WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

GPU Image Convolution

Michael Beukman

July 28, 2021

Contents

1 Introduction

Image convolution is a process where a small filter or kernel is placed over each pixel in an image and an element wise product is done between overlapping filter values and image values. The resultant grid is then summed and that value is assigned to the output image at that location. This is repeated for every pixel in the input image.

Convolution is used to modify images, or to extract meaningful information. This can be to, among others, obtain the edges in an image, sharpen it, blur it (Gonzalez and Woods, 2001).

This can clearly be very computationally intensive, especially for larger filters. For an image of size $M \times M$ and a filter of size $S \times S$, we need to perform S^2 operations for every one of the M^2 pixels in the image, resulting in an overall complexity of $O((MS)^2)$

Because of this, convolution is usually performed in the frequency domain by taking the discrete fourier transform of the image and filter, multiplying the results and performing the inverse fourier transform on the image. This is usually faster, especially for larger filters. Another optimisation method involves a filter that is separable (i.e. it can be written as the outer product of two vectors) and convolving the image with each filter one after the other (Gonzalez and Woods, 2001).

Since the objective of this report is to study the performance implications of using different methods of GPU acceleration, we will use the simple, spatial method of convolution.

2 CUDA

NVIDIA's CUDA programming api (NVIDIA *et al.*, 2020) is a way to use graphics cards, which were initially mainly intended for the graphics in games, for general purpose parallel tasks (Sanders and Kandrot, 2010).

Because GPUs have a large number of cores and processing units, some tasks that can be effectively decomposed can achieve massive speedups over a serial (or even parallel) CPU based implementation. Examples of these are the molecular dynamics simulator GROMACS (Bekker *et al.*, 1993), which recently added native GPU support (Abraham *et al.*, 2015) and ChaNGa (Jetley *et al.*, 2008, 2010), which is a massively parallel N-body code that also makes use of GPU acceleration.

CUDA programs are based on decomposing a task into *thread blocks* which contain a number of threads that can execute the same instructions on different data. There are then multiple of these thread blocks that each access a different part of the data.

2.1 CUDA Memory

One very imporant consideration to make when writing CUDA programs is the way the algorithm accesses memory. Using different accessing techniques and accessing different parts of the GPU's memory can have a great impact on overall performance.

The main types of memory in a GPU are the following:

- Global Memory: This memory is accessible from every thread and accessing this is often slow due to limited bandwidth and (relatively) high latency.
- Constant Memory: This is a small (64KB) constant memory cache, to which all the threads have read-only access to. Using this instead of global memory when applicable can lead to large speed increases. This is due to its broadcasting feature, whereby if multiple nearby threads read the same address, the data can be broadcast to multiple threads at once, instead of having each single read send data independently (Sanders and Kandrot, 2010, chapter 6).
- Shared Memory: Each thread block has its own cache memory which is shared between all of the threads in the block. This is again much faster than global memory (Sanders and Kandrot, 2010, chapter 5). It is limited in size though.
- Texture memory: Originally intended for graphics textures, this is another read only memory that resides in device memory and is larger than the constant memory cache. The texture memory is usually cached in the texture cache. Textures in particular take advantage of the concept of spatial locality in two dimensions, which leads to fast memory accesses when nearby threads access nearby elements (which don't have to be adjacent and can be in the next 'row', and can be nearby in the 2D or 3D sense of the word) (Sanders and Kandrot, 2010, chapter 7).

In this report, we explore different implementations of image convolution that use the different memory types described above and compare their performance and results to each other and a CPU reference implementation.

3 Methods Used

We used a range of different methods which mainly differed in what type of memory they used.

We decompose the problem into 2d, square thread blocks with dimensions of either 4×4, 8×8, 16×16 or 32×32. The number of blocks in each dimension was then dependent on the size of the image.

- Our first, naive approach was simple. Spawn as many threads as there are pixels in the image, and each thread computes the value in the output image at that location by multiplying the corresponding image and filter elements. This just used normal global memory. For each output pixel, we performed $2S^2$ (S^2 for each of the filter and image) global memory accesses and $2S^2$ floating point operations. This gives us a very low ratio of compute (1) to global memory accesses.
- Our first modification was small: we just store the filter in constant memory. We could not store the image there, as constant memory was too small. Here we perform half the amount of global memory accesses, as we only access the image from global memory.
- Our second modification used shared memory to store the chunk of the image that the current block was working on (and still stored the filter in constant memory). This also used constant memory as described above. Specifically, the size of the shared memory that every block had was $(\text{block\_size} + S - 1)^2$. This means that the shared memory contained the pixels that the threads in the current block were working with, as well as the requisite extra padded pixels that are required for the convolution. When copying the data from global memory to shared memory, each thread copied multiple values (its assigned pixel and some of the padding), by sliding the thread block over the padded size. An alternative could have been to assign as many threads as the total number of padded pixels. This would improve the speed of the memory copy, but it would result in many threads being idle during the computation. For each pixel, we perform $\frac{(S+\text{block\_size}-1)^2}{\text{block\_size}^2}$ global memory accesses accesses (because each thread block copies $(S + \text{block\_size} - 1)^2$ elements, which are shared among block_size^2 threads) and still $2S^2$ operations. Thus, for a simple example of using a block size of 16 and $S = 17$, our compute to global memory access ratio is then $\frac{2\times 17^2}{32^2/16^2} = 144.5$, which is much larger than the 1 or 2 we previously had.
- Our third one used texture memory to store the image instead of global memory.
- Our fourth used texture memory to store both the filter and image.

Since there was only limited space in the plots, we somewhat shortened the names when showing these graphs. The following is a mapping between what was described above and what will be shown in the plots.

- *Naive*: The first, simple method described above.
- *Constant*: Used constant memory for the filter.
- *Constant and Shared*: Used constant memory for the filter and shared memory for the image.

- *Texture with Padding*: Used constant memory for the filter and texture memory for the image.
- *Texture Image and Filter*: Used texture memory for the image and filter.
- *Naive without padding*: The naive method, but it used explicit boundary checking instead of padding.

We used slight variations on the above methods, which performed largely similar. The results from these extra methods will be shown in Section 5.9.

The serial method used was quite simple and we didn't perform any explicit optimisations. It consisted of 4 *for* loops over the width and height of the image, as well as the width and height of the filter.

3.1 Design Choices

We made some specific choices when implementing the different methods, which affects the exact results obtained. Notably, when using constant memory, we only copied the filter to the constant memory, as the image was in general too large to fit in this limited space.
When using shared memory, each thread calculates the value of a single pixel in the output image. We opted to save the region of the input image, as well as the requisite padding in the shared memory because even the padding will be used multiple times. We investigated using different methods of copying.
We also did not use as many threads as input elements that need to be copied, but rather used as many threads as the number of output pixels. This prevented having many threads be idle during computation.

For texture memory, we used a texture reference and cuda arrays because we did not need to pass a texture object to a function, and there were no features that we desired that texture references did not provide.

Note, we padded the image with zeroes beforehand to a size of $(M+S-1)\times(M+S-1)$. This was done instead of explicit boundary checking in the kernel because it was the most extensible. When other boundaries are desired (i.e. reflection/replication padding), it will be easy to change. This also shouldn't impact results too much, as we show in one experiment where we did not use padding. Crucially, since we use the same method to compare the different techniques, we can sensibly compare the different results. Also, when writing the images to disk, we clamped the results to be between 0 and 255, to be able to get valid results for the sharpening and edge detection filter. This was done on the CPU. One could also use image scaling, but that would result in the whole image becoming lighter, and potentially obscuring the effect of some of the aspects of the filtered image. We did not measure the time that this process took, as it was mainly for display purposes and it was also constant across all methods.

4 Experiments

For our experiments, we compared each of the methods described above across a range of parameters and for a few different metrics. The different parameters were:

- Which filter to use (i.e. box filter, sharpening, edge detection).
- The size of the filter in the averaging case.
- Which of the given example images to perform convolution on.
- Using different sizes of random images to be able to judge how the performance is affected by the size of the image.
- The number of threads in a block. We tried 4×4, 8×8, 16×16 and 32×32.

The times measured were:

- The time taken for the actual convolution.
- The time taken allocate memory on the GPU and to copy data from the CPU to the GPU and vice versa.

For the serial version we only measured the computation time and set the copy time to 0. The time taken to generate the filter, allocate memory on the host and pad the image was not measured, as this was constant across all the methods. We used the CUDA event system to time the actual execution time of the kernels, and the *chrono* library to measure the other aspects (e.g. data copy/ memory allocation).

Each experiment was performed 20 times and the results were averaged. Every run was validated against the reference serial implementation to ensure that the GPU results were the same as the CPU one.

Regarding the validation, $\varepsilon = 1e-8$ was used to check equality between the floating point numbers in the GPU and CPU implementations (i.e. $a = b \iff |a-b| < \varepsilon$). By default, when using this value of ε, the values weren't considered equal. This was caused by the default *nvcc* argument *(--fmad=true)*, which contracts multiply and add operations into a single fused operation. This results in slightly faster speeds and slightly less precise results. When using *--fmad=false*, the answers were equal up to $\varepsilon = 1e-8$ and the performance was slightly slower. We used -O3 for the host code to be able to fairly compare the results, as the CUDA code is optimised by -O3 by default. If one does not necessarily need very precise results, then using *--fmad=true* can result in some slight performance increases, and the results will still be functionally indistinguishable from the CPU version.

The final flags used for the majority of this report, where thus *nvcc -O3 --fmad=false*. We investigate the effect of using the *fmad* flag in Section 5.10.

We will mainly use the time taken in milliseconds as our metric of comparison. We will also use the number of floating point operations that our methods perform in a second (FLOPS). This will give us some indication of how the performance changes when the problem size changes. We use the unit GFLOPS, which is defined as 1 GFLOPS $= 10^9$ FLOPS

The number of floating point operations will be calculated as $2 * (MS)^2$, as for each pixel, we perform S^2 multiplies and S^2 additions.

All computations were performed on an *Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz* and the GPU used was an *NVIDIA GeForce GTX 1060 6GB*. The serial version that is referred to is simply a single core CPU implementation without any extra optimising. Theoretically one could parallelise this using OpenMP, but the maximum possible speedup will be between 4 and 8, depending on the number of cores, which is an insignificant speedup compared to the GPU versions.

Most of the experiments were performed using an averaging filter to be able to judge the different methods as a function of filter size. Later experiments used other filters. If not specified, the image used was the 512×512 image in Figure 1.

5 Results

Since there are many different variables at play that might affect performance, we study each independently. In Section 5.1 we consider the results of the convolution and validate that all of our implementations do the right thing. Then, in Section 5.2 we compare the GPU versions vs the serial one. Thereafter we more closely look at the differences between the different methods in terms of time in Section 5.3, in terms of FLOPS in Section 5.4 and in terms of distribution of time in Section 5.5. We look at the effect of block size in Section 5.6. We then investigate the effect of using other image sizes in 5.7 and using different filters in Section 5.8. At the end we investigate small changes in the method in Section 5.9 and the effect of compiler flags in Section 5.10.

Note: The times (and GFLOPS) reported are either just considering the time spent in the kernel, which are referred to as *Time Taken* and *GFLOPS_raw*. We also consider the total time, which includes the compute as well as data copy times, which are then referred to as *Total Time* and *GFLOPS_corrected*.

5.1 Images

Figure 1: Original Image

Figure 2: Image filtered using an averaging filter of size 3×3. From Left to Right: All texture memory, constant memory, constant and shared, naive, texture memory and constant

Figure 3: Image filtered using an averaging filter of size 23×23 From Left to Right: All texture memory, constant memory, constant and shared, naive, texture memory and constant

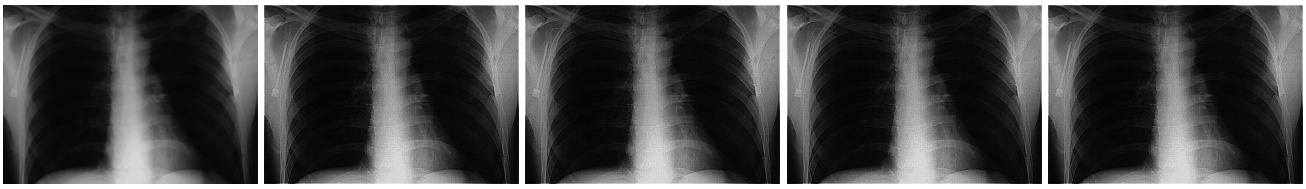


Figure 4: Non-square image filtered using a sharpening filter of size 3×3. From Left to Right: Original Image, constant memory, constant and shared, naive, texture memory and constant

We can see here that the results from the different methods are identical (which we also verified during the running of our experiments). From now on, we will only show one image instead of one for each different method to save space.

The averaging filter clearly blurs the image and the size of the filter impacts how much it blurs.

In Figure 5 we can see the effect of the different filters on images. All the filters were of size 3×3. The blurring filter blurs the images, the sharpening filter makes more detail visible and the edge detection filter clearly extracts the edges.

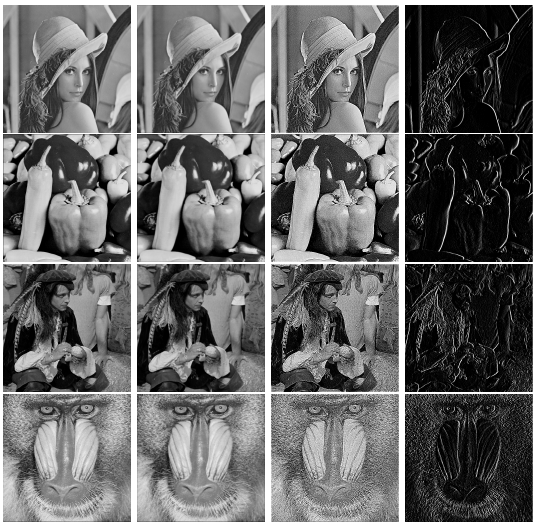


Figure 5: From Left to right: Original Image, Blurred image, Sharpened Image, edge detection

5.2 Comparing to Serial

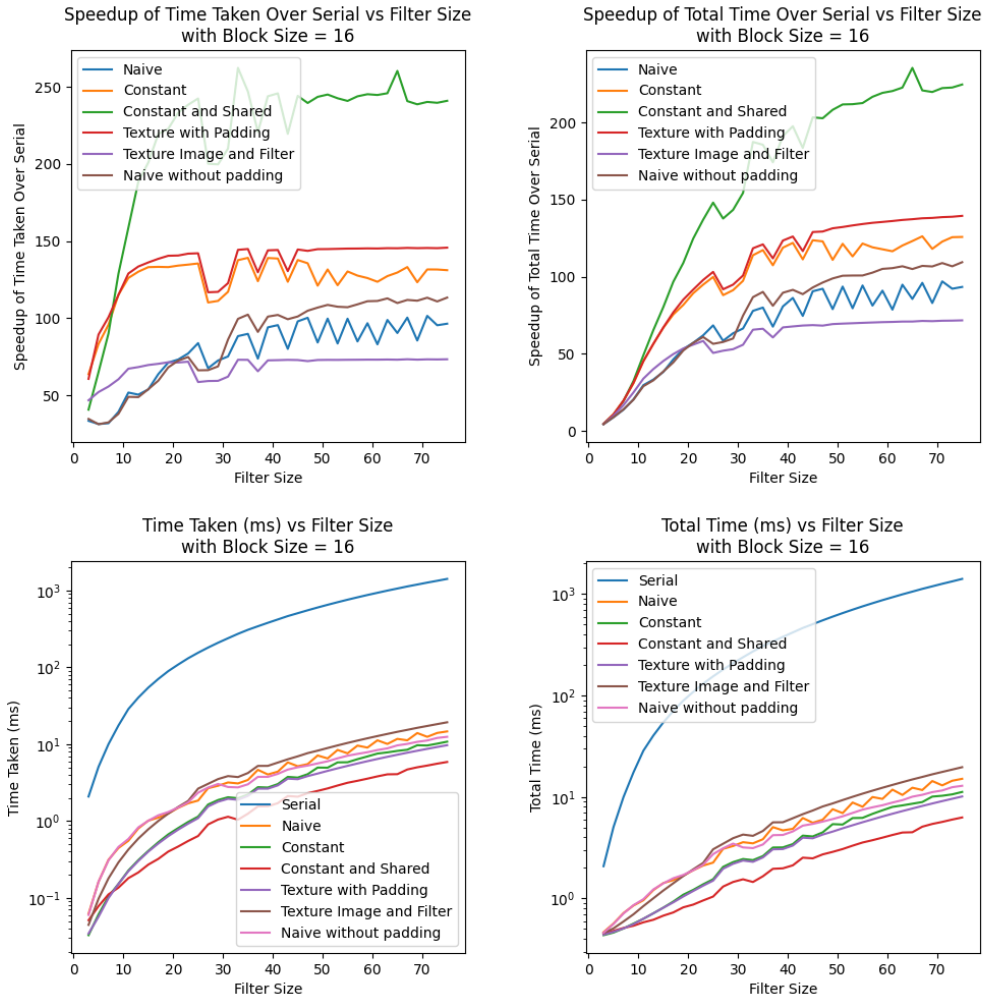


Figure 6: Time Taken for the actual convolution and total times including copying.

From Figure 6 it is clear that the serial version is much slower than the GPU version on all the tested filter sizes. The difference in GPU times are there, but they are much smaller than the difference between the CPU and GPU. Even when taking into account the time to copy the data to and from the GPU, the GPU methods are still hundreds of times faster than the serial CPU version. The reason that the GPU version is so much faster is that the problem can be parallelised to a very high degree, since value of a certain output pixel is independent of any of the other output pixels. This means that the entire process can be performed in parallel on the GPU. When performing the CPU process, each pixel was calculated sequentially.

For the rest of this report, we omit the serial implementation's performance, since it is difficult to see the differences between the different GPU implementations on the graphs without zooming in.

5.3 Comparing the times of different methods

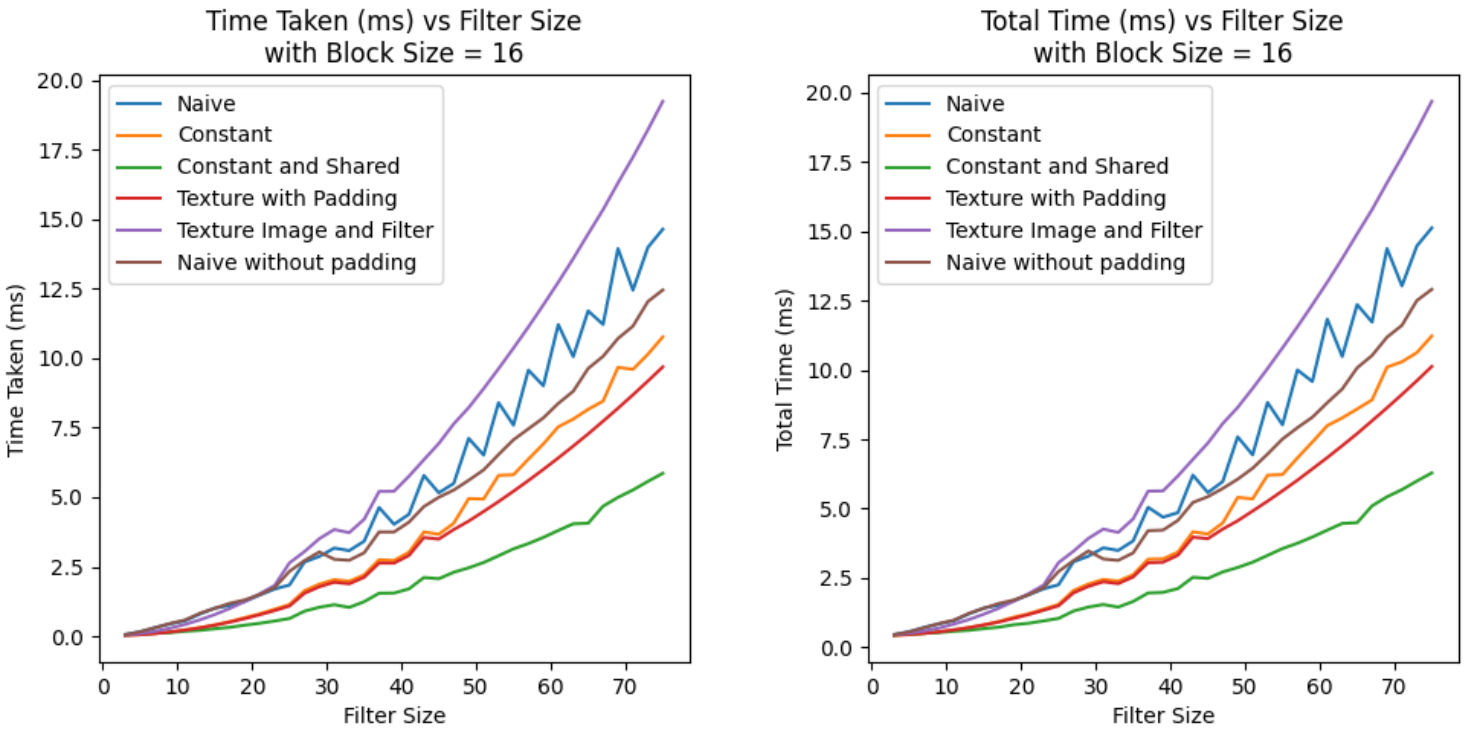


Figure 7: The total and computation time taken for each method

We can see that the worst methods are the naive one and the method that uses texture memory for the filter. This could be because of the high amount of global memory traffic (as texture memory is still stored in device memory). Using constant memory improves the performance and using shared memory with constant results in the best performance. In general, texture memory looks to perform similarly (but slightly better) than global memory, even though we would expect it to perform much better due to the spatial locality of this problem.

Later, in Section 5.6, we see that the performance of texture memory is quite good when using a small block size.

The best GPU method performs roughly 60% faster than the second best and about 3 times better than the worst GPU method. Even the slowest GPU method takes only 20ms for a filter of size 71×71.

5.4 Comparing Throughput and GFLOPS

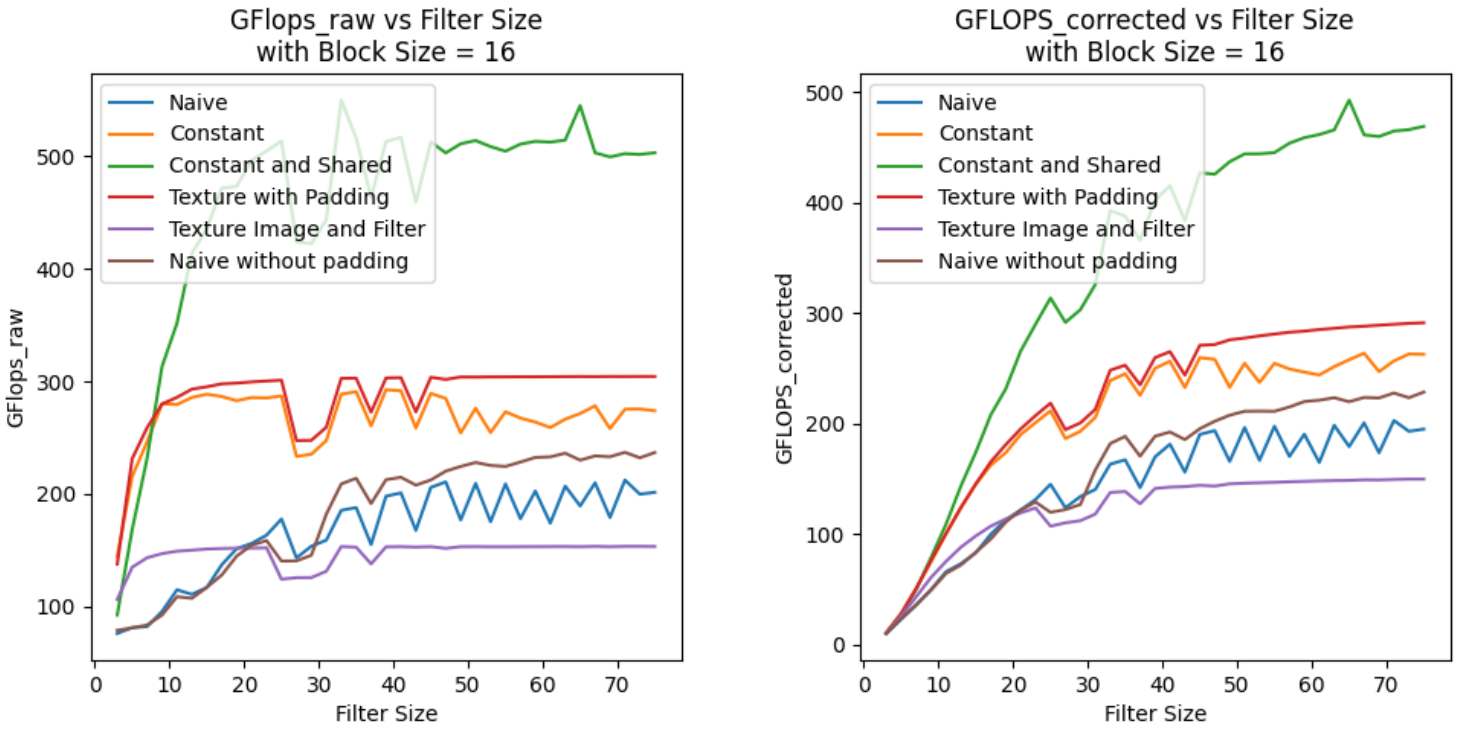


Figure 8: The number of floating point operations performed. The left plot only takes into account time spent in the kernel and the right one takes into account the total time, which includes data copying.

After a certain filter size, the amount of computation performed remains relatively constant when not measuring data transfer times. When measuring these however, the performance increases steadily as the filter size increases. This could be because the transfer time is dominated by a relatively constant latency, and when we send more data and perform more computation, this constant latency becomes less and less relevant to the total time taken.

Again, texture memory and constant memory gives similar performance as global and constant memory. Shared and constant memory performs the best by far.

5.5 Distribution of Times

Here we discuss how much of the total time the computation and data transfer components take.

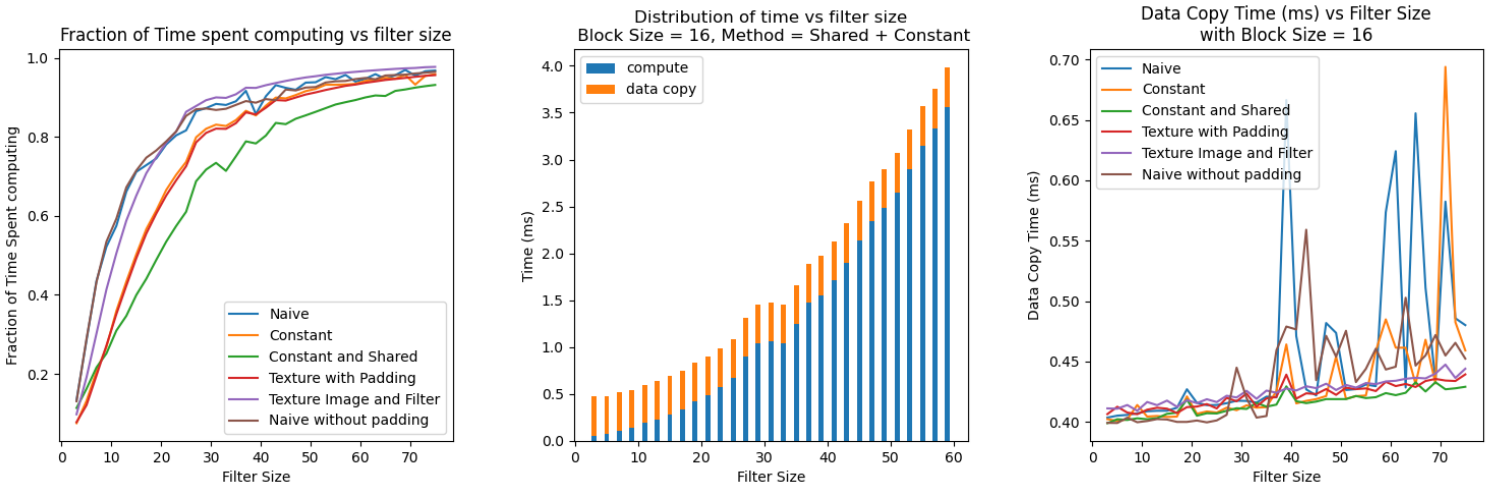


Figure 9: How much time is spent in the kernel (as opposed to copying data)

As the filter size increases, we spend much more time in the kernel than copying data. This makes sense, because if the filter width doubles, we have to copy 4 times the filter data to the gpu (which is a small proportion of the total copied data, because the image is much larger). But, each thread now also has 4 times the computation to perform.

We see that the constant and shared memory implementation spends the least time in the kernel. Given that the data copy times between the different methods are similar, this also confirms that this method is the best one.

The time required time to copy the data is quite similar across the different methods, as for each method we have to copy the image data to the GPU, which takes up the bulk of the copying and allocating time. There are some anomalies, but they can be considered as just random deviations. There is a slight upward trend in the data copy times as the filter size increases, confirming that the image data copying takes up the bulk of the total copy time.

5.6 Block Sizes

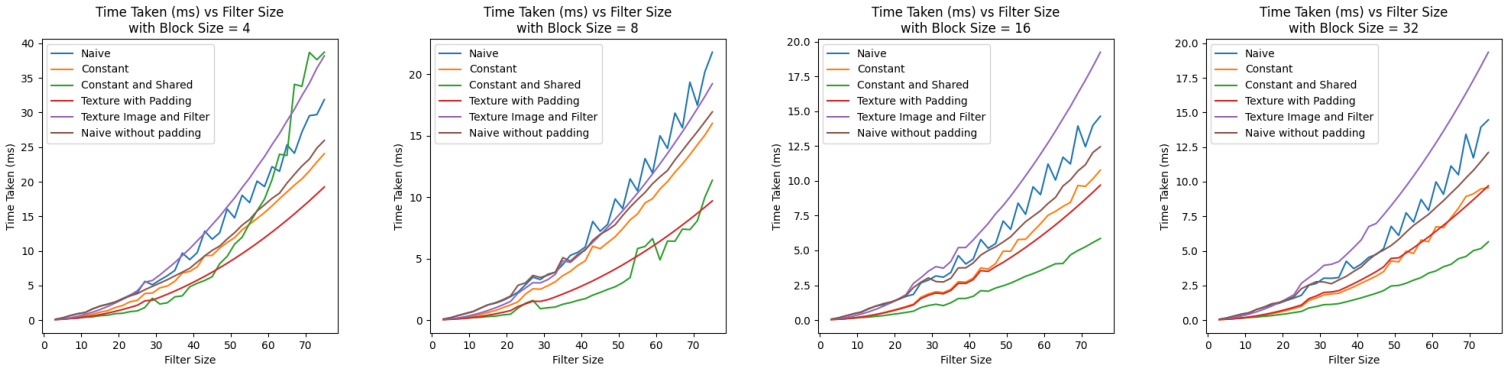


Figure 10: The time spent in the kernel with different block sizes

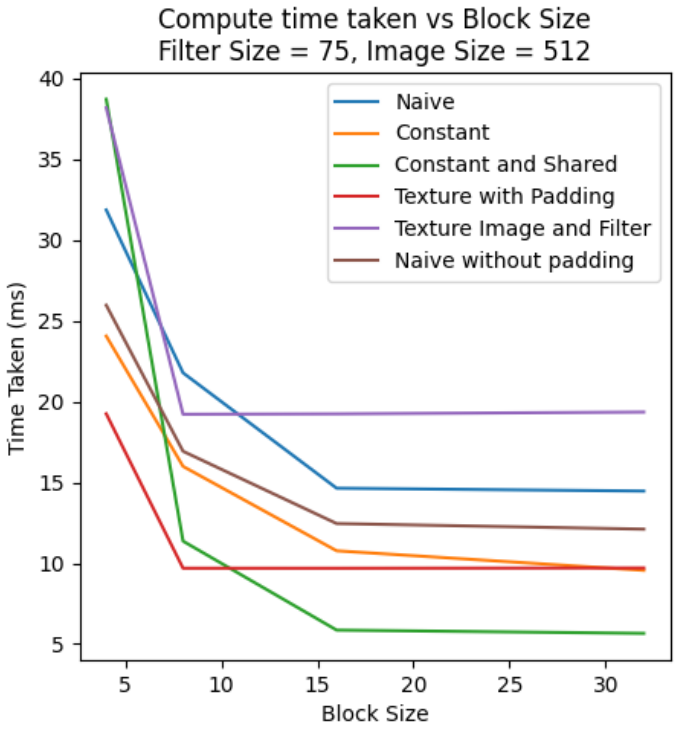


Figure 11: Time taken vs block sizes

Here we can see that the relative performance of the different methods are roughly the same when using different block sizes (except constant and shared vs texture memory). There is a substantial difference between a small block size of 4 and a larger one of 16 or 32. The larger block size performs almost twice as fast as the

smaller one.

A small block size hampers the performance, and we achieve similar performance with 16 and 32 block sizes. Texture memory does not gain much increase in performance with block sizes larger than 8.

One interesting observation is that for smaller block sizes, using texture memory outperforms the shared memory implementation for a large filter size. This could be because as the filter size increases, each thread copies more data into shared memory, whereas the texture memory implementation's results are mostly independent of the block size. When using a small thread block and a large filter, it is possible that most of the time in the kernel is spent copying data, rather than actually performing computation. This is alleviated when using more threads, because the number of threads that do the copying increase by a large amount, whereas the data to be copied increases by a smaller amount.

To illustrate this point, consider a simple example of using a filter of size 61×61 and a thread block of size 4×4. The total data to copy is then $(61-1+4)^2$, and the number of pixels to copy per thread is equal to 256. When using the same filter size with a block size of 16×16, the number of pixels to copy is $(61-1+16)^2$, which leaves about 23 per thread. So more computation is performed per copy.

5.7 Images of different sizes

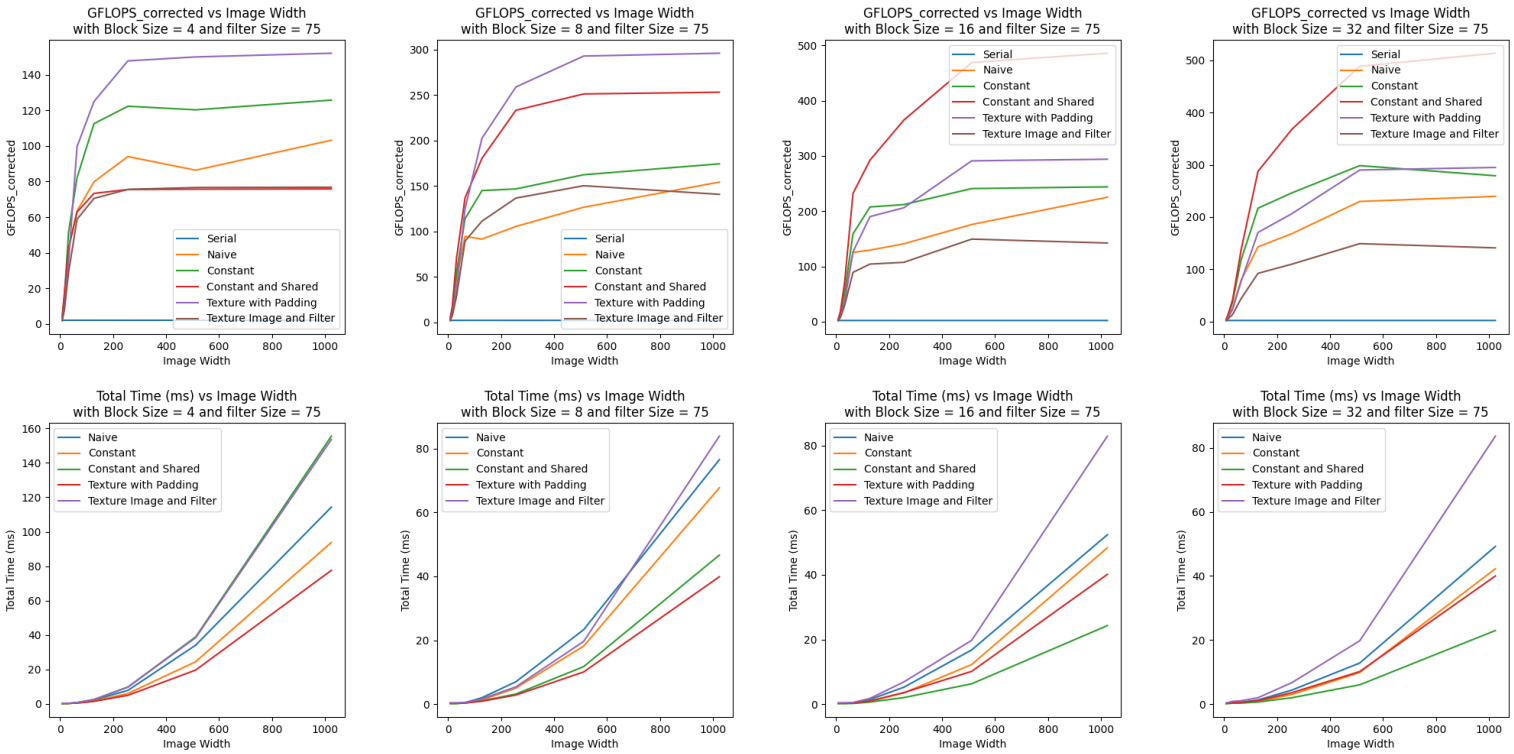
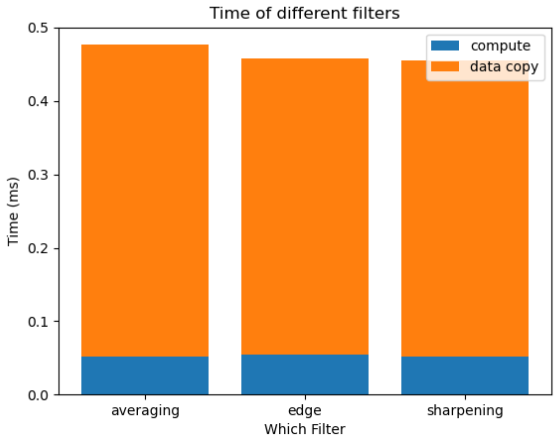


Figure 12: Comparing Different Image Sizes

In this experiment, we used random images of varying sizes to investigate the impact of the size on the performance of the different methods. Again, using constant and shared memory vastly outperforms the other methods. The time taken has a roughly quadratic relationship with the size of the image. We only tested square images in this experiment, but we hypothesise that the same trend would occur for non-square images.

One very interesting point to note is the impact of the block size on each method. When using a small block size of 4, the texture memory implementation performs best, followed by the constant memory method. Using a block size of 8 causes the texture memory to double in performance, the constant and shared to triple and the constant only implementation to stay roughly the same. When using 16 and 32 as the block size, the constant and shared memory method achieves the best performance while the other methods stay roughly the same. This indicates that to actually reach the maximal performance for shared memory, there needs to be enough threads per block to offset the copying time. Since the results here were with a large filter, when using a small block size, most of the time is spent copying data to the shared cache, whereas with a larger block size, this ratio is skewed towards more compute per data element transferred.

5.8 Different filters



We don't really see any difference in performance when using different types of filters of the same size. This is because the amount of computation performed is the same. When using a small (3×3) filter, most of the time is spent copying data.

5.9 Slight Variations

Name	Filter Size	Block Size (x)	Image Width	Time Taken (ms)	Copy Time (ms)
Texture and Shared	75	32	1024	21.77	1.18
Constant and Shared	75	32	1024	21.83	1.15
Constant and Shared Row Wise	75	32	1024	22.07	1.16
Constant and Shared (Bad) V2	75	32	1024	22.83	1.16
Constant and Shared (Bad) V1	75	32	1024	24.21	1.17
Constant	75	32	1024	37.85	1.18
Texture with Padding	75	32	1024	38.81	1.17
Texture without Padding	75	32	1024	38.82	1.12
Naive without padding	75	32	1024	48.58	1.18
Naive	75	32	1024	49.89	1.21
Texture Image and Filter	75	32	1024	79.59	1.24
Serial	75	-	1024	5694.88	0.00

Table 1: Table of times. This used a square image.

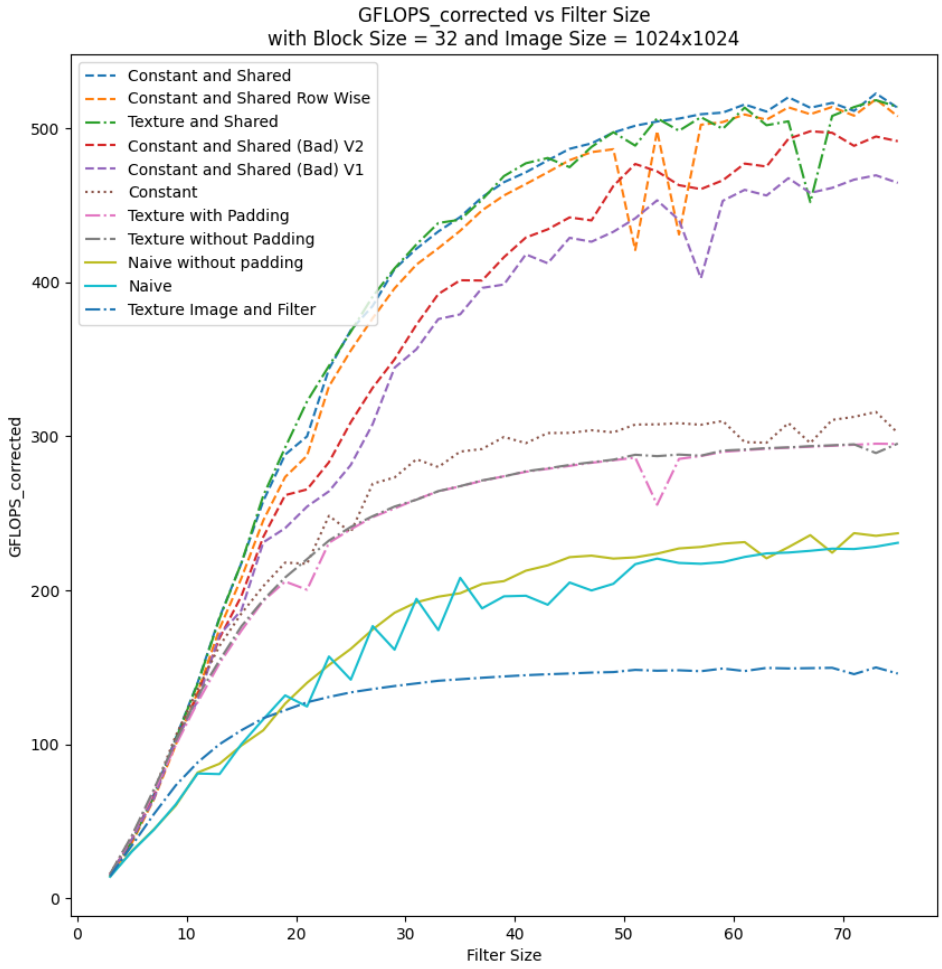


Figure 13: Comparing all of our different methods on one graph

Here we compare all of the methods used. Besides the methods that were discussed and explained early on, we have a few other methods. Most methods except the Naive method and the one where we used texture memory for the filter stored the filter in constant memory.

- Constant and Shared Row Wise: This is the same method as the normal constant and shared, but we copy data in a row-wise manner instead of copying multiple 2D blocks. The goal was to improve memory coalescing when copying the data from global memory by performing row-wise (and thus coalesced) copying.
- Texture and shared: This used texture memory, but copied the image from it to shared memory.
- Constant and Shared (Bad) V1 and V2: These still used constant and shared memory, but the copying methods were not ideal. They were initial, simpler attempts at copying and we include them, as the results are interesting. In V1, each thread copied one element, and thread (0, 0) copied the rest of the padding elements. In V2, each thread copied one element, except the edge threads (i.e. in the first or last row or column). Thread (0, 0) copied the rest.
- Texture without padding used the same structure as the normal texture memory method, but used the texture address mode 'cudaAddressModeBorder'. It thus did not make use of explicitly padding the image.

In Figure 13 we compare all of our methods, and the conclusions that we can draw from this is: Using shared memory, and copying from either global or texture memory gives the best results. Using constant memory with a sub-par copying method is also good, but not as good as when using a balanced copying method.

Then we see constant memory with global memory gives similar results to constant memory with texture memory (with this block size). It didn't really matter if we used padding or not for texture memory. One would expect texture memory to improve the performance on a spatially local task like convolution, but we didn't see this for larger block sizes. The reason could be that for a larger block size, the global memory accesses are cached quite often, and the texture memory overhead might become more noticeable.

The naive method comes in second last place, and not using padding looks to perform slightly better than using padding on average. This could potentially be the result of the reduced data copy times.

Finally, using texture memory for both the image and filter performs badly. This could be because texture memory is not designed to access the same elements over and over again.

In Table 1, we can see that the first two methods perform similar since they are essentially the same method, except one uses a slightly different way of copying global memory to the shared memory. There is no real difference (except a slight copy time difference) between texture memory using padding and not using padding. Using constant memory performs better than the naive way, but since most of the global memory access are still there, it doesn't do great. Storing the filter in texture memory performs terribly.

5.10 Compiler Flags

As described above, the compiler flags used can impact the precision and performance of a kernel. In this test, we used the flag *--use_fast_math*, which enables some features that enable the GPU to perform quicker math, at potentially a cost of accuracy. This flag also enables the *--fmad=true* flag. We only validated the results with an $\varepsilon = 1e-5$, but that is still precise enough to not see any difference in the resulting image.

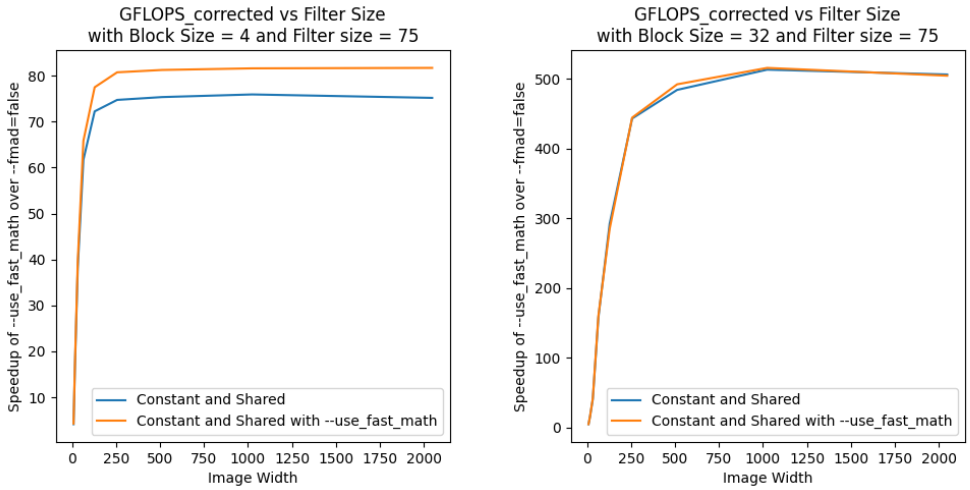


Figure 14: Comparing Using *--use_fast_math* vs *--fmad=false*

In these graphs we can see that when using a small, inefficient block size, *--use_fast_math* provides a small increase in performance. When using a viable block size of 32 though, this performance increase reduces to nearly nothing, which is somewhat surprising. So it does not necessarily make sense to use this flag for this problem.

6 Limitations

There are some limitations with our approaches as detailed above, most importantly regarding the maximum size of images/filters we can support.

Notably, the amount of constant memory is capped at 64KB [1], which means that our filters cannot be larger than 128×128. Also, in our shared memory implementation, we use shared memory that contains $(S + \text{block\_size} - 1)^2$ floating point numbers per thread block. For the GPU that our experiments ran on, the maximum amount of shared memory per thread block is limited to 48KB, which means that for a block size of 16×16, we can only support filters up to 93×93 in size. Newer GPUs increase this amount slightly, but there

[1] All of this information was retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities on 2021/05/02 11:36

is still a limit. Using a tiling method can partially address the latter problem, but preliminary testing showed that this was much slower than the best method. One could always just fall back to the texture memory method when the necessary pixels won't fit in the shared memory. This will run about twice as slow, but it is still much faster than a CPU version and should still run in very reasonable time. If the filter is too large for constant memory, one can always store that in either shared memory or global memory, which will still be fast enough for most use cases.

7 Conclusions

In this report we discussed many different GPU methods for image convolution and compared them on a wide variety of input parameters. The main methods focused on used constant memory with global memory, constant and shared memory as well as texture memory with constant memory. We found that the convolution of larger filters and larger images took more time to complete, and increased roughly quadratically.

A bigger block size usually resulted in better performance, with 16 being a good choice. Using shared memory allowed each thread block to reuse data, thus significantly reducing the memory traffic.
Constant memory could broadcast accesses to the same address across a warp in a thread block, which again improved memory access times. This type of memory was also aggressively cached on high speed local cache, which further improved results. Texture memory did not deliver much better results than global memory for larger block sizes, but performed relatively well on smaller block sizes.

The main conclusions that we can draw from this report is that using GPU accelerated convolution methods can improve performance significantly over a simple CPU based implementation. The specific way that one implements this GPU acceleration, specifically which type of memory is used is also quite important. We found that using a combination of constant and shared memory performed the best, although this is only feasible up to a certain filter size. Using texture and constant memory is a good fallback that is easy to implement.

8 Acknowledgements

Computations were performed using High Performance Computing infrastructure provided by the Mathematical Sciences Support unit at the University of the Witwatersrand.

References

[Abraham *et al.* 2015] Mark James Abraham, Teemu Murtola, Roland Schulz, SzilÃąrd PÃąll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1-2:19–25, 2015.

[Bekker *et al.* 1993] Henk Bekker, Herman Berendsen, E.J. Dijkstra, S. Achterop, Rudi Drunen, David van der Spoel, A. Sijbers, H. Keegstra, B. Reitsma, and M.K.R. Renardus. Gromacs: A parallel computer for molecular dynamics simulations. *Physics Computing*, 92:252–256, 01 1993.

[Gonzalez and Woods 2001] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*, pages 154–160. Addison-Wesley Longman Publishing Co., Inc., USA, 4th edition, 2001.

[Jetley *et al.* 2008] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V Kale, and Thomas Quinn. Massively parallel cosmological simulations with changa. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE, 2008.

[Jetley *et al.* 2010] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V Kalé, and Thomas R Quinn. Scaling hierarchical n-body simulations on gpu clusters. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[NVIDIA *et al.* 2020] NVIDIA, PÃl'ter Vingelmann, and Frank H.P. Fitzek. *CUDA, release: 10.2.89*, 2020.

[Sanders and Kandrot 2010] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

Bibliography

[Ichimura 2019a] Naoyuki Ichimura. *Accelerating Convolution Operations by GPU (CUDA), Part 1: Fundamentals with Example Code Using Only Global Memory*. https://qiita.com/naoyuki_ichimura/items/8c80e67a10d99c2fb53c, May 2019.

[Ichimura 2019b] Naoyuki Ichimura. *Accelerating Convolution Operations by GPU (CUDA), Part 2: Utilizing Constant and Shared Memory for Data Reuse*. https://qiita.com/naoyuki_ichimura/items/519a4b75f57e08619374 with code found at https://github.com/NaoyukiIchimura/cuda_image_filtering_shared and https://github.com/NaoyukiIchimura/cuda_image_filtering_constant, Feb 2019.

[NVIDIA] NVIDIA. *CUDA Samples*. https://docs.nvidia.com/cuda/archive/10.1/pdf/CUDA_Samples.pdf.