Digital Systems

EEE4084F


# Practical 2: Programming a GPU with NVIDIA CUDA

**Introduction**

GPUs are an example of a special purpose processor, designed to carry out the specific task of rendering graphics for computer games. They started out as very expensive devices, aimed at scientific computer vision and photography, but have become low cost devices for the commodity computer games market. The need to render volumes of graphical data has lead to a highly parallel architecture. However, the arithmetic precision required, and the desire not to include operations more easily carried out on the main processor of the computer system makes them very specialised.

In the first part of this practical you will read up about the CUDA programming environment, with background from the lectures and notes on the NVIDIA family of GPUs. You will then view a sample CUDA program and describe its operation.

The second part of the practical will require you to develop a simple CUDA application suited to parallel execution on the GPU, tested in simulation. Having completed the development of your application, you will test and characterise your application on a real GPU.

## Part A: Getting started with CUDA

Before beginning the prac, ensure that you run the **gpucomputingsdk_3.2.16_linux.run** script in the bluelabuser home directory.

The folder Prac02a provides a complete working CUDA example program, which implements a scalar product algorithm. The Prac02a folder contains the following files:

| | |
|---|---|
| **Makefile:** | for compiling the program |
| **prac02a.cu:** | Host side source code |
| **prac02a_kernel.cu:** | GPU side source code |
| **prac02a_gold.cpp:** | Serial sample solution for comparison |

This is the standard directory structure for most of the CUDA SDK examples and will be used in this practical. The binary is compiled to **release** directory in the prac folder, after typing the **make** command in the prac directory.

The Prac02a program starts by allocating dynamic memory for a set of vectors, namely h_A, h_B, h_C_CPU, h_C_GPU. The vectors h_A and h_B will be used to contain inputs vectors. The vector h_C_CPU is an output vector that will hold the result computed by the CPU; the h_C_GPU will hold the result computed by the GPU. After allocating the vectors, a large CSV file is read from the disk. This CSV file contains two columns; the first column is read into h_A and the second column into h_B. After this, the h_A and h_B vectors are copied over to the GPU memory (using cudaMemcpy),

the GPU program activated (the complicated prac02GPU statement), the CPU waits for the GPU to complete and then reads back the results into the h_C_GPU vector.

When viewing **prac02a.cu** take note of how the timing is done, you will need to time your code in Part B. ***Note:*** when timing code, the timed portion should include setting up the GPU, copying data to and from the GPU memory, this is important for a true comparison of speed up.

Also ensure you understand how the results generated by the CUDA solution are compared with the serial solution, this is also required for Part B.

## Part B: Design and develop and simple parallel application for a GPU using CUDA

You are given the following 'gold' algorithm (i.e. linear implementation in C++) which you need to translate into a GPU version, making it as efficient as you can.

Polynomial generator – creates a big array of N floats, given as input constant multipliers, and a range (starting x value to ending x value). The step value for incrementing x needs to be (ending x - starting x) / N.

See **prac02b_gold.cpp**

The output of this file is an array that represents samples from a polynomial.

Your task is to think of how you can make the algorithm more efficient for CUDA.

When you think you have a working solution, and are ready to finish with this prac, call the tutor for your short oral and to check your result.

***Short Report***

Write a short report (it can be one or two pages, and does not need to be too fancy; e.g. point-form) discussing your observations and results. You've probably already used one set of input data provided with the gold example for the **N**, **consts**, and **range** arrays. Include another set of input data for a second comparison between the serial code and GPU code results. Report on the timing and average error between the gold and CUDA version for both the first and second data sets.

For example, the first data set is:

```
float consts[] = {0.8, -1.5, 0.5, 0.2};
float range[]  = {0.0, 2.0};
polygenCPU(y,consts,range,4096);  // N = 4096

CPU time = XXX (replace with your results)
CGU time = XXX
ERROR = XXX
```

The short report can be done as a take-home task; if you write down the inputs used and error figures, you shouldn't need to run the program again to finish the report.

**Hand in procedure:** Submit your report using the Prac02 VULA assignment.