Learn CUDA in an Afternoon: Hands-on Practical Exercises

Paul Richmond and Michael Griffiths, CUDA Research Centre, The University of Sheffield

Material Developed by

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 will run the examples using the Central HPC facility at The University of Sheffield ('Iceberg').

You may also run the course examples on a different computer with a CUDA-enabled NVIDIA GPU and a unix operating system. For Windows operating systems, it is necessary 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.

Connecting to the HPC Service at The University of Sheffield

We recommend using the Internet Explorer browser which can be started from the desktop icon on Managed Windows.

We will be accessing GPUs on the iceberg high performance computer. To access this system you have been provided with an iceberg username and password (see printed lists). Please use this to login to the MyApps portal. Open a web browser and login to iceberg using the myApps portal, to do this open the link
http://www.shef.ac.uk/wrgrid/using/access/browser

From this page, click on the link "connect via myApps portal".

There are instructions on this page (http://www.shef.ac.uk/wrgrid/using/access/browser) of what to do.
Note if that if you are prompted to update Java then we recommend that on managed windows at this time you do not update Java.

1. A warning dialog will appear click run
2. A security warning will ask if you wish to accept the security certificate click Accept and then yes.
3. The global desktop should start and you will have a window with a tree of clickable items on the left hand pane. Click Iceberg applications and then click interactive job.

The preceding instructions will leave you with a command terminal and by typing the following commands you can start GPU programming using the provided course examples by typing the following sequence of commands.

1. **mkdir shefgpucourse**

2. **cd shefgpucourse**
3. **tar –zxvf /usr/local/courses/learnCUDA.tar.gz**
4. **module add /libs/cuda/4.0.17**

Please make sure you are running an interactive session and that you are not on the login node of iceberg.

To edit files you may use the editor nedit or gedit, simply type the command **nedit** or **gedit**. You should now be ready to start the first practice session

Alternatively you can get the template files:
wget http://gpucomputing.sites.sheffield.ac.uk/training-and-events/learnCUDA.tar.gz?attredirects=0&d=1

To get the solutions
wget http://gpucomputing.sites.sheffield.ac.uk/training-and-events/learnCUDAsolutions.tar.gz?attredirects=0&d=1

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, 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 will be 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.

2.2 The Algorithm

You will be given a data file that represents the output from a very simple edge-detection algorithm applied to a greyscale image of size M _N. The edge pixel values are constructed from the image using

$edge_{i;j} = (image_{i-1;j} + image_{i+1;j} + image_{i;j-1} + image_{i;j+1} - image_{i;j})$

If an image pixel has the same value as its four surrounding neighbours (i.e. no edge) then the value of edgei;j will be zero. If the pixel is very different from its four neighbours (i.e. a possible edge) then edgei;j will be large in magnitude. We will always consider i and j to lie in the range 1; 2; : : :M and 1; 2; : : :N respectively. Pixels that lie outside this range (e.g. imagei;0 or imageM+1;j) are simply considered to be set to zero.

Many more sophisticated methods for edge detection exist, but this is a nice simple approach.
The exercise is actually to do the reverse operation and construct the initial image given the edges. This is a slightly artificial thing to do, and is only possible given the very simple approach used to detect the edges in the first place. However, it turns out that the reverse calculation is iterative, requiring many successive “stencil” operations each very similar to the edge-detection calculation itself:

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

The fact that calculating the image from the edges requires a large amount of computation makes it a

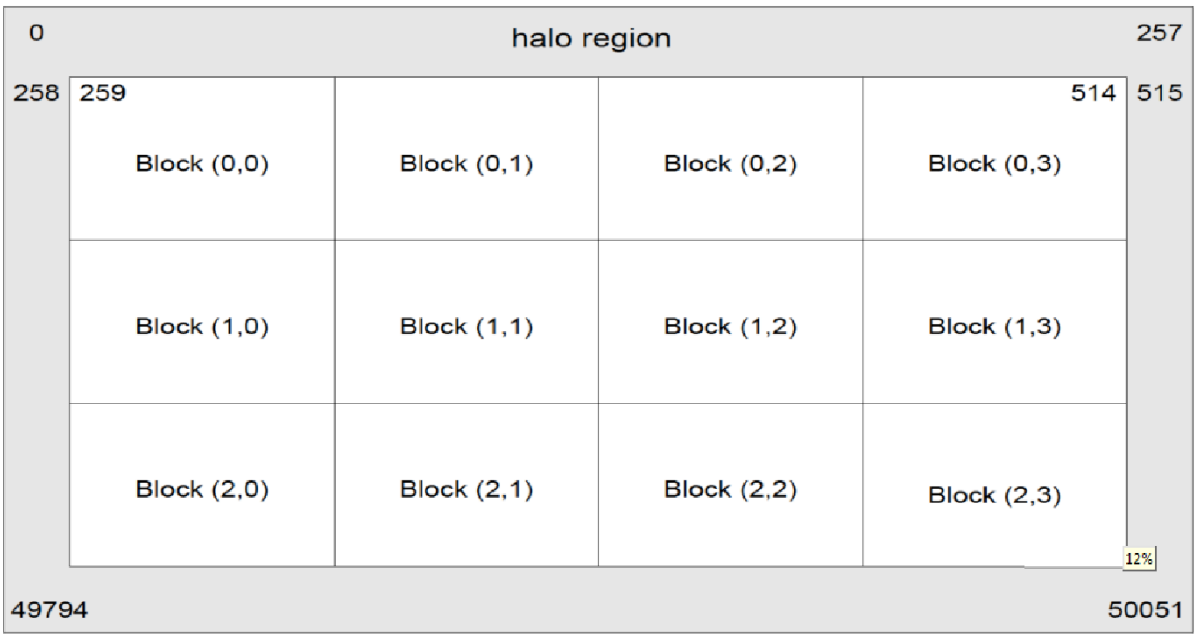
much more suitable program than edge detection itself for the purposes of GPU acceleration and performance measurement.
As an aside, this inverse operation is also very similar to a large number of real scientific HPC calculations that solve partial differential equations using iterative algorithms such as Jacobi or Gauss-Seidel.

2.3 Optimising the Algorithm for CUDA

You are provided with a working CUDA implementation of the reverse-edge-detection algorithm in the reverse.cu file. You will find three kernels in there—one of these (inverseEdgeDetect1D_col) is already complete and functional; the other two will be completed later on in the optimisation process.
There is also some support code for file I/O and timing, and a simple host implementation of the algorithm.

In the host code, the image data arrays are two-dimensional, which maps intuitively onto the two dimensions of the image. A one pixel wide “halo” region of zeroes is added to each edge; this simplifies the computation as it allows the edge pixels to be treated in the same manner as other pixels (note that the edge array, which holds the original edge data, does not have a halo as it does not need one). For the GPU kernel, the data arrays are “flattened” into a single dimension but are otherwise the same format.

Taking a 256 _ 192 pixel image as an example, the total image array size with the haloes added is 258 _ 194 = 50; 052 elements. The first 258 elements (0 to 257) are the top halo. The next 258 (258 to 515) are the top row of the image with left-halo pixel at 258, actual data from 259 to 514, and right-halo pixel at 515. The next 258 elements (516 to 773) are the next-to-top row of the image, and so on as far as the final row (49,536 to 49,793) and the bottom halo (49,794 to 50,051). See below for a graphical illustration of the array layout.



The provided completed kernel, inverseEdgeDetect1D_col, assigns each row of the image to its own CUDA thread. The kernel contains a loop over the columns of the image, determines the element index that it should work on (getting the column from the loop counter and the row from the CUDA thread and block information), and processes that pixel as described above.

As the algorithm is iterative, there is a loop in the top-level program that invokes the kernel several times (by default 10 times). In between the kernel invocations it copies the output from the previous invocation back to the input buffer ready for the next iteration, via a buffer in host memory.

The code is set up to run the algorithm on both the GPU and the host. It compares the outputs from the two runs to verify that they are correct, and then displays timings for each run.

2.3.1 Testing on Iceberg

At this point you may want to compile and run the code. To compile, use the nvcc command:

nvcc -arch=sm_13 -o reverse reverse.cu

To submit your job to the Iceberg backend for execution, use qsub:

qsub job_script.sh

Once your job has executed, its output will appear in a file named something like job_script.sh.o8684 (where the number at the end is different for each job submitted). This includes the time taken for the host and GPU runs of the algorithm, and an indication of whether the results matched or not. (If they didn't match, the values of the elements that differed will be printed out). The output data is written to the file output.pgm by default. You can view this file using the display command to see the reconstructed image. If you increase the number of iterations that the algorithm runs by changing the value of the ITERATIONS constant in the source, the image will be reconstructed more accurately.

2.3.2 The Data Transfer Bottleneck

A problem with GPUs and other accelerators is that transferring data between host memory and device memory is often relatively slow. In many codes this can become a bottleneck, sometimes even cancelling out any performance gains achieved by using the accelerator. An important part of optimising such codes is therefore to minimise the amount of data that is transferred between host and device.

Notice that in the main loop of our code, 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.
- add a new cudaMemcpy call after the end of the loop (in between the two calls to get_current_time())

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, run the code again and take note of the time taken by the GPU version. How does it compare to the previous timing?

2.3.3 Enabling Coalesced Memory Accesses

There is another bottleneck in the current implementation of the algorithm. The GPU performs best when kernel threads are reading from adjacent memory locations, allowing some of the memory reads to be combined; this is known as coalescing. However, in our current code, the threads are reading not from consecutive addresses but from different rows of the image, so no coalescing can take place. This can be improved by decomposing the problem in a different way. Instead of having each thread work on one

row of the image data and loop over the pixels within that row, we can have each thread work on one column of the image and loop over the rows of that column.

This means that the threads will always be reading consecutive memory locations, allowing the reads to be coalesced. This is implemented in the inverseEdgeDetect1D_row kernel, which needs two changes to make it complete:

- calculate the column index (col) for the thread from the CUDA block and thread information
- loop over all the rows of the image

Notice that the body of the loop is identical to that in the original kernel; it is only the computation of the row and column values that has changed.

Try running the code with the new kernel (remember to change the main loop to invoke the new kernel and provide suitable grid and block dimensions for it). How does the performance compare to the previous version?

2.3.4 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 very 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 so this is what we will use.

A 2D version of the kernel is included in the source file — see inverseEdgeDetect2D. It requires two additional lines of code to make it work:

- calculate the column index (col) from the X dimensions of the CUDA thread and block information.
- calculate the row index (row) from the Y dimensions of the CUDA thread and block information.

Notice that the main part of the kernel is identical to the previous two versions and only the row and column computations have changed. Also notice that there is no longer a loop within the kernel, as each thread is now operating on an individual pixel.

Before testing this final version of the code, you will need to change the kernel invocation in the main loop to invoke the 2D version of the kernel. You will also need to initialise the block and grid dimensions as 2D rather than 1D as before.

Observe how the performance of the 2D version compares to the previous 1D versions of the algorithm.

2.3.5 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. These are defined as constants near the top of the source file so they can be easily adjusted. Some suggested sizes to try are given in the table below, and you may also want to experiment with others.

Block Width	Block Height	Threads per Block	Grid Width	Grid height	Time
1	1	1	256	192	
2	2	4	128	96	
4	4	16	64	48	
8	8	64	32	24	
16	16	256	16	12	

Bear in mind that:

- So that the correct number of pixels is processed, the grid and block widths must multiply to give the image width (256), similarly the heights must multiply to the image height (192).
- For Ness, the total block size should cannot exceed 512 threads (although you may wish to investigate what happens when the block size is set to a value larger than this).

How does changing the grid and block sizes affect the total runtime?

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?