Evaluating AMD HIP Toolset for Migrating CUDA Applications to AMD GPUs

Yan Zhan
NCSA SPIN 2016 Research Intern
Computer Engineering, Junior

Dr. Volodymyr Kindratenko
Senior Research Scientist
National Center for Supercomputing Applications

1 Abstract

The Heterogeneous-compute Interface for Portability (HIP) is part of AMDs Boltzmann Initiative launched at SC15[1]. HIP is a framework and a toolset that converts CUDA code to more standardized C++ code, and enables CUDA applications to run on AMD GPUs while retaining the ability to run on NVIDIA GPUs. This research aims to evaluate the effectiveness of the HIP toolset and serves as the basis for future efforts to utilize HIP for migrating CUDA applications.

2 Introduction

AMD (previously ATI) and NVIDIA have been competing in the GPU computing field for a long time. Both companies have their own software stack for developing applications on their hardware. Since AMD tends to release hardware with better theoretical peak performance sooner than NVIDIA while NVIDIA tends to have better software support, a need for easy code migration from NVIDIA to AMD arises. However, it has always been a challenge to migrate from one platform to the other, due to the differences in their software stacks. The HIP toolset is therefore developed by AMD to help this process. It automatically replaces CUDA-specific keywords in source files with keywords of the HIP framework, eliminating the need for painstakingly changing code by hand to OpenCL[2]. Also, instead of converting the code to suit AMDs software stack, HIP converts the code to use an intermediate framework, the HIP framework, which could run on both AMD and NVIDIA GPUs, making it much easier to develop applications for both hardware platforms.

3 Evaluation methodology

3.1 Hardware/software environment

We deployed two compute nodes, one with NVIDIA Geforce GTX 980 GPU and one with AMD R9 Nano GPU. Other than GPUs, the server hardware configurations of these two nodes are identical (see Table 1, Table 2). We installed Ubuntu Server 15.04 on both nodes. We installed CUDA 7.5[3] and HIP 0.86[4] on the NVIDIA node, and the ROCm 1.2 software stack[5] on the AMD node.

Processor	2x Intel Xeon E5-2609 v3
Core Count	2x 6-core
Core Clock	1.90 GHz
Memory	32 GB DDR4
Memory Clock	2133 MHz
Storage	NFS-mounted file system over Infiniband QDR network

Table 1: Node configuration

GPU	AMD Radeon R9 Nano	NVIDIA Geforce GTX 980
Architecture	GCN 3rd generation	Maxwell
Core Count	4096	2048
Core Clock	1000 MHz	1126 MHz, 1216 MHz (boost)
Memory	4 GB HBM	4 GB GDDR5
Memory Interface Width	4096-bit	256-bit
Memory Bandwidth	512 GB/s	224 GB/s
Theoretical FP32 Performance	8192 GFLOPS	4612 GFLOPS
Theoretical FP64 Performance	512 GFLOPS	144 GFLOPS
Board Power	175 W	165 W

Table 2: GPU characteristics[6][7]

3.2 Workflow

Since CUDA uses C and HIP uses C++, some minor work may need to be done to update C source code to compile with the C++ compiler. Once this is done, the following workflow can be applied to convert CUDA-based application:

1. Use **hipify** utility to convert CUDA source (*.cu) files:
   `hipify cuda.cu > cuda.cpp`

2. Find the kernel function in the converted code and add the HIP launch parameter to the argument list. HIP uses a standard C++ structure to pass execution configuration as opposed to <<< >>> notation used in CUDA:
   ```
   __global__ void kernel(hipLaunchParm lp, float *out, float *in)
   ```

3. Use **hipcc** to compile all source files:
   ```
   hipcc -I./ -O3 -c main.cpp -o main.o
   hipcc -I./ -O3 -c cuda.cpp -o cuda.o
   hipcc main.o cuda.o -o main -lm
   ```

Now we have an executable file. Note that manual code tuning may be necessary for optimal performance.

3.3 Initial experiments with NVIDIA SDK sample code

We used a series of tests with gradually increasing complexity to test the capability of HIP toolset. To start, we installed HIP 0.82 on an NVIDIA-based system and we have chosen matrixMul as the first program to test. This is a simple program included with NVIDIA's CUDA SDK that performs matrix multiplication (see Source code 1 in Appendix). Since matrixMul consists of only one source file and does not use any CUDA-optimized libraries, it should present minimum difficulty to HIP. Indeed, HIP converted the program effortlessly (see Source code 2 in Appendix), and the resulting code compiled and ran without any issues on NVIDIA hardware.

Next, we installed HIP 0.82 on a system with AMD GPU. The converted matrixMul code failed to compile at our first attempt. A closer look at the code revealed that the code uses some helper functions from CUDA SDK, which do not exist in an AMD environment. These functions don't actually do anything specific to CUDA, so we simply replaced them with generic implementations using standard C++. This was sufficient to compile and run the program.

3.4 Initial experiments with cuda-lapl code

Cuda-lapl code[8] provided by the Computation-based Science and Technology Research Center (CaSToRC) at the Cyprus Institute performs a discrete 2D laplacian operation. We used it to test the capability of HIP 0.82 as well as to develop a workflow for converting CUDA programs for execution on AMD hardware. As mentioned earlier, HIP

only supports C++, so some minor changes were required to make the code compilable by the C++ compiler. With these changes, HIP was able to compile the code, but the executable program would crash due to an error within the HIP 0.82 framework. We filed a bug report with AMD.

3.5 Current state with cuda-lapl code

In June 2016, AMD released HIP version 0.86[9]. The new version brought several new features as well as bug fixes. With this release of HIP, we were able to successfully compile and run cuda-lapl code on our AMD GPU system. Figure 1 shows the achieved performance as a function of lattice size. We ran the converted version of cuda-lapl on both nodes, as well as the original CUDA version on the NVIDIA node as a reference.

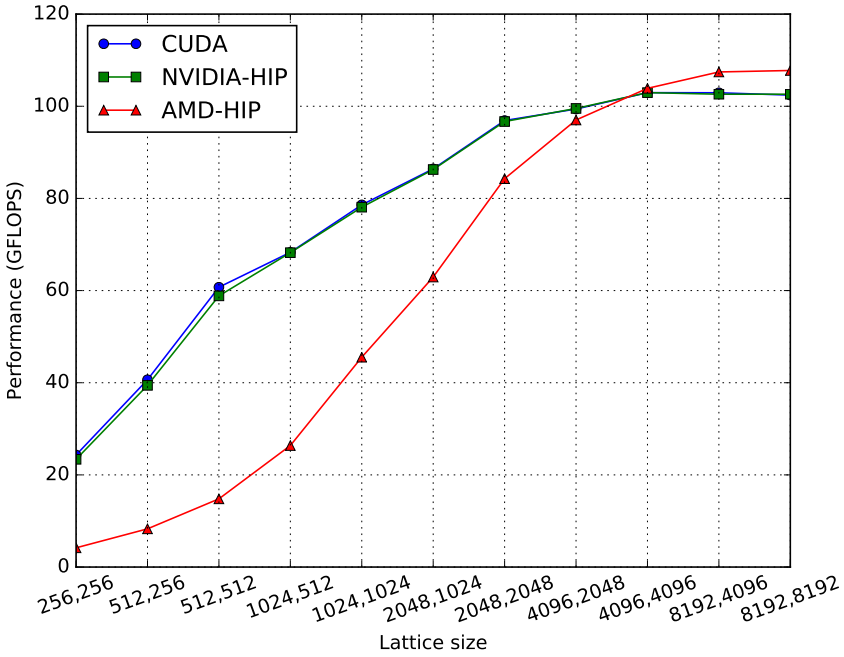

Figure 1. Performance of cuda-lapl vs. Lattice size

As we can see from the chart above, HIP introduced little to no performance impact for the NVIDIA GPU. The performance of the converted program is nearly identical to that of the native CUDA implementation. Running on the AMD GPU, the program starts out slower, but its performance improves steadily as the problem size grows. For a sufficiently large lattice, code converted with HIP tool delivers the same performance on AMD GPU as the original CUDA code on the NVIDIA GPU. It is worth noting, however, that the AMD GPU we used in this work has 75% more theoretical peak performance than its NVIDIA

counterpart. Since the program is not particularly optimized for either GPU, there are probably plenty of opportunities for improvement on both systems.

3.6 Experiments with varying launching parameters

In an attempt to improve the performance without modifying the code extensively, we ran cuda-lapl and its converted version with different launching parameters. The graphs below show the effect of different thread block dimensions on the performance. The grid dimensions are calculated from the dimensions of thread blocks and input lattices so that there are enough blocks to cover input lattices.

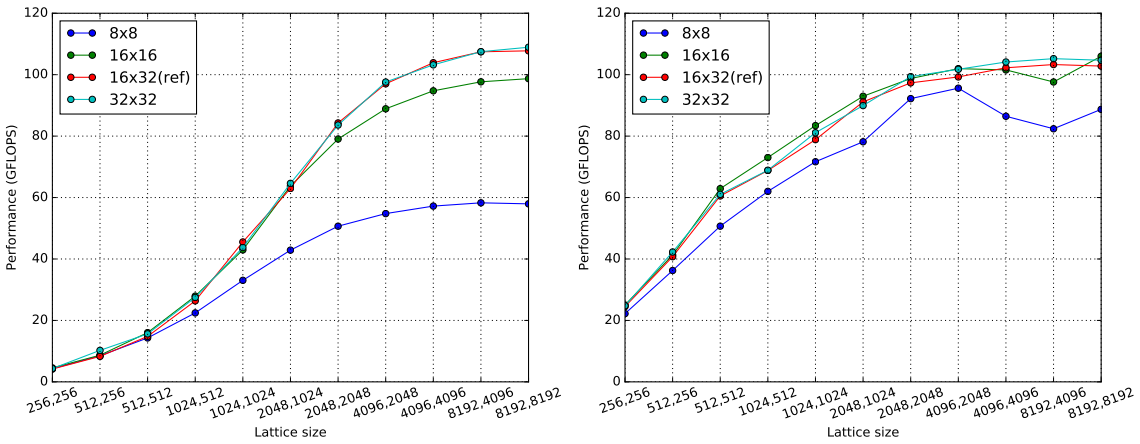

Figure 2. Performance variation with different block sizes. Left: AMD; Right: NVIDIA.

Experiment data shows that the block size we used initially is already close to optimal. With different block sizes, we achieved only a minor and inconsistent performance increase. An improved algorithm would be required to improve performance further.

4 Conclusions & future work

With performance data from our tests, we have demonstrated that HIP is a workable solution for migrating CUDA applications to AMD GPUs. The main advantage of AMD R9 Nano over NVIDIA GPUs is its high performance-to-power ratio (in single-precision), and HIP tools can help us utilize the computing power of these cards for suitable codes. Our future work will focus on performance optimizations of the codes converted with HIP tool as well as code scalability on a multi-GPU cluster. Ultimately, we would like to port QUDA library[10] to AMD GPUs.

References

[1] AMD, Inc. (2015, November 16). *AMD Launches Boltzmann Initiative to Dramatically Reduce Barriers to GPU Computing on AMD FirePro Graphics*. Retrieved August 1, 2016, from http://www.amd.com/en-us/press-releases/Pages/boltzmann-initiative-2015nov16.aspx

[2] AMD, Inc. (n.d.). *Porting CUDA Applications to OpenCL*. Retrieved June 27, 2016, from http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/programming-in-opencl/porting-cuda-applications-to-opencl/

[3] NVIDIA Corp. (2013). *CUDA Toolkit*. Retrieved August 25, 2016, from https://developer.nvidia.com/cuda-toolkit

[4] AMD Inc. (n.d.). *HIP: C Heterogeneous-Compute Interface for Portability*. Retrieved August 25, 2016, from http://gpuopen.com/compute-product/hip-convert-cuda-to-portable-c-code/

[5] AMD, Inc. (n.d.). *ROCm: Open Platform For Development, Discovery and Education around GPU Computing*. Retrieved August 04, 2016, from http://gpuopen.com/compute-product/rocm/

[6] NVIDIA Corp. (n.d.). GeForce GTX 980 Desktop Graphics Card. Retrieved August 18, 2016, from http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications

[7] AMD, Inc. (n.d.). AMD Radeon R9 Nano, World's Smallest and Most Power-Efficient Enthusiast Graphics Card, Brings 4K Gaming to the Living Room. Retrieved August 18, 2016, from http://www.amd.com/en-us/press-releases/Pages/amd-radeon-r9-nano-2015aug27.aspx

[8] Giannis Koutsou. (2014, April 29). *cuda-lapl*. Retrieved August 02, 2016, from https://github.com/gpucw/cuda-lapl

[9] AMD, Inc. (n.d.). *HIP Release 0.86 Now Available*. Retrieved August 04, 2016, from http://gpuopen.com/hip-release-0-86/

[10] M. A. Clark, R. Babich, K. Barros, R. Brower, and C. Rebbi, *Solving Lattice QCD systems of equations using mixed precision solvers on GPUs*, Comput. Phys. Commun. 181, 1517 (2010) [arXiv:0911.3191 [hep-lat]].

5 Appendix

Highlighted lines were modified by the HIP tool.

5.1 Original CUDA matrixMul kernel

```
template <int BLOCK_SIZE> __global__ void
matrixMulCUDA(float *C, float *A, float *B, int wA, int wB)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd   = aBegin + wA - 1;
    int aStep  = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep  = BLOCK_SIZE * wB;
    float Csub = 0;

    for (int a = aBegin, b = bBegin; a <= aEnd;
            a += aStep, b += bStep)
    {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];
        __syncthreads();

#pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            Csub += As[ty][k] * Bs[k][tx];
        }
        __syncthreads();
    }

    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

5.2 matrixMul kernel translated with HIP tool

```
template <int BLOCK_SIZE> __global__ void
matrixMulCUDA(hipLaunchParm lp, float *C, float *A, float *B, int wA, int wB)
{
    int bx = hipBlockIdx_x;
    int by = hipBlockIdx_y;
    int tx = hipThreadIdx_x;
    int ty = hipThreadIdx_y;
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd   = aBegin + wA - 1;
    int aStep  = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep  = BLOCK_SIZE * wB;
    float Csub = 0;

    for (int a = aBegin, b = bBegin; a <= aEnd;
            a += aStep, b += bStep)
    {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];
        __syncthreads();

#pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            Csub += As[ty][k] * Bs[k][tx];
        }
        __syncthreads();
    }

    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

5.3 Snippets of host code from matrixMul

5.3.1 Memory allocation & error handling

```
1  cudaError_t error = cudaMalloc((void **) &d_A, mem_size_A);
2  if (error != cudaSuccess)
3  {
4      printf("cudaMalloc d_A returned error %s (code %d), line(%d)\n",
           cudaGetErrorString(error), error, __LINE__);
5      exit(EXIT_FAILURE);
6  }
7  ...
8  error = cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
9  ...
10 cudaFree(d_A);
```

Original code

```
1  hipError_t error = hipMalloc((void **) &d_A, mem_size_A);
2  if (error != hipSuccess)
3  {
4      printf("hipMalloc d_A returned error %s (code %d), line(%d)\n",
           hipGetErrorString(error), error, __LINE__);
5      exit(EXIT_FAILURE);
6  }
7  ...
8  error = hipMemcpy(d_A, h_A, mem_size_A, hipMemcpyHostToDevice);
9  ...
10 hipFree(d_A);
```

Translated code

5.3.2 Kernel invocation

```
1  matrixMulCUDA<16><<< grid, threads >>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
```

Original code

```
1  hipLaunchKernel(HIP_KERNEL_NAME(matrixMulCUDA<16>), dim3(grid),
2      dim3(threads), 0, 0, d_C, d_A, d_B, dimsA.x, dimsB.x);
```

Translated code

5.3.3 Device query

```
int devID = 0;
if (checkCmdLineFlag(argc, (const char **)argv, "device"))
{
    devID = getCmdLineArgumentInt(argc, (const char **)argv, "device");
    cudaSetDevice(devID);
}

cudaError_t error;
cudaDeviceProp deviceProp;
error = cudaGetDevice(&devID);
... /*** error checking code omitted for clarity ***/
error = cudaGetDeviceProperties(&deviceProp, devID);
... /*** error checking code omitted for clarity ***/
printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID,
   deviceProp.name, deviceProp.major, deviceProp.minor);
```

Original code

```
int devID = 0;
if (checkCmdLineFlag(argc, (const char **)argv, "device"))
{
    devID = getCmdLineArgumentInt(argc, (const char **)argv, "device");
    hipSetDevice(devID);
}

hipError_t error;
hipDeviceProp_t deviceProp;
error = hipGetDevice(&devID);
... /*** error checking code omitted for clarity ***/
error = hipGetDeviceProperties(&deviceProp, devID);
... /*** error checking code omitted for clarity ***/
printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n", devID,
   deviceProp.name, deviceProp.major, deviceProp.minor);
```

Translated code