|epcc|

Learn CUDA in an Afternoon: Hands-on Practical Exercises

Alan Gray and James Perry, EPCC, The University of Edinburgh

Introduction

This document forms the hands-on practical component of the *Learn CUDA in an Afternoon* tutorial available here: `www.epcc.ed.ac.uk/online-training/learnCUDA`.

It is assumed that you have access to a computer with a CUDA-enabled NVIDIA GPU and a unix operating system. For Windows operating systems, you will just have to adapt the compilation commands etc (please refer to NVIDIA documentation).

The first exercise will get you started with your first CUDA code. The second exercise uses, as a starting point, an existing CUDA application that performs poorly. You will go through several optimization steps, measuring the performance benefits of each. You will see the importance of minimizing data transfer, enabling coalesced memory access, and tuning the parallel decomposition.

To get the template files:

```
wget  https://agray3.github.io/files/learnCUDA.tar.gz
```

(or download using your web browser from this address)

Then:

```
tar zxvf learnCUDA.tar.gz
cd learnCUDA
```

For each of the exercises there exists a subfolder `src`, containing the exercise templates, and `solution_src`, containing sample solutions.

1 Getting Started with CUDA

1.1 Introduction

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/nvidia-gpu-computing-documentation`

1.2 Copying Between Host and Device

The introductory exercise is a simple CUDA code that negates an array of integers. It introduces the important concepts of device-memory management and kernel invocation. The final version should copy an array of integers from the host to device, multiply each element by -1 on the device, and then copy the array back to the host.

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`.

To compile, run the command:

```
nvcc -o intro intro.cu
```

To run:

```
./intro
```

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.

1.3 Launching Kernels

Now we will 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 run 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);
```

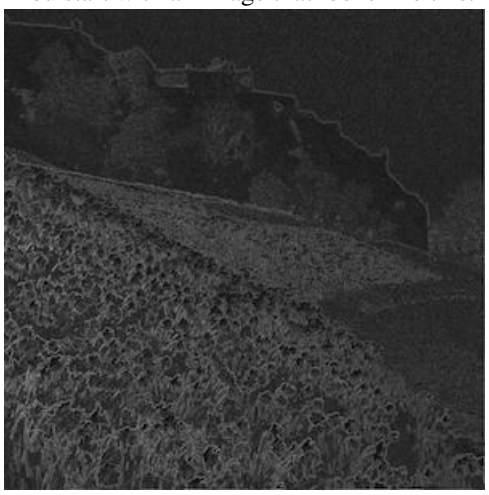
Remember to also change the kernel invocation to invoke `negate_multiblock` this time. With this version you can change `NUM_BLOCKS` and `THREADS_PER_BLOCK` to have different values — so long as they still multiply to give the array size.

2 Optimising an Application: Image Reconstruction

2.1 Introduction

This exercise involves performing a series of optimizations to an existing CUDA application.

You start with an image that looks like this:



Which has been generated from the original:

On the left is an image of Edinburgh Castle, processed such that the edges between light and dark areas replace the original picture. Your job is to reconstruct the initial image. This is an artificial thing to do, but it mimics many scientific applications (e.g. that solve systems of PDEs) since the algorithm is iterative, requiring many successive *stencil* operations. Each pixel of the new *image* is generated based on it's neighboring pixel values and the original *edge* data by repeatedly performing the following update:

$$image_{i,j} = (image_{i-1,j} + image_{i+1,j} + image_{i,j-1} + image_{i,j+1} - edge_{i,j})/4$$

The more iterations, the better the reconstruction (although for simplicity we work in greyscale rather than colour).

You are provided with a working but slow CUDA implementation of the reconstruction algorithm. First of all, let's compile and run the code. The code is set up to run the algorithm on both the GPU and the CPU. It compares the outputs from the two runs to verify correctness, and then displays timings for each run.

Enter the `reconstruct/src` directory and type `make` to build the code. Then type `./reconstruct` to run the code.

It is not necessary, but if you like you can view the resulting image. If you have the netpbm toolkit installed you can convert it to jpeg format with the commands

```
pgmtoppm white output.pgm > output.ppm
ppmtojpeg output.ppm > output.jpeg
```

Hopefully you will see that the picture is starting to become clearer. As the algorithm is iterative, there is a loop in the `reconstruct.cu` that invokes the kernel N times, where N is defined in `reconstruct.h`: it has default value of 100 and can be increased to increase the quality of the reconstruction, but this will mean a longer runtime and is not necessary for completing this tutorial.

Now it's time to optimise the code and improve on the GPU timing printed when you ran the code, by editing the source code.

Note that a one pixel wide halo region of zeroes is added to each edge of the various image-data arrays; this simplifies the computation as it allows the edge pixels to be treated in the same manner as other pixels. (The edge array, which holds the original edge data, does not have require a halo.)

2.2 Minimizing Data Transfer

A challenge with GPUs and other accelerators is that transferring data between host memory and device memory is often relatively slow. An important optimization technique involves minimizing the amount of data that is transferred between host and device.

Notice that in the main loop in `reconstruct.cu`, the data is copied from GPU memory to host memory and then back to GPU memory at each iteration. This is not in fact necessary; with the exception of the final iteration when the data must be copied back to the host, it is going to be processed on the GPU again in the next iteration. Therefore, we can avoid most of the data copying by simply reversing the roles of the input and output buffers in device memory after each iteration.

In order to do this you will need to:

- remove the `cudaMemcpy` calls from inside the main loop.
- replace them with code to swap the pointers `d_input` and `d_output` (you will need to declare a new temporary pointer).
- add a new `cudaMemcpy` call after the end of the loop to copy the final result back from the GPU to the `gpu_output` buffer in host memory. (Note: remember that the buffer pointers will have been swapped at the end of the loop, so the output from the last iteration will now be pointed to by `d_input`!).

Once you have made these changes, compile and run the code again by re-executing the corresponding cells above, and take note of the time taken by the GPU version. How does it compare to the previous timing?

2.3 Enabling Coalesced Memory Accesses

The GPU performs best when consecutive CUDA threads access consecutive memory locations, allowing memory coalescing. However, for the kernel in `reverse_kernels.cu`, it can be seen that consecutive threads correspond to consecutive rows of the image, but consecutive memory locations instead correspond to consecutive columns. The threads are not reading from consecutive locations.

- Update the kernel such that the problem is decomposed by column rather than row, i.e. such that each thread operates on a different column of the image (looping over all rows in that column).
- Since the image is perfectly square, you will not need to change the way the kernel is launched.
- How does the performance compare to the previous version?

2.4 Improving Occupancy

You should hopefully have seen a noticeable improvement in performance as a result of the changes you made to reduce data transfers between the host and the device and to enable coalescing. However, the current solution is still sub-optimal as it will not create sufficient threads to utilise all the SMs on the GPU - it has low occupancy.

GPU codes typically run best when there are many threads running in parallel, each doing a small part of the work. We can achieve this with our image processing code by using a thread for each pixel of the image, rather than for each row or column as before. CUDA supports 1-, 2- or 3-dimensional decompositions. A 2D decomposition maps most naturally onto the pixels of an image.

- Update your both your kernel (in `reconstruct_kernels.cu`), and the code responsible for specifying the decomposition (in `reconstruct.cu`) such that that a 2D decomposition is used.

- The original code uses 256 threads per block in a 1D CUDA decomposition. Replace this with 16 threads in each of the X and Y directions of the 2D CUDA decomposition, to give a total of 256 threads per block. Ensure that the number of blocks is specified appropriately in each direction.
- Ensure that you retain memory coalescing
- Again, measure performance and compare to the previous versions.

2.5 Investigating Grid and Block Sizes

Once you have the 2D kernel working correctly, you can try altering certain parameters and see what effect this has on its performance. In particular, you can investigate the effects of different grid and block sizes. How does changing the grid and block sizes affect the total runtime?