18-645: How to write Fast Code, Spring 2017

HW #2 – Many-core Programming Infrastructure

Due At: March 1, 2017 11 PM EDT, 8PM PDT

Submission:

To submit your homework, you must carefully follow these instructions:

1. Use the text-based answer template for filling in your answers. You can find it at: Module 2 Homework 2: hw2_answer_sheet.txt
2. To get credit for the homework, please submit your completed answer sheet, save it as: **hw2_team0XX<yourFirstName>_<yourLastName>_andrewId.txt** on Canvas website under Homework 2.

Note: Completing HW2 and submitting it as instructed is required to stay in the course. As you start the homework project, you will realize that the homework is designed to help you get started on the project.

Introduction

This homework includes a CUDA installation tutorial. CUDA, stands for "Compute Unified Device Architecture". It is a general purpose programming model for programming the NVIDIA graphics processing unit (GPU) platform. The CUDA programming model allows users to kick off batches of thousands of threads on the GPU, thereby utilizing the vast amount of parallelism the hardware provides. With the CUDA programming model, the GPU based platform can allow computation to take place on a single GPU card at a throughput of TeraFLOPS (10^12 Floating point Operations Per Second), which is as fast as a TOP500 Supercomputing Center just 6 years ago. Today, many core processors are used in 3 of the 5 fastest supercomputers in the world!

This document provides a guide to help you setting up a CUDA programming environment on the ghcXX.ghc.andrew.cmu.edu, XX={26,…,46} cluster machines. By the end of this setup, you will have an environment to start programming in CUDA.

Background

To understand what we are setting up, we must first understand a bit about the underlying system infrastructure. The GPUs are acceleration cards that plugs into a computer system.

In a standard PC, the CPU (top of Figure 1) runs the operating system, and the GPU is attached through the PCI Express bus on the North Bridge (82925X MCH in Figure 1), which gives the CPU access to peripherals with high bandwidth requirements.

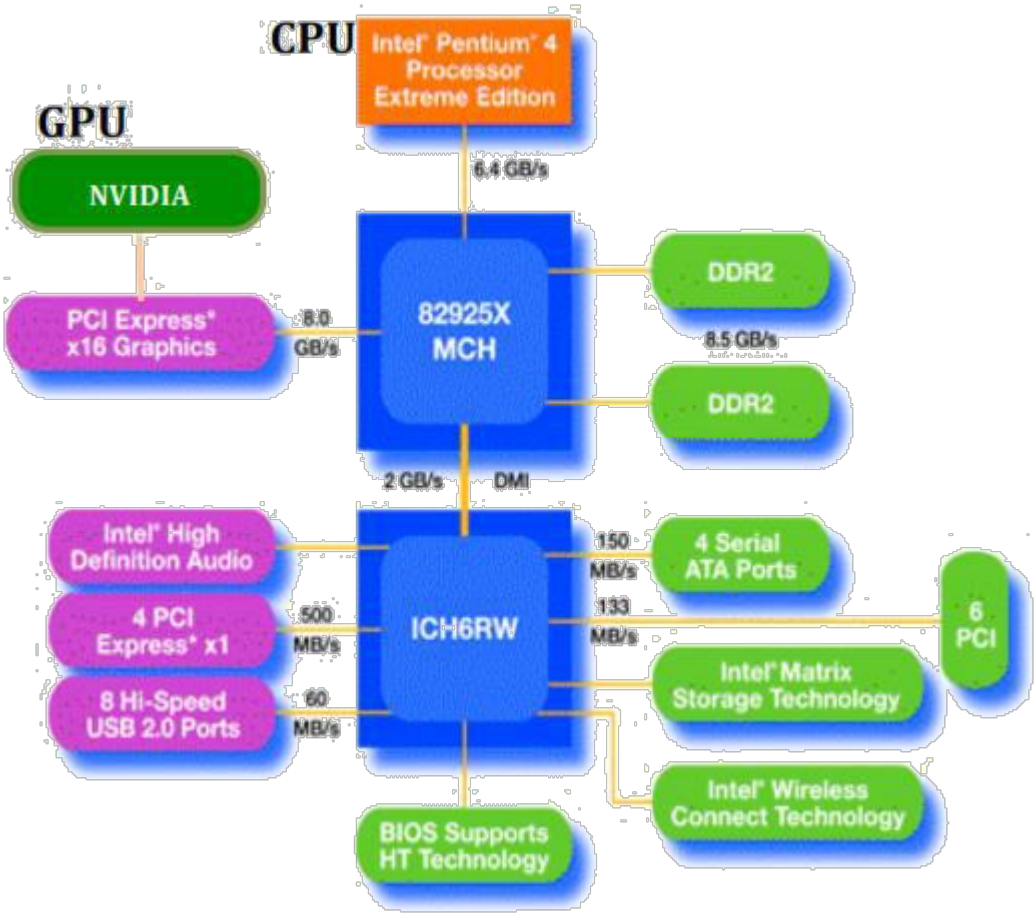


Figure 1: The system architecture for using a GPU in a PC

From a PC system builder's perspective, we are plugging a GPU graphics card on to the motherboard.

Figure 2 illustrates where a GPU can be plugged into a motherboard, and how a completely assembled system looks like. The GPU is circled in the photo of the completely assembled system.

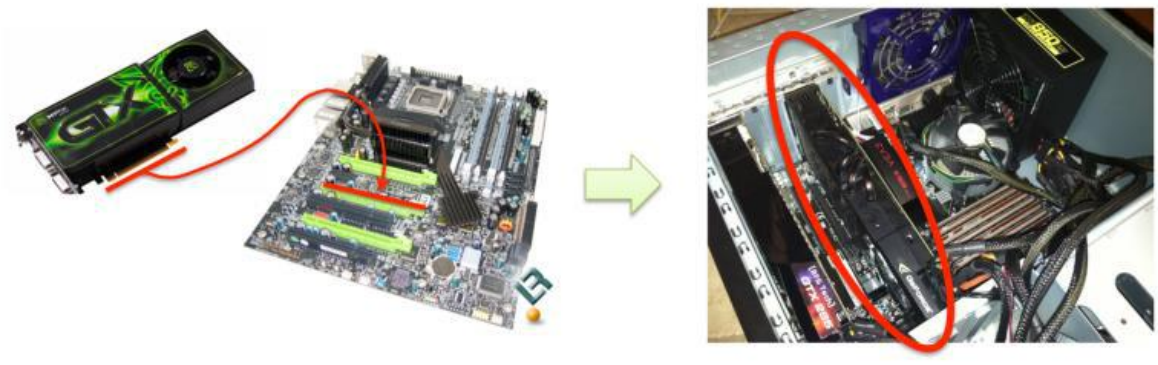


Figure 2: How GPU is plugged into the CPU motherboard

The ghc cluster consists of 20 machines, the majority of which are fitted with GPU cards. To use a machine with a GPU, you should choose a machine with the least number of teams logged in. There

are a few machines in the 20 that don't appear to have a functional GPU—if you run into problems, try a different machine in the cluster.

Setting up of the CUDA environment involves three steps: installing the CUDA developer driver, installing the CUDA toolkit, and installing the CUDA software development kit (SDK).

1. The developer driver provides a set of interfaces for the operating system to talk to the GPU subsystem.

2. The CUDA toolkit provides a compiler, a debugger, a performance profiler, and a set of pre-optimized CUDA libraries.

3. The CUDA SDK provides an infrastructure and examples to help users quickly get start on using the CUDA infrastructure.

The CUDA developer driver and CUDA toolkit has already been installed on the ISL cluster. In this tutorial, you will be setting up the CUDA SDK in your personal PROJECTS directory.

Let the fun begin!

Tasks

There are three tasks in this tutorial. The first task provides guidelines for installing the CUDA SDK. The second task will provide steps to profile your CUDA program. The third one gets you started in looking at the CUDA programs. Kindly refer the table below to find the list of GHC machines that have your GPU of interest. Try to use a random machine each time to make sure that the entire class does not hog one machine's resources.

Task 1: CUDA SDK Installation

Step 1: Setting up the CUDA environment

After logging into any ghcXX.ghc.andrew.cmu.edu, set the following environment variables.

If you are bash shell:

```
export CUDA_HOME=/usr/local/depot/cuda-8.0
export PATH=$PATH:${CUDA_HOME}/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:${CUDA_HOME}/lib64:${CUDA_HOME}/lib:$HOME/cppunit/lib
```

If you are using csh shell:

```
setenv CUDA_HOME "/usr/local/depot/cuda-8.0"
setenv PATH:${CUDA_HOME}/bin:${PATH}
setenv LD_LIBRARY_PATH
setenv LD_LIBRARY_PATH
"${LD_LIBRARY_PATH}:${HOME}/cppunit/lib:${CUDA_HOME}/lib64:${CUDA_HOME}/lib"
```

Step 2: Install SDK and build Executables

We will install GPU Computing SDK under ~/645 directory:

```
cp -r /usr/local/cuda/Samples/NVIDIA_CUDA-8.0_Samples/ ~/645
```

You should see samples copied into your 645 folder.

Once you install the SDK compile some of the samples and run

```
cd ~/645/NVIDIA_CUDA-8.0_Samples/
cd 1_Utilities/deviceQuery
make
cd ../bandwidthTest/
make
```

Step 3: Run Tests

Go to the path: ~/645/NVIDIA_CUDA-8.0_Samples/1_Utilities/deviceQuery Execute the deviceQuery executable. You should get an output like this:

./deviceQuery Starting...

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

Detected 2 CUDA Capable device(s)

Device 0: "GeForce GTX 1080"
CUDA Driver Version / Runtime Version 8.0 / 8.0
CUDA Capability Major/Minor version number: 6.1
Total amount of global memory: 8113 MBytes (8507555840 bytes)
(20) Multiprocessors, (128) CUDA Cores/MP: 2560 CUDA Cores
GPU Max Clock rate: 1734 MHz (1.73 GHz)
Memory Clock rate: 5005 Mhz
Memory Bus Width: 256-bit
L2 Cache Size: 2097152 bytes
Maximum Texture Dimension Size (x,y,z) 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 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 / 4 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

Device 1: "Quadro K620"

CUDA Driver Version / Runtime Version 8.0 / 8.0
CUDA Capability Major/Minor version number: 5.0
Total amount of global memory: 2000 MBytes (2096824320 bytes)
(3) Multiprocessors, (128) CUDA Cores/MP: 384 CUDA Cores
GPU Max Clock rate: 1124 MHz (1.12 GHz)
Memory Clock rate: 900 Mhz
Memory Bus Width: 128-bit
L2 Cache Size: 2097152 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 / 3 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
> Peer access from GeForce GTX 1080 (GPU0) -> Quadro K620 (GPU1) : No
> Peer access from Quadro K620 (GPU1) -> GeForce GTX 1080 (GPU0) : No

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 8.0, CUDA Runtime Version = 8.0, NumDevs = 2, Device0 = GeForce GTX 1080, Device1 = Quadro K620
Result = PASS

Now, Go to bandwidthTest folder:
Go to the path: cd ~/645/NVIDIA_CUDA-8.0_Samples/1_Utilities/bandwidthTest
Execute bandwidthTest: ./bandwidthTest
You should get an output such as this:

[CUDA Bandwidth Test] - Starting...
Running on...

Device 0: GeForce GTX 1080
Quick Mode

Host to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers

Transfer Size (Bytes)	Bandwidth(MB/s)
33554432	11693.1

Device to Host Bandwidth, 1 Device(s)
PINNED Memory Transfers

Transfer Size (Bytes)	Bandwidth(MB/s)
33554432	12883.5

Device to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers

Transfer Size (Bytes)	Bandwidth(MB/s)
33554432	231255.7

Result = PASS

Question 1:
a) What is the Host to Device bandwidth you observe on your cluster machine?
b) What is the Device to Host bandwidth you observe on your cluster machine?
c) What is the Device to Device bandwidth you observe on your cluster machine?

Task 2: Get information about the GPU and Run the CUDA Visual Profiler

Step 1: Running the CUDA Visual Profiler

The purpose of this task is to learn how to run the CUDA Visual Profiler.

1. First you need to connect to any of the cluster machines through X. For detailed instructions on connecting to the cluster using X:

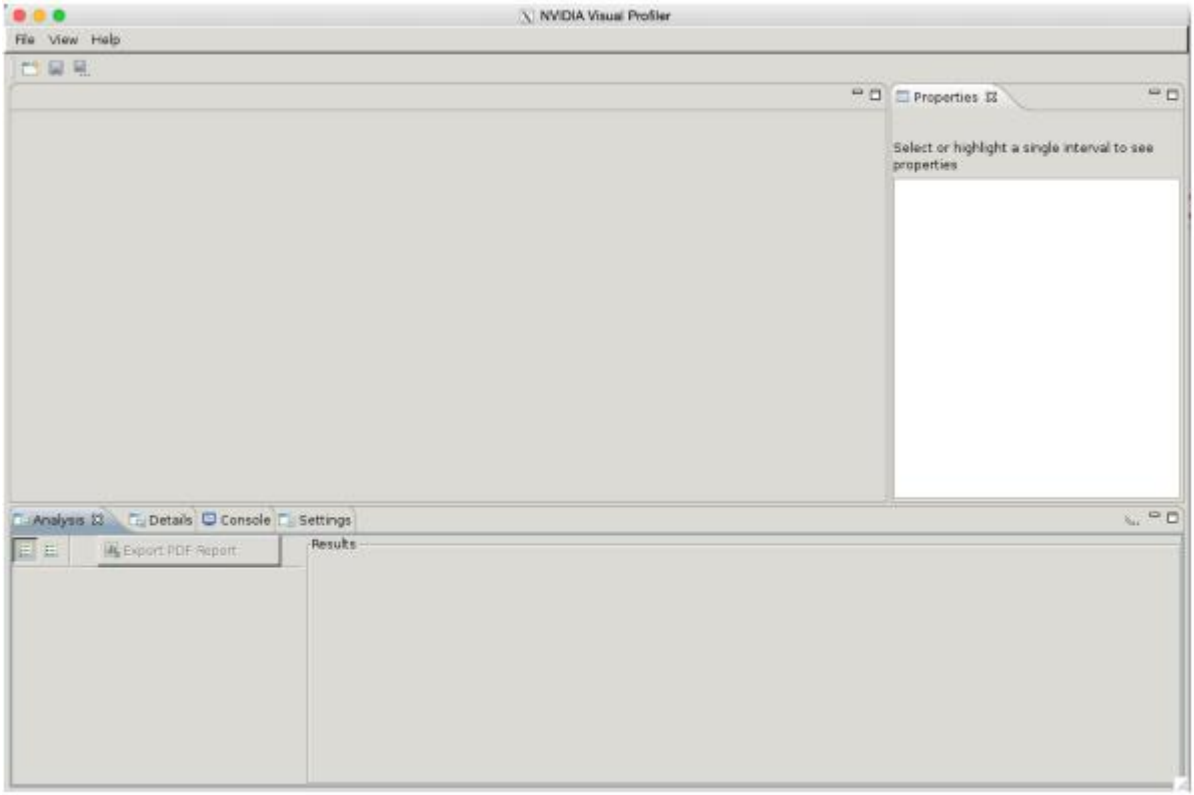
```
ssh –X <your_andrew_id>@ghcXX.ghc.andrew.cmu.edu
```

Warning! Your local machine needs a X server. Linux should have it by default, OS X probaby needs XQuartz and Windows users can look at Xming or Cygwin. On successful connection you should just see the prompt as usual

2. You should get your command prompt if you successfully logged in
3. Run the visual Profiler by typing in this command

```
$computeprof &
```

Now you should see the visual profiler GUI on your screen.



If you get an error message such as
computeprof: cannot connect to X server localhost:11.0

logout and login with –X again!

Step 2: Questions on CUDA and GPUs

Please answer the following questions about the GPU device (GTX 1080) on any of cluster machines

You may find the CUDA Programming Guide very useful for answering these questions:

http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

Question 1:

a) What is the maximum number of threads per block?

b) How many threads are in a warp?

c) What is the maximum number of threads per Streaming Multiprocessor?

d) What is the maximum number of warps per block?

e) What is the maximum number of warps per Streaming Multiprocessor?

Question 2:

d) What is the size of the register file per Streaming Multiprocessor?

e) The shared memory is divided into 16 memory banks, true or false?

f) What is the size of the global memory on the device?

g) What is the total amount of shared memory per block?

Question 3:

a) What is the maximum number of floating point operations per thread per cycle?

b) One day, you read a paper where the authors claim that they have implemented an amazingly fast Matrix to Matrix Multiplication algorithm in CUDA for the GPU devices installed on the cluster machines. The paper claims that the fast algorithm can achieve performance of 1.5 Teraflop/s for multiplying many couples of 32x32 element matrices of single precision floating point numbers on a single GTX1080 GPU. Unfortunately they won't share the code. Can you tell if what the paper's claim is possible or not?

c) What is the peak performance of a single GTX1080 GPU?

Question 4:

a) What is the total amount of L2 cache?

b) The total amount of shared memory is bigger than the total amount of register file on the GPU, true or false?

c) The number of blocks that can executed simultaneously on one Streaming Multiprocessor is restricted only by the amount of shared memory each block allocates, true or false?

Task 3: Getting Started on the CUDA Matrix Multiply

In the code repository, you will find a set of files, of which you will only optimize a few of the CUDA files.

```
matrix_mul:
    Makefile
    cuda
        Makefile
        matrix_mul.cu  <<== To optimize
        matrix_mul.h
        tests.cpp
    omp
        Makefile
        matrix_mul.cpp
        matrix_mul.h
        tests.cpp
    sequential
        Makefile
        matrix_mul.cpp
        matrix_mul.h
        tests.cpp
    tests
        testutil.h

kmeans
    LICENSE
    Makefile
    README
    benchmark.sh
    cuda_io.cu
    cuda_kmeans.cu   <<== To optimize
    cuda_main.cu
    cuda_wtime.cu
    file_io.c
    gmon.out
    go
    kmeans.h
    omp_kmeans.c
    omp_main.c
    sample.output
    seq_kmeans.c
    seq_main.c
    wtime.c
    Image_data
```

Question 1:

a) In your matrix_mul_01.dat test file for CUDA, add one more test case for matrices of size 33x33. Then compile and run the code. Does the code execute normally for this test case? If yes, what kind of performance do you get for this test case? If not, explain why the code fails in this case.