COMS4040A & COMS7045A
Assignment 2 – Report

Kenan Karavoussanos
1348582
Computer Science Honours

April 14, 2019

1 Introduction

The Convolution operation has important applications in the fields of mathematics, computer science and electrical engineering amongst others. The convolution operation in simplified terms is the response of a system when given some input.

In the context of Image Processing, the discrete convolution operation is used for image filtering. This operation is computationally expensive however, due to the fact that each computation in this operation is independent of others, this operation is a very good candidate for parallelization. There is a significant amount of input data overlap between each of these computations. As such, data sharing between these computations can be exploited to improve the performance of the parallel convolution algorithm. The purpose of this work is to implement image convolution using CUDA and evaluate the efficacy of using the different CUDA memory hierarchies.

The report contains a discussion of the different CUDA memory hierarchies, a description of the design choices made during implementation, as well as a presentation and discussion of the performance of each parallel algorithm.

2 CUDA Memory Hierarchies

This section provides an overview of the various CUDA memory hierarchies. The main reference for this section is The CUDA C Programming Guide[2].

2.1 CUDA Architecture

The following diagram shows the CUDA Architecture. This diagram will be used as reference in the discussion to follow.

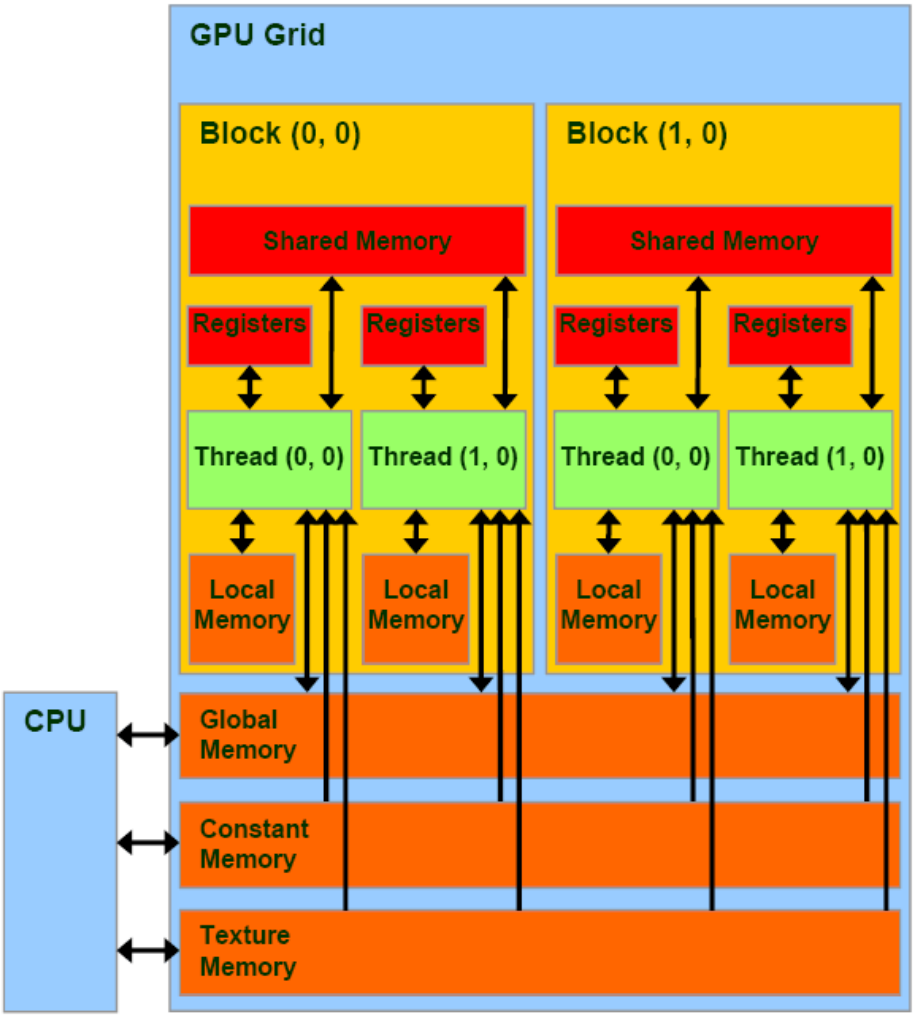

Figure 1: CUDA Architecture

2.2 Global Memory

Global memory resides in Device RAM (DRAM). This is memory is named as such due to the fact that the device and host can both modify this memory i.e the scope of access is global to device and host. Memory transactions are restricted to 32,64 and 128-bytes at a time and only memory addresses that are a multiple of these sizes can be accessed. When a warp of threads executes a memory transaction, it will coalesce the memory accesses of each thread according to the locality of the memory addresses and the size of each access. This is done to maximize throughput. As can be seen in figure 1, global memory (and DRAM) is off-chip so the latency when compared to local and shared memory is higher.

2.3 Constant Memory

Constant Memory is read-only memory that resides in DRAM. It is cached in a read-only *constant cache*. It is a small memory on the order of 64kB. If the memory accesses are coalesced properly then the memory transaction can be as fast as the on-chip registers. The contents of constant memory can only be changed prior to the kernel launch, after which it becomes read-only.

2.4 Texture Memory

Texture memory resides in DRAM and is cached in the *texture cache*. On a cache miss, there is only one read access to global memory. The texture cache is optimized for *2D spatial locality* so if threads access memory addresses that are close in 2D, the performance of texture memory will be improved.

2.5 Shared Memory

Shared memory is on-chip memory. All threads in a block have access to the same shared memory address spaces and threads in other blocks do not have access to this memory. However, due to this sharing, careful consideration must be taken to synchronize threads to avoid race conditions. This memory has much higher bandwidth and lower latency than the other memory types due to it being on-chip.

3 Design Choices

This section describes the design choices for each parallel algorithm as well as the implications of each choice.

3.1 General

It was decided that the parallel algorithms would use boundary checking instead of padding the image with zeros to treat halo cells as zero values. This was decided as the low computational cost boundary checks would be less expensive than the overhead of padding the image.

3.2 Naive/ Global Memory Parallel Approach

This algorithm is the most basic approach to parallelization as such there were not many design choices to be made. However, it was decided that for this approach(and all others except texture memory) the image array would be linearized for simplicity of the code as well as the fact that 1D addressing is faster than 2D addressing.

3.3 Constant Memory

Due to the limited size of constant memory, it was concluded that only the filter can reside in constant memory. Another approach would be to load a subset of the image into constant memory as well however this would complicate the code significantly and as such was decided against.

3.4 Texture Memory

The address mode used for this algorithm was the *Border* mode. This was to satisfy the requirement that the halo cells should be treated as zeros. The texture memory was not normalized as the image gray levels are already normalized as well as the fact that normalization requires two extra floating point operations per texture fetch. It was also decided that 2D texture references would be used for the simplicity of the code.

3.5 Shared Memory

Due to the fact that, for the naive/global memory approach, the number of global memory accesses to filter pixels and image pixels is equal and tiling significantly complicates code and has some added overhead. It was decided that the filter pixels would be stored in shared memory only. This approach still produced significant performance gains compared to the naive approach. One could argue to place both inside shared memory but due to the small size of shared memory both could not fit without significant complication of the code.

4 Results

4.1 Figures

The following section contains graphs illustrating the performance of the various parallel algorithms

4.2 Time vs Image Size

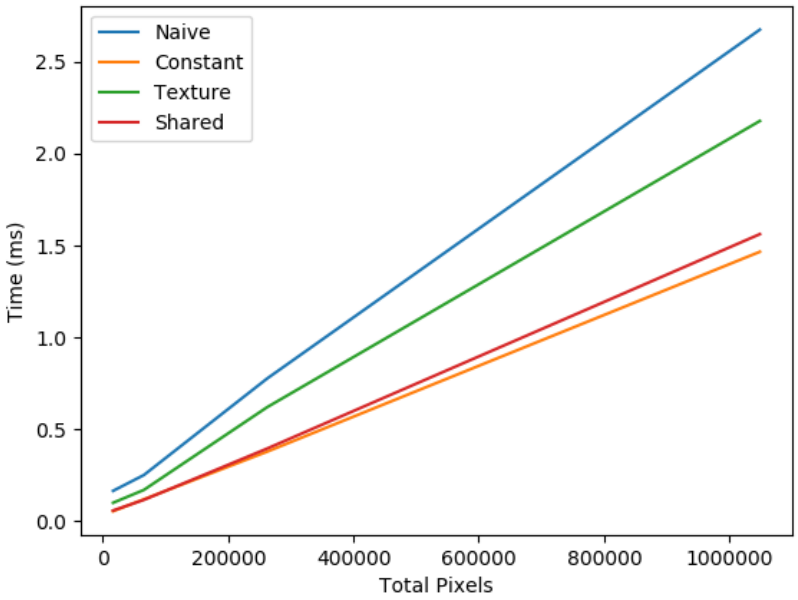

Figure 2: Time for 7x7 filter vs Image Size

4.3 GFLOPS vs Image Size

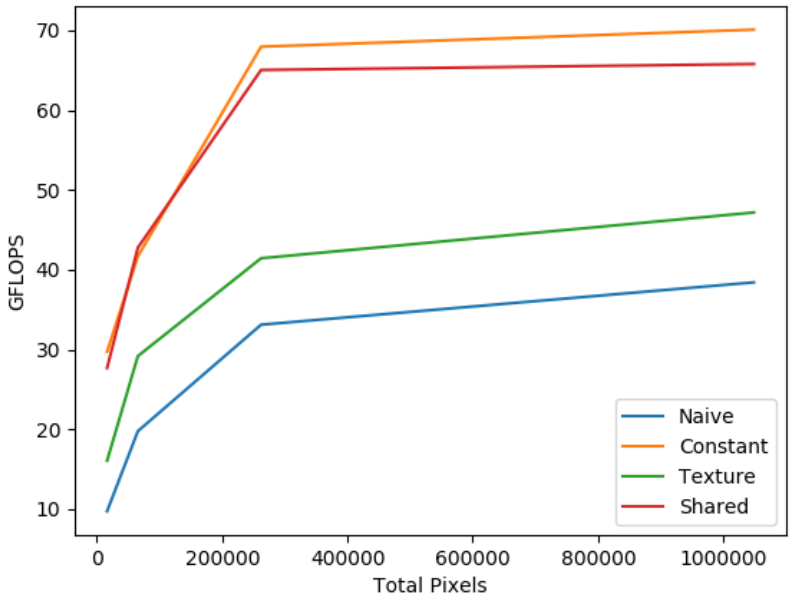

Figure 3: GFLOPS for 7x7 filter vs Image Size

4.4 Overhead vs Image Size

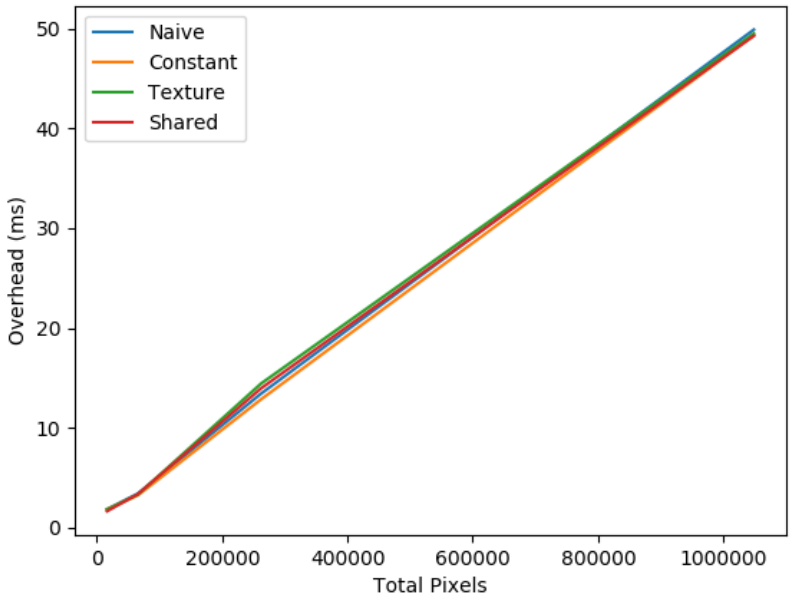

Figure 4: Overhead for 7x7 filter vs Image Size

4.5 Time vs Filter Size

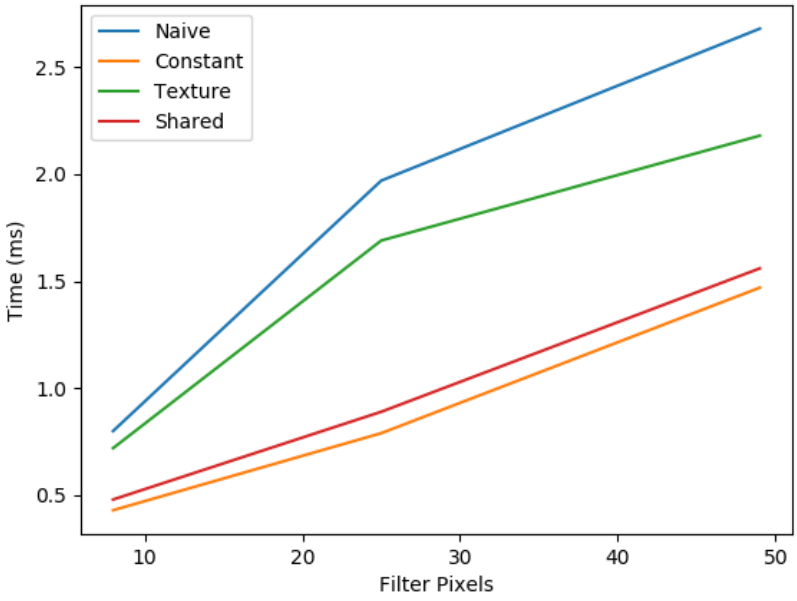

Figure 5: Time for 1024x1024 Image vs Filter Size

4.6 GFLOPS vs Filter Size

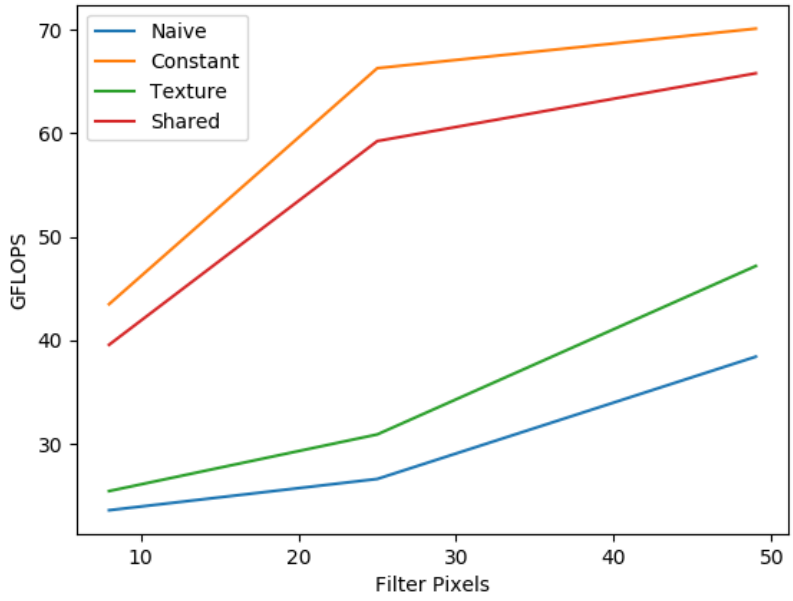

Figure 6: GFLOPS for 1024x1024 Image vs Filter Size

5 Discussion

The serial implementation is drastically slower than all of the parallel implementations, the time for the 1024 x 1024 image and a 7x7 mask is of the order of 400ms. While the GFLOPS is significantly lower at 0.25 GFLOPS, this figure seems to not scale at all with the problem size. This is probably due to reaching a performance peak at a small problem size. The parallel implementations consistently outperform the serial on all metrics. The overhead time and its scaling of the serial implementation is similar to that of the parallel implementations.

For the Naive Approach, the total number of global memory accesses is equal to $Width \times Height \times FilterSize \times FilterSize$ where $Width \times Height$ is the number of pixels in the image and $FilterSize$ is the total number of coefficients in the filter. For constant and shared memory it is equal to $Width \times Height \times FilterSize$. For texture memory the number of global memory accesses is only the accesses for the filter i.e $Width \times Height \times FilterSize$.

From section 4.2 we can see that for all the parallel convolution algorithms, the time scales linearly with the image size. The best performance is achieved

by the use of Constant Memory is slightly faster than Shared memory, this result is as expected as constant memory access is as fast as register memory access when the memory accesses are coalesced correctly. The shared memory is only slightly slower and this is expected as Shared Memory is on-chip memory. Texture Memory is moderately slower, this is potentially due to a poor texture memory access pattern which results in a high texture cache miss rate.

From section 4.3 we can see that GFLOPS for all the parallel algorithms is roughly logarithmic. The formula for GFLOPS is [1]

$$\frac{Width \times Height \times FilterSize \times 2}{time \times 10^9} \tag{1}$$

The sudden "capping off" of the GFLOPS is potentially due to high bandwidth due to poor memory access patterns. This explains why the Naive,Constant and Shared graphs follow roughly the same shape as all the algorithms are still reading the image from global memory. The reason the performance difference between Texture and Naive is constant is explained by the fact that texture memory access is faster than global memory access. This supports the point that the texture memory implementation has a poor memory access pattern and is just accessing memory in the same pattern as the Naive approach.

From section 4.4 we can see that the overhead scales linearly with the number of pixels in the image. The difference in overhead between the algorithms is negligible. Thus it can be said that one should choose between these methods purely on the performance of the kernel and not the overhead.

From section 4.5 we can see that shared and constant memory performance scales linearly with the filter size as expected. The cause of the roughly logarithmic scaling for Naive and Texture Memory is unexpected as Naive is expected to scale quadratically and Texture is expected to scale linearly. These anomalies require further investigation. From section 4.6 we can see a similar but inverted pattern due to GFLOPS being inversely proportional to time.

References

[1] How to implement performance metrics in cuda c/c++. http://cuda-programming.blogspot.com/2013/01/how-to-implement-performance-metrics-in.html. Accessed: 2019-04-13.

[2] Nvidia. Nvidia cuda c programming guide. https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Accessed: 2019-04-13.