile your code to investigate the performance.
Porting code to LUMI (not official)
Profiling/Debugging

• AMD will provide APIs for profiling and debugging
• Cray will support the profiling API through CrayPat
• Some well known tools are collaborating with AMD and preparing their tools 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));
```
Tuning

• Multiple wavefronts per compute unit (CU) is important to hide latency and instruction throughput
• 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)
Programming models

- OpenACC will be available through the GCC as Mentor Graphics (now called Siemens EDA) is developing the OpenACC integration.

- Kokkos, Raja, Alpaka, and SYCL should be able to be used on LUMI but they do not support all the programming languages.
Conclusion

• Depending on the code the porting to HIP can be more straightforward.
• There can be challenges, depending on the code and what GPU functionalities are integrated into an application.
• There are many approaches to port a code and you should select the one that you are more familiar with and provides as much as possible as good performance.
• It will be required to tune the code for high occupancy.
• Profiling can help to investigate data transfer issues.
• Probably is more productive to try OpenMP with offload to GPUs initially with Fortran codes.
Thank you!

georgios.markomanolis@csc.fi