06-26945 & 06-26944
*Distributed and Parallel Computing + Extended*
Autumn Semester 2016

The University of Birmingham
School of Computer Science
© Alan P. Sexton 2016

# *CUDA SDK — Getting Started*

These instructions are for setting up and getting started with CUDA programming in the school labs on Linux systems.

Most of the machines in the lg04 lab have an NVidia Geforce GTX 660 graphics card driving the monitor. In rows A and B, there are 12 machines, labelled cca-lg04-16063 to cca-lg04-16074 that have a much more powerful GeForce GTX 960. These machines are configured to use the Intel graphics chip on the motherboard to drive the monitors so that the NVidia cards are free for pure general purpose parallel programming tasks. **Please do not hog these machines!!!**.

## 1. Environment

To set the necessary environment variables (on the School computers only) run:

```
    module load cuda
```

When you load the `cuda` module, it will display two directories. The second, the examples directory, is where the computer officers have downloaded and compiled a large set of sample programs that come with the CUDA SDK. The binaries can be found below the `bin` sub-directory. Run the sample program `deviceQuery`. Save the results as you will be referring to it often as you develop CUDA programs. You can try the other programs but most of the graphical ones will not work on the GeForce GTX 960 machines as those machine do not use the GPU cards to drive the monitor.

Here is the output from `deviceQuery` for the GeForce GTX 960:

```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 960"
  CUDA Driver Version / Runtime Version          8.0 / 7.0
  CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 1996 MBytes (2092957696 bytes)
  ( 8) Multiprocessors, (128) CUDA Cores/MP:     1024 CUDA Cores
  GPU Max Clock rate:                            1291 MHz (1.29 GHz)
  Memory Clock rate:                             3600 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(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:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 8.0, CUDA Runtime Version = 7.0, 
Result = PASS
```

Here is the output from `deviceQuery` for a different GPU: a Geforce GTX 660. Compare the above values to those for this one:

```
CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 660"
 CUDA Driver Version / Runtime Version          7.5 / 7.5
 CUDA Capability Major/Minor version number:    3.0
 Total amount of global memory:                 2047 MBytes (2146762752 bytes)
 ( 5) Multiprocessors, (192) CUDA Cores/MP:     960 CUDA Cores
 GPU Max Clock rate:                            1098 MHz (1.10 GHz)
 Memory Clock rate:                             3004 Mhz
 Memory Bus Width:                              192-bit
 L2 Cache Size:                                 393216 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 1 copy engine(s)
 Run time limit on kernels:                     Yes
 Integrated GPU sharing Host Memory:            No
 Support host page-locked memory mapping:       Yes
 Alignment requirement for Surfaces:            Yes
 Device has ECC support:                        Disabled
 Device supports Unified Addressing (UVA):      Yes
 Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
 Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

eviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 7.5, CUDA Runtime Version = 7.5,
            NumDevs = 1, Device0 = GeForce GTX 660
Result = PASS
```

For the moment, pay particular attention to the values for :

- CUDA Capability Major/Minor version number
- Total amount of global memory
- Number of Multiprocessors
- Number of Cores
- Total amount of shared memory per block
- Warp size
- Maximum number of threads per multiprocessor
- Maximum number of threads per block
- Maximum dimension size of a thread block
- Maximum dimension size of a grid size

Investigate the meaning of these values on the Web.

## 2. The nsight IDE

`nsight` is a version of the `Eclipse` IDE modified by NVidia for CUDA project development. Like `Eclipse`, `nsight` uses a workspace directory (by default `~/cuda-workspace`) and, when you create a project, offers to store your project files there. I recommend that you do **NOT** do so, but instead that you put your project files somewhere outside the workspace directory. A good approach is to make a directory called “`cuda`” somewhere in your school subversion repository (you can make a new repository if you wish to keep it separate from your other repositories). This way you can easily delete the `~/cuda-workspace` directory if you mess up your `nsight` settings without losing your projects. Also it means you can easily work at home on your own machine sharing your project files through subversion. Note: it is possible to install the CUDA SDK on your home machine and be able to edit and compile your programs even if you don’t have a CUDA capable GPU — in this case, of course, you won’t be able to run the programs locally.

First we will import one of the sample applications and compile and run it:

1. Start `nsight`
2. Select `File|New|Cuda C/C++ Project`
3. Set the project name to “`deviceQuery`”, untick “`use default location`”, set the location to a new directory “`DeviceQuery`” under your chosen `cuda` directory, select `Executable|Import CUDA Sample` and click next
4. Select “`deviceQuery`” and click next
5. This page lets you select the version of CUDA code to generate. If you have a CUDA capable GPU on the machine it will detect it and automatically set it correctly. If not you must set it manually. The “`CUDA Capability Major/Minor version number`” from the output of `deviceQuery` tells you what you should set this to to work on that GPU. Click next
6. It is possible to run `nsight` locally while compiling for a different architecture and running the result code remotely. In this case we don’t wish to do that so click next
7. click finish
8. The “`deviceQuery.cpp`” file will open. Click on the hammer button in the toolbar to build the project.
9. Click on the dropdown arrow beside the green run button in the toolbar and select `Run Configurations...`
10. Double click on `C/C++ Application`. So long as you had first built the project, this should set all the parameters correctly. Click Run

**3. Importing existing project (e.g. from subversion)**

Assuming you have kept your project files in a directory called `cuda`, away from your `~/cuda-workspace` directory, then, in `nsight`, select `File|Import...|General|Existing Projects into Workspace`, select your `cuda` directory in the `Select root directory` field and click next. You can then select any projects not previously imported to import. Note that you should not try to copy your `~/cuda-workspace` directory between your home and school machines.

**4. Installing the CUDA SDK on your own machine**

I will assume you are using a (fairly) recent version of Ubuntu. Do not use the Ubuntu CUDA packages. If you have them installed remove them. With one exception follow the instructions on: `https://www.quantstart.com/articles/Installing-Nvidia-cuda-on-Ubuntu-14-04-for-Linux-GPU-Computing`

The exception is to get the latest version of the CUDA SDK available that matches your system. The website `https://developer.nvidia.com/cuda-downloads` helps you to choose the correct one.

Once it is installed, there may be one final problem. The latest version of the CUDA SDK can only work with `gcc` and `g++` versions 4.9 or earlier. The newest Linux distributions have moved to version 5 or better. The symptom will be, on any attempt to compile a CUDA program, an error message of the form:

```
cuda #error -- unsupported GNU version! gcc versions later than 4.9 are not supported!
```

The workaround is first to install `gcc-4.9` and `g++-4.9`. These will not interfere with the existing later versions and your default compilers will remain as the later versions. Second you need to tell the CUDA SDK about them. In the CUDA bin directory (`/usr/local/cuda/bin`) create symbolic links to the two compilers under the names `gcc` and `g++`:

```
cd /usr/local/cuda/bin
ln -s /usr/bin/gcc-4.9 gcc
ln -s /usr/bin/g++-4.9 g++
```