GPU Computing Workshop: Hands on exercises (Day 2)

Dr Paul Richmond (University of Sheffield)

Accessing Amazon GPUs and Setup

Navigate to the NVlabs website and begin the CUDA Training course. You will be provided log in details for an Amazon AWS virtual machine instance which has a Kepler series GPU. Log into the instance using ssh (via Putty on windows) with the log in details provided to you. If you are logging in from a Linux machine then use `-l` argument to specify the username.

Get the Starting code from github by cloning the master branch of the handsoncuda_day2 repository from the AccleratedComputing github page. E.g.

```
$git clone
https://github.com/AcceleratedComputing/handsoncuda_day2.git
```

This will check out all the starting code and images for you to work with.

Introduction

For this session we are going to start by improving the performance of an existing CUDA program "`boxblur.cu`". The starting code provided contains an implementation of a simple box blur. The box blur (also known as a box linear filter) is an operation which samples neighbouring pixels of an input image to output an average value. When applied iteratively to an image the box filter can be used to approximate a more complicated Gaussian blur (wiki link). The box blur can be described as follows.

$$Out_{\{x,y\}} = \frac{(In_{\{x-1,y-1\}}+In_{\{x,y-1\}}+In_{\{x+1,y-1\}}+In_{\{x-1,y\}}+In_{\{x,y\}}+In_{\{x+1,y\}}+In_{\{x-1,y+1\}}+In_{\{x,y+1\}}+In_{\{x+1,y+1\}})}{9}$$

Within the implementation provided the box blur has the property that outside of the bounds of the input image values are `0`. The code works for fixed sized square images. An image *'dog.ppm'* is provided in the ppm format and code is provided for image reading and writing. You can use your own image but make sure that the `IMAGE_SIZE` macro is changed to reflect your image size.

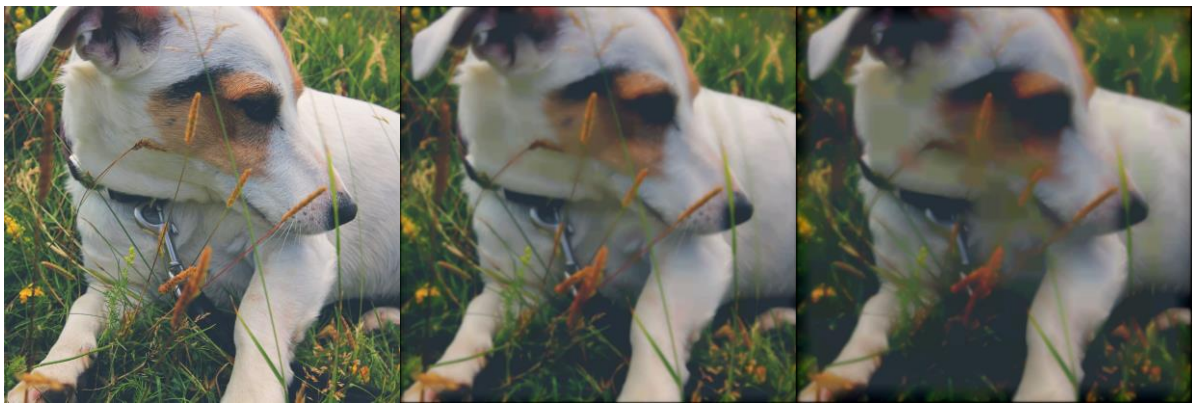

Figure 1 - Result of applying the Box filter for 0, 50 and 100 iterations

Try compiling and running the code and examine the output of the blurred image. Make a note of the execution time reported.

Viewing the Image using Jupyter Notebook

Using the notebook link provided on the nvlabs training session open a link to the notebook on your local machine. The notebook server runs from your GPU instance and can accesses the local files. From the notebook will by default show the contents of the Ubuntu user directory. Navigate to the `handsoncuda_day2` directory and open the PPM Viewer notebook file to start the interactive notebook. From here you can display the image file in your web browser by following the instructions provided.

Exercise 01

The code has a number of inefficiencies. We will first consider the transfer bottleneck. For each iteration of applying the box filter/blur the algorithms performs the following steps

1) Copy the previous iterations (or input) image from the host to the device
2) Apply the box blur GPU kernel
3) Copy the results back to the host and repeat the above.

It is not necessary to copy the results of each filter operation back to the host. We can simply pass the pointer of the previous iterations output as the input for the next iterations. This will drastically reduce memory movements via PCIe. To implement pointer swapping complete the following steps.

1.1 Starting from the code in the `STARTING_CODE` switch case make a copy into the `EXERCISE_01` switch case
1.2 Move the memory copy of the input image outside of the `ITERATIONS` loop so that the host data is copied to the device only once.
1.3 A pointer `d_image_temp` has been defined for you. Use this as a temporary pointer to swap the areas of memory pointer to by `d_image` and `d_image_output` after the box blur kernel is applied. This will
1.4 Move the memory copy of the output image outside of the `ITERATIONS` loop so that the device data is copied back to the host only once. *Note: Be careful that you copy back from the correct device pointer if you have swapped them!*
1.5 Compile and execute your code. Ensure that the variable `exercise` is set to `EXERCISE_01` so that your modified code is executed. Make a note of the execution time. It should be considerably faster than previously.

Exercise 02

The `image_blur_columns` kernel currently has a poor memory access pattern. Let us consider why this is. For each thread which is launched the thread iterates over a unique row of `IMAGE_DIM` pixels to perform the blurring on each pixel. Between each thread this creates a stride of `IMAGE_DIM` between memory loads. CUDA code is much more efficient when sequential threads read from sequential values in memory (memory coalescing). To improve the code we can implement a row wise version on the kernel by completing the following.

2.1 Copy the `image_blur_columns` kernel and call the new kernel `image_blur_rows`.
2.2 Modify the `image_blur_rows` kernel so that each thread operates on a unique column (rather than row of the images). This will ensure that sequential threads read sequential row values from memory.
2.3 Implement the `EXERCISE_02` switch case (by copying the previous one) ensuring that your host code calls your new kernel

2.4 Compile and execute your code. Ensure that the variable `exercise` is set to `EXERCISE_02` so that your modified code is executed. Make a note of the execution time. It should be considerably faster than previously.

Exercise 03

Our previous implementations of the blur kernel have a limited amount of parallelism. There are in total `IMAGE_DIM` threads launched and each of the threads is responsible for calculating a unique row or column. Whilst this number of threads might seem reasonably large it is unlikely that it is sufficient to occupy all of the Streaming Multiprocessors of the device. To increase the level of parallelism and improve the occupancy it is possible to launch a unique thread for each pixel of the image. To implement this complete the following steps.

3.1 Make a copy of the `image_blur_rows` kernel and call it `image_blur_2d`. Modify the new kernel so that the `x` and `y` locations are determined form the thread and block index. You can then remove the row loop as the kernel is responsible for calculating only a single pixel value.
3.2 Implement the `EXERCISE_03` switch case (by copying the previous one). You will need to change the block and grid dimensions so that they launch `IMAGE_DIM`^{2} threads in total.
3.3 Compile and execute your code. Ensure that the variable `exercise` is set to `EXERCISE_03` so that your modified code is executed. Make a note of the execution time. It should be considerably faster than previously.

Exercise 04

It is possible to improve the code further by using shared memory and there are a number of ways in which this could be implemented. To avoid branching the best solution is to remap threads within the block to a shared memory tile which is larger than thread block (18x18 rather than 16x16). Each thread can then load a consecutive value into shared memory. The entire shared memory block can be loaded with *at most* two loads per thread. Some threads will not be required for the second load. Check out the solutions and read through the implementation and comments for exercise 4. Compile and execute the code. Ensure that the variable `exercise` is set to `EXERCISE_04` so that the correct version of the modified code is executed. Make a note of the execution time. It should be slightly faster than the previous version.

Exercise 05 (Additional Exercise)

For this exercise we are going to optimise a new piece of code which implements a simple ray tracer. We will explore how changing the different types of memory affect performance. The ray tracer is a simple ray casting algorithm which casts a raw for each pixel into a scene consisting of sphere objects. The ray checks for intersections with the spheres, where there is an intersection a colour value for the pixel is generated based on the intersection position of the ray on the sphere (giving an impression of forward facing lighting). For more information on the ray tracing technique read Chapter 6 of the CUDA by Example book which this exercise is based on. Try compiling and executing the starting code “raytracer.cu” and examining the output image (`output.ppm`).

The initial code places the spheres in GPU global memory. We know that there are two good options for improving this in the form of constant memory and read only memory. Implement the following changes.

1.1 Create a modified version of ray tracing kernel which uses the read-only data cache (`ray_trace_read_only`). You should implement this by using the `const` and `__restrict__` qualifiers. Calculate the execution time of the new version alongside the old

version so that they can be directly compared: You will need to also create a modified version of the sphere intersect function (`sphere_intersect_read_only`).

1.2 Create a modified version of ray tracing kernel which uses the constant data cache (`ray_trace_const`). Calculate the execution time of the new version alongside the two other versions so that they can be directly compared.

1.3 How does the performance compare? Is this what you expected and why? Modify the number of spheres to complete the following table. For an extra challenge try to do this automatically so that you loop over all the sphere count sizes in the table and record the timing results in a 2D array.

Sphere Count	Normal	Read-only cache	Constant cache
16			
32			
64			
128			
256			
1024			
2048			

Exercise Solutions

The exercise solutions are available from the solution branch of the repository. To check these out either clone the repository using the branch command to a new directory as follows;

```
$git clone –b solution
https://github.com/AcceleratedComputing/handsoncuda_day2.git
```

Alternately commit your changes and switch branch

```
$git commit –m “my local changes to src files”
$git checkout solution
```

You will need to commit your local changes to avoid overwriting your changes when switching to the solutions branch. You can then return to your modified versions by returning to the master branch.