University of the Witwatersrand

Assignment

COMS4040: HPC

Image Convolution

Author:
Liam Pulles
Student No: 855442

March 31, 2017

1 Introduction

The purpose of this assignment is to design and evaluate the use of various GPU memory types in a set of CUDA Image Convolution programs. I shall detail the design methodology, performance, and analysis of the performance of these programs.

2 Design

2.1 Serial

The serial design is the most elementary, and is a large basis for the other implementations.

Important notes:

- The image is loaded using standard SDK functions.
- The kernel or convolution mask is loaded and stored in a dynamic array.
- I decided that to avoid doing a check for each pixel in the main convolution algorithm (to verify the range of it's neighbors), it would be more efficient to enlarge the input image with a black border (thus when pixels on the edge of the "original" image are accessed, they simply fetch neighbors as 0). I did this by creating a larger region in memory (to include borders) and to copy the old image pixel by pixel into the correct locations. I also needed to initialize the border pixels to 0.
- I created a region in memory for the result, having the size of the original input image,

2.2 Naive Approach

The naive approach extends the serial approach by allocating the host memory as in the serial implementation, copying those arrays to the GPU's memory, executing the kernel function with the convolution algorithm (working in global memory) and then copying the result in the device back to the host memory.

2.3 Shared Memory Approach

The shared memory approach extends the naive approach by creating a more or less block (as in CUDA thread block) sized region of memory to store the image data, so that threads local to the block only need read from that shared memory. It is not exactly block sized as the blocks need to include a border of extra pixels to account for the kernel operating on the edge of the block. Implementing this accounts for the majority of the kernel function.

I tried to distribute the work of setting this border as widely as possible among threads, while also attempting to minimize conditional checks, as can be seen in the code.

2.4 Constant Memory Approach

The constant memory approach extends the naive approach by moving the kernel in dynamic memory on the host to constant memory on the GPU. It is otherwise largely similar.

2.5 Texture Memory Approach

The texture memory approach extends the naive approach by binding the input image to a texture set to optimize for 2 dimensional locality. It is otherwise largely similar.

3 Performance

3.1 Results

Firstly, it is clear that the serial performance is significantly worse than the performance of any of the CUDA implementations (See Performance Graphs).

It is also clear that the difference in performance for the CUDA implementations is not that high on the 3x3 kernel, and somewhat erratic on the 5x5 kernel (See Performance Graphs).

3.2 Number of Floating Point Operations

I shall assume that we wish to count the effective number of Floating Point Operations - that is the operations which are directly involved in the convolution algorithm, and which are independent of implementation.

The number of floating point operations in the main convolution algorithm is given as follows:

$$FLO = 2 \times imagewidth \times imageheight \times kernelwidth \times kernelheight \quad (1)$$

This arises from the fact that we need to do one multiplication and one addition for each element in the kernel, for each pixel in the image. If we presume that the kernel is square, than this reduces to:

$$FLO = 2 \times imagewidth \times imageheight \times kernelwidth^2 \quad (2)$$

3.3 Global Memory Reads

In general, we need one unique element in the image array for each multiplication addition, as well as the corresponding element in the kernel array. It is reasonable to presume however that the relatively tiny and regularly used kernel array will stay in cache for the duration of the convolution process, and thus can be discounted. Then not accounting for persistence of the image pixels in the cache, the number of global memory reads for the main image convolution will be FLO/2. This will be similar for the other CUDA approaches, excluding the shared memory implementation...

In the shared memory implementation, we only read global memory for the kernel and for setting the shared memory. However, since the shared memory for each block is slightly larger than a block, the number of global memory reads at this stage isn't quite the number of pixel in the image. It would be the size of the image, plus the number of additional border pixels per block times the number of blocks. Again, assuming the kernel is immediately put into cache and remains there, this is a fair approximation.

3.4 GFLOPS

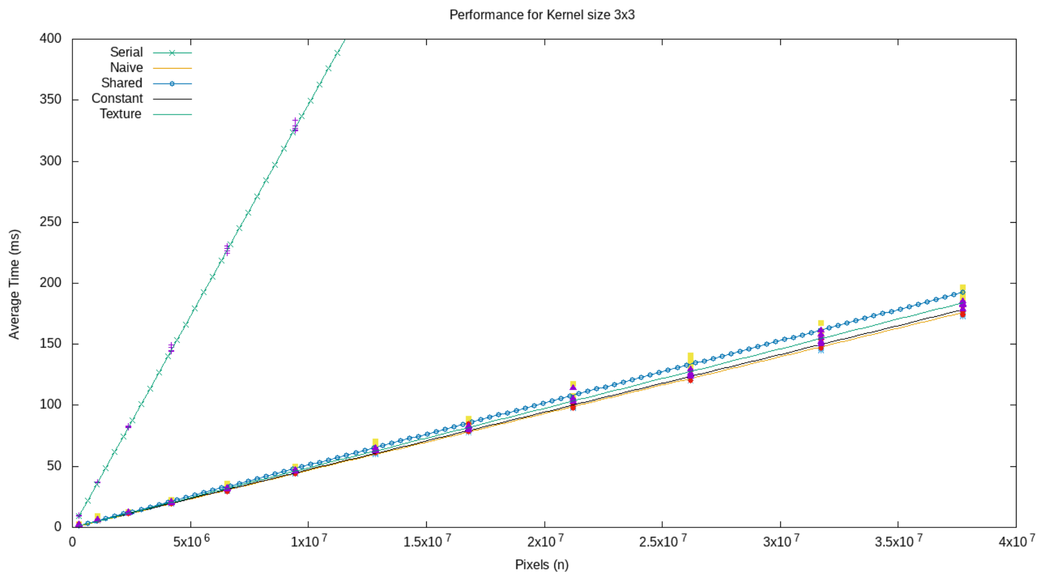
For the parallel implementations, the GFLOPs ranged between 1.2 and 1.9 for the 3x3 kernel, increasing with larger image size, and 3.6 to 5.0 for the 5x5 kernel, being maximum in the middle range of image sizes.

For the serial implementation, the GFLOPs ranged between 0.23 and 0.26 for the 3x3 kernel,increasing with larger image size, and 0.3 to 0.32 for the 5x5 kernel, staying relatively constant with larger image size.

3.5 Overhead

As can be seen from the overhead graphs, the overhead scales more or less linearly with the size of the image.

Figure 1: 3x3 Kernel Performance



4 Discussion

4.1 Serial

The serial implementation tends towards a constant GFLOPs rate with increasing image and kernel sizes, indicating that the overhead increases linearly.

Figure 2: 5x5 Kernel Performance

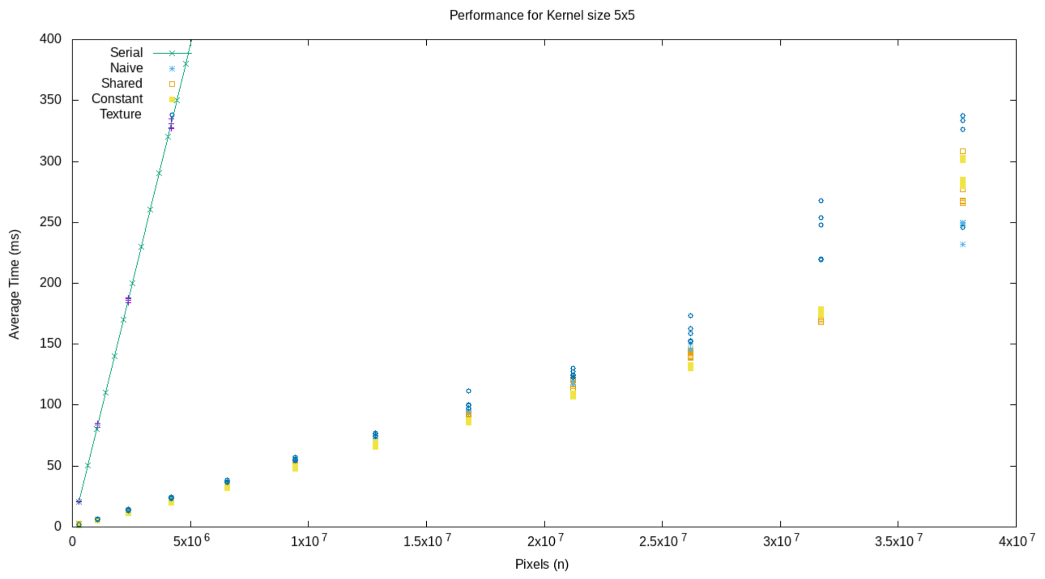
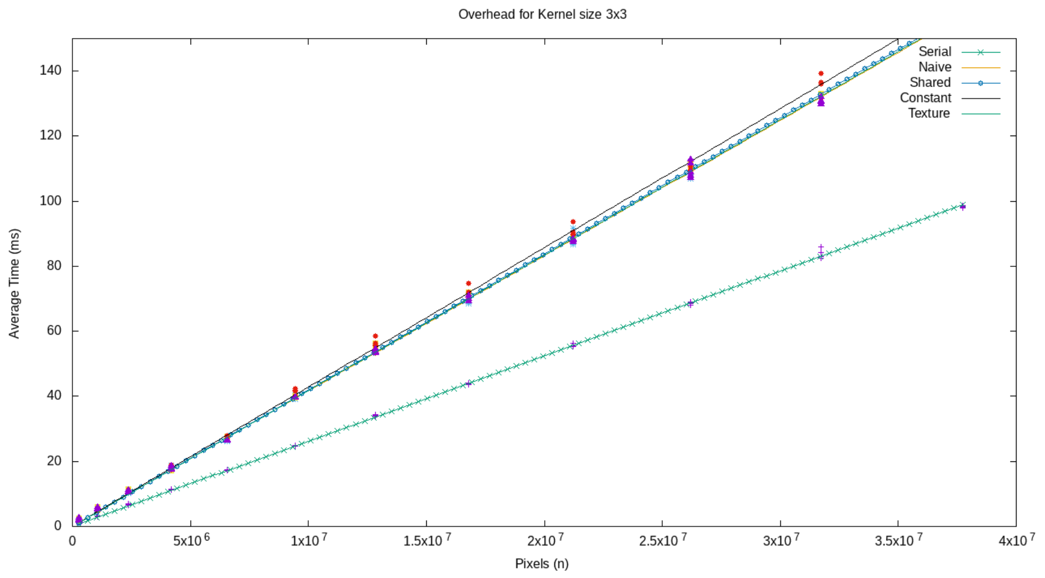


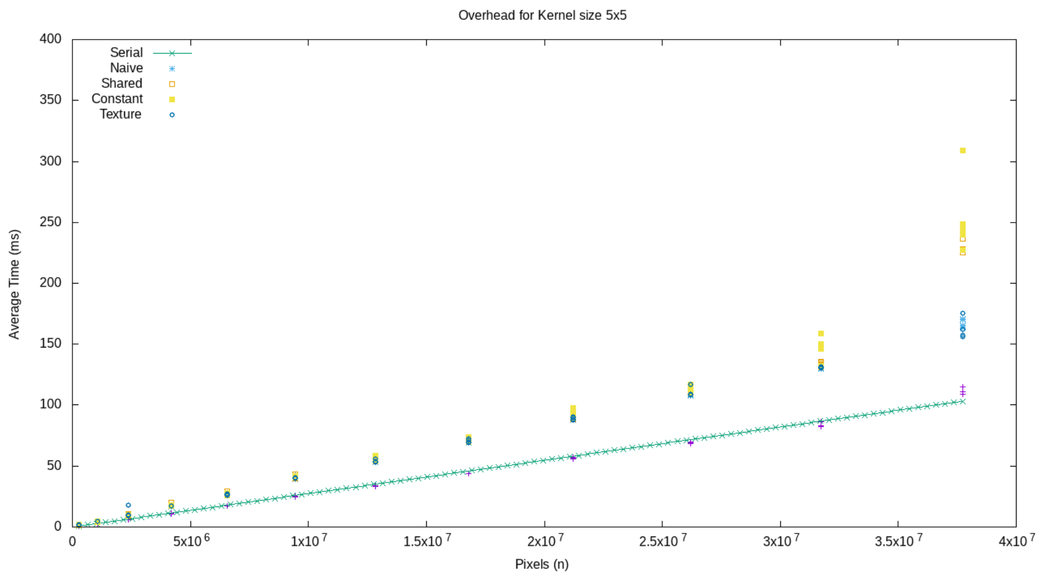
Figure 3: 3x3 Kernel Overhead



4.2 Parallel Implementations

The parallel implementations all appear to have performed very similarly. The lack of performance increase seems to point to the overhead dominating

Figure 4: 5x5 Kernel Overhead



the processing time of the programs - and this is indeed enforced by the graphs. While there are a number of a ways to speed up the main convolution, in the hopes of reducing global memory reads, it is entirely outweighed by the cost of copying and allocating the necessary arrays.

It is worth pointing out here that this could be due to my design choice in copying the image into a larger image with borders. Setting all of the new values is a lengthly process, directly proportional to the size of the image, and so perhaps the better route would have been to use conditional checks in the convolution function instead.