Iowa High Performance Computing Summer School 2015
GPU Programming Using CUDA

By Michael J. Schnieders and Gregory G. Howes

This document describes how to get set up to write and run CUDA C codes for execution on the NVIDIA Kepler GPUs installed on the Neon cluster at The University of Iowa.

1 Getting Started with Kepler GPUs on Neon

1. Note that there is CUDA information for the Neon cluster available in the online documentation at https://wiki.uiowa.edu/display/hpcdocs/CUDA

2. Important Note: There are three Kepler GPU nodes 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. Keep in mind that editing of code does not need to be done on a GPU node.

3. To log into a GPU computer on Neon, use the command
   `qlogin -q IHPC -l kepler`

4. To edit, compile or run CUDA code, load the CUDA module to set up the environment
   `module load cuda/6.5`

5. To compile your CUDA C code `cuda_add.cu`, use the command
   `nvcc -o cuda_add.e cuda_add.cu`

6. The CUDA executable can be run while logged into a Kepler node:
   `cuda_add.e`
   Or, it can be run via a qsub job with the following *additional* lines added to the file:
   ```
   #$ -l kepler
   module load cuda/6.5
   cuda_add.e
   ```

7. To install CUDA samples from NVIDIA into your home directory, follow these steps:
   a) First, be sure you have installed the cuda 6.5 module (see above).

   b) Issue the command
      `cuda-install-samples-6.5.sh samples`

   c) Navigate into the newly installed directory
      `cd samples/NVIDIA_CUDA-6.5_Samples`
      Inside this directory are a number of example CUDA C source codes. You can look at these for inspiration, however, most 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 considerable time, and if others are waiting to use a GPU node, this could present a problem. To compile all of the examples, make the installation
`make`
This will compile all of the example codes in the NVIDIA GPU Computing SDK. The codes will show up in the directory
`bin/x86_64/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 7.0, available at
http://docs.nvidia.com/cuda/cuda-c-programming-guide
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 GPU specification. For the Neon Kepler GPUs, `threadsperBlock <=` 1024.

   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 Neon

```
Device 0: "Tesla K20m"
  CUDA Driver Version / Runtime Version          6.5 / 6.5
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 4800 MBytes (5032706048 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Clock rate:                                706 MHz (0.71 GHz)
  Memory Clock rate:                             2600 Mhz
  Memory Bus Width:                              320-bit
  L2 Cache Size:                                 1310720 bytes
  Maximum Texture Dimension Size (x,y,z)  1D=(65536),2D=(65536,65536),3D=(4096,4096,4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           130 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >
```