GPU programming with CUDA: Practical exercises 2

From materials developed by Alan Gray and James Perry, EPCC, The University of Edinburgh

These exercises should be conducted on ARC3.

1. CUDA programs should have the extension **.cu** ; for example **mycode.cu**
2. On ARC3, the CUDA tools are available in the cuda module; **module load cuda**
3. Codes can be **compiled** on any node with the nvcc compiler
   (eg. nvcc mycode.cu -o mycode.x) but can only be **executed** on a GPU node
4. These small exercises should be submitted through the batch scheduler using a submission script like this:

```
#$ -cwd -V
#$ -l h_rt=00:10:00
#$ -l coproc_p100=1
module load cuda
./mycode.x
```

You can get the templates:

```
cp /nobackup/issmcal/learnCUDA.tgz .
tar -zxvf learnCUDA.tgz
```

and for the solutions:

```
cp /nobackup/issmcal/learnCUDAsolutions.tgz .
```

And then:

```
cd learnCUDA
```

Practical 1: Getting started
Example 1:
A simple introductory exercise is available in the intro/src folder. This contains a template CUDA file that you will edit. The template source file is clearly marked with the sections to be edited, e.g. /* Part 1A: allocate device memory */

Please see below for instructions. Where necessary, you should refer to the CUDA C Programming Guide and Reference Manual documents available from http://developer.nvidia.com/nvidiagpucomputingdocumentation

Start from the intro.cu template.

Part 1A: Allocate memory for the array on the device: use the existing pointer d_a and the variable sz (which has already been assigned the size of the array in bytes).
Part 1B: Copy the array h_a on the host to d_a on the device.
Part 1C: Copy d_a on the device back to h_out on the host.
Part 1D: Free d_a.

So far the code simply copies from h_a on the host to d_a on the device, then copies d_a back to h_out, so the output should be the initial content of h_a — the numbers 0 to 255.

Compile, submit and inspect the output.

Example 2:
Edit the intro.cu template to actually run a kernel on the GPU device.

Part2A: Configure and launch the kernel using a 1D grid and a single thread block (NUM_BLOCKS and THREADS_PER_BLOCK are already defined for this case).

Part2B: Implement the actual kernel function to negate an array element as follows:

```
int idx = threadIdx.x;
d_a[idx] = 1 * d_a[idx];
```

Compile and submit the code as before.

This time the output file should contain the result of negating each element of the input array.

Because the array is initialised to the numbers 0 to 255, you should see the numbers 0 down to 255 in the output file this time. This kernel works, but since it only uses one thread block, it will only be utilising one of the multiple SMs available on the GPU.

Multiple thread blocks are needed to fully utilize the available resources.

Part 2C: Implement the kernel again, this time allowing multiple thread blocks. It will be very similar to the previous kernel implementation except that the array index will be computed differently:

```
int idx = threadIdx.x + (blockIdx.x * blockDim.x);
```

Remember to also change the kernel invocation to invoke negate_multiblock this time. With this version you can change NUM_BLOCKS and THREADS_PER_BLOCK to have different value as long as they still multiply to give the array size.

Practical 2: Optimising an Application- Image Reconstruction

2.1 Introduction

This exercise, in the reverse_edge folder, involves optimising a simple image processing algorithm using CUDA for C. It is an iterative reverse edge detection code that, given a data file containing the output from running a very simple edge detector on an image, reconstructs the original source image.

You are given a CUDA source file containing all of the necessary support code and a working, but slow, GPU implementation of the algorithm. Your task is to apply various optimisations to the code to improve performance, as described below.