Iowa High Performance Computing Summer School 2013

GPU Programming Using CUDA

This document describes how to get set up to write and run CUDA C codes for execution on the NVIDIA GeForce GTX 580 GPU that is installed on the Helium cluster at the University of Iowa.

1 Getting Started with the GPU on Helium

1. Note that somewhat minimal information on the GPU on the Helium cluster is available in the online Helium documentation at
   `https://www.icts.uiowa.edu/confluence/display/ICTSit/GPU+computer`
2. VERY IMPORTANT NOTE: There is only one GPU available for use in this summer school. Therefore, any students wishing to try out CUDA programming need to share this resource with the other students in the class. Only one person can be logged onto the computer with the GPU at one time (via `qlogin`, see below). It is necessary to be logged onto the machine to compile (using `nvcc`) or to run your CUDA C code on the GPU. Editing of your code, on the other hand, does not need to be done on the GPU computer.

   Therefore, you should write your program while are you are not logged into the GPU computer, and only log in to compile and run, then log out from the GPU to let someone else have an opportunity.
3. To log into the GPU computer on Helium, first check that no one else is using the GPU using the command
   `qstat -q GPU`
   If the command returns nothing, then the GPU is available. To login to the GPU computer, use the command
   `qlogin -q GPU`
4. Load the CUDA module to set up the environment
   `module load cuda_4.0.17`
5. To compile your CUDA C code `cuda_add.cu`, use the command
   `nvcc -o cuda_add.e cuda_add.cu`
6. To run the CUDA code, there is no queuing system (since there is only a single machine, and if you are logged onto the machine, you have full control of it—again, during this IHPC 2013 Summer School course, please be considerate of others who may want to use the GPU). So you can simply run the compiled executable
   `cuda_add.e`
7. To install the Software Developer Kit (SDK) from NVIDIA in your home directory, follow these steps:
   (a) First, be sure the you have installed the `cuda_4.0.17` module (see above).
   (b) Issue the command
   `sh /share/apps/CUDA/gpucomputingsdk_4.0.17_linux.run`
   and follow the instructions (you can just hit enter at the prompts for the default installation in your home directory).
   (c) Navigate into the newly installed directory
   `cd NVIDIA_GPU_Computing_SDK/`
   Inside this directory in the `src` directory are a number of example CUDA C source codes. You can look at these for inspiration, however all of these examples are incredibly complicated (in particular, error checking and other supplementary routines easily obscure the important lines of code necessary to compute on the GPU).

(d) WARNING: *This step takes about 5 min, and if others are waiting to use the GPU, this could present a problem.* To compile all of the example codes in the SDK, make the SDK installation
`make cuda-install=/opt/cuda`
This will compile all of the example codes in the NVIDIA GPU Computing SDK. The codes will show up in the directory `C/bin/linux/release`. You can run these tests that will use the GPU, but I have not found them to be particularly illuminating.

2 CUDA C Programming

The general concept of GPU programming is to use the tremendous computational horsepower of the GPU to perform calculations in parallel and achieve significant speedups over a serial code. The programmer defines C functions, called *kernels*, that are called by the host (CPU) and are executed N times in parallel on the device (GPU) by N different CUDA threads. The host and device have separate physical memory, and the GPU can only perform a computation using data on device memory. Therefore, the programmer must first copy the data from the host memory to the device memory, then call the kernel to compute using the GPU, and finally copy the result back from the device memory to the host memory.

1. An excellent resource for learning to program CUDA C is the NVIDIA CUDA C Programming Guide, version 4.2, available at
`http://developer.nvidia.com/nvidia-gpu-computing-documentation`
This document is a very thorough introduction to CUDA programming, both presenting the main concepts of GPU programming with CUDA as well as discussing in detail the programming interface. I highly recommend anyone interested in learning GPU computing to spend the time reading through this valuable guide. As well, the NVIDIA website contains a lot of interesting and useful information about GPU computing
`http://www.nvidia.com/object/what-is-gpu-computing.html`

2. Declaration specifications

   (a) Kernel functions are called by the host (CPU), but executed on the device (GPU). Kernel functions are specified by `__global__`.

   (b) In addition, there is a `__host__` specification for functions that are called from the host and executed on the host (these are normal C functions, in which case the `__host__` specifier is unnecessary), and a `__device__` specification for functions that are called from the device and executed on the device.

   (c) To summarize

Specifier	Executed on	Called from
`__global__`	Device	Host
`__device__`	Device	Device
`__host__`	Host	Host

3. Memory functions

   (a) To allocate memory on the GPU device, use `cudaMalloc`, for example
   `cudaMalloc(&d_A, size);`

   (b) To free allocated memory,
   `cudaFree(d_A);`

   (c) To copy data from host memory to device memory, or vice versa, use `cudaMemcpy` with the `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost` argument, for example
```
cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );
```

4. Kernel functions

   (a) Kernel functions are declared with specifier `__global__`

   (b) Kernels only operate on data stored in device memory

   (c) The kernel function is a typical C function, but the loop over which the operation is to be performed is removed. For example, when perform a computation on a 1D array (vector), instead of
   ```
   for (int i=0; i < n; i++)
       y[i]= a*x[i] + y[i];
   ```
   you will have
   ```
   int i=blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i]= a*x[i] + y[i];
   ```

   (d) The kernel is invoked using the new *execution configuration* syntax,
   `VecAdd<<<blocksPerGrid,threadsperBlock>>>(d_a,d_b,d_c);`
   where the execution configuration parameter define the thread hierarchy for the parallel computation on the GPU device.

   (e) The number of threads per block `threadsperBlock` is limited by the number of cores the GPU has. For the GeForce GTX 580, `threadsperBlock`≤ 512.

   (f) The number of blocks used for the computation is unlimited, and generally depends on the size of the problem being computed. For example, for the vector addition of two N element vectors, the number of blocks is effectively the total number of elements divided by the number of threads per block,
   `int blocksPerGrid=(N+threadsPerBlock-1)/threadsPerBlock;`

   (g) The variables `blocksPerGrid` and `threadsperBlock` can be specified as 1D, 2D, or 3D unsigned integer arrays. To allow the kernel to access the appropriate thread ID and use it to compute the correct matrix element to operate on, there are several built-in variables from CUDA:
   `threadIdx`
   `blockDim`
   `blockIdx`
   To access the appropriate dimension of these built in variables, you use `threadIdx.x`, `threadIdx.y`, or `threadIdx.z`.

3 GPU Specifications on Helium

NVIDIA GeForce GTX 580 Graphics Processing Unit (GPU):

CUDA cores	512
Fermi Architecture	Compute Capability 2
Memory	1536 MB
Core clock	772 MHz
Processor clock	1544 MHz
Effective Memory clock	4008 MHz