COMS4040A & COMS7045A Assignment 2 – Report

Tasneem Abed, 1408535, BDA Hons

14th April 2019

1 Introduction

In this assignment, we aim to study the effects on computation performance by using different types of memory spaces that exist within the CUDA framework. The algorithm that will be the basis of this experiment is image convolution. This is a process by which images are transformed by a filter (or mask) to produce a new image. These filters are matrices with specific values. For example, to sharpen an image, we apply a matrix of -1's where the middle value is equal to the dimension of the matrix i.e. a $3x3$ filter has a middle value of 9. We make use of pgm images. These can be imported as a 1D array of pixel values that range between 1 and 0 and represent the depth of grey of that pixel.
Convolution is an algorithm that takes the mask and places it over each pixel of the image and multiplies each pixel value with the corresponding mask value. It then proceeds by shifting the mask until every pixel value is updated with the sum of these multiplications. The middle value of the mask sits on top of the pixel value that is to be updated.
In the following sections, we will describe the different design choices that were used, that is, we will discuss the different memory architectures and the effect they had on the performance. there are a few tests that we will run: different sizes of input images, different dimensions of masks (for sharpening and averaging effects), we will test the number of global memory reads and how this effects the performance and we will look into floating point computation rates for CPU and GPU kernels.

2 Design choices and effect on performance

The basic algorithm is the same in each different function, however other features were implemented from a trial and error base. For constant memory, the mask was chosen to go into constant memory as it is not big enough to hold the image. This implementation performs well. Shared memory takes in tiles of the input image only while the mask sits in constant memory. Texture memory takes in the image and the mask is left in global memory. For all functions, an S value is used to represent the offset of the mask from its middle element. For example, a $3x3$ mask will have an S value of 1 as there is one element to the left, right, top and bottom of the middle value. A $5x5$ mask will have an S value of 2. The S value allows us to make sure that the middle mask element that sits on the pixel we are currently updating has the index (0,0) by starting the loop from -S. Another design choice was the use of padding. Padding can be applied to the image in the size of S. S rows and columns will be added to the border of the image. Each padded value is set to 0. This is to accommodate for when the mask is sitting on edge pixels of the image but requires elements from outside the image to compute. With padding, these elements do exist in the form of 0. This means we are adding 0 values in the summation for those outside pixels. This has the same effect as simply ignoring the elements if they fall outside the image. In this experiment, we have just checked to see if we are outside the image and if so ignored those calculations. When dealing with shared memory, we load a tile of the image into memory. In this case, we can not just pad the tile with 0's because the rows and columns outside the tile are in fact part of the original image. The size of our tile is 16x16. To accommodate for this, we have set a variable called TILE_WIDTH to a certain size, say 14, and then added $2S$ rows and columns to it to create a padding that gets us up to 16. The $14x14$ tile is the active part of the tile and the surrounding pad is inactive. We can see this in Figure 1, the red part is inactive.We check to see if the mask falls in the inactive part

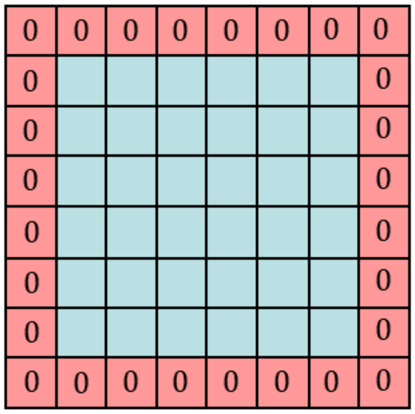


Figure 1: 0 padding

of the tile and if so, we call those elements from global memory where our full image is stored. Although this may not be the optimal design choice for shared memory as we have more global memory access than if we loaded the correct amount of pixels to the tile, it is notably efficient. The fact that the mask is sitting in constant memory already improves the performance as well.
The depth of grey of a pixel is measured by a number ranging between 1 and 0. 1 is white and 0 is black. In certain cases, the summation of the product of each mask element and its corresponding pixel values could return a result that is above 1 or below 0. As a solution to this we have added a threshold to our sum. If the sum value goes above 1 we set it to 1 and if it goes below 0 we set it to 0.

3 Performance comparisons

For a bit of background, let's understand how a serial implementation would work versus a naive parallel one. In a serial implementation, everything will be run on the CPU. Needing to iterate through each pixel requires a double for loop, then needing to iterate through each mask element and perform the multiplication and summation requires another set of for loops. This gives 4 nested for loops which is slow. In a parallel approach, the kernel runs for each thread. Each thread updates one pixel, so there is no need to iterate through each pixel within the kernel. Thus we are left with a double for loop. In the following subsections we give the results of our experiments and in section 3 we will discuss these result.
To create a visual understanding of what we are doing, the resultant images after applying convolution are shown below on an image of Lenna Soderberg. 7 masks were tested. The edge detection mask of size $3x3$ emphasized the edges in the image. The sharpening mask sharpened the image and 3 different sizes were used, $3x3, 5x5$ and $7x7$. For the same 3 sizes, an averaging mask was also used which created a blurring effect. As the size of the mask increases, so does the impact of its effect.

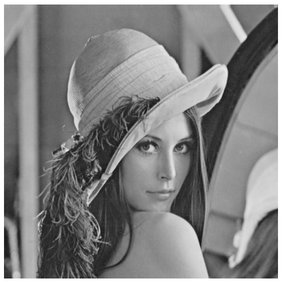
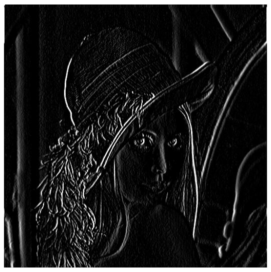


Figure 2: Original image


(a) 3x3 Edge detection mask

(b) 3x3 Sharpen mask

(c) 5x5 Sharpen mask

(d) 7x7 Sharpen mask

(e) 3x3 Averaging mask

(f) 5x5 Average mask

(g) 7x7 Average mask

3 different images with different sizes were tested on. The Lenna image above was of size 512x512, an image of an airfield had size 1024x1024 and an image of a character from the Avengers movie, Thanos was of size 2048x2048. The different implementations were: serial, naive parallel, parallel with constant memory, parallel with shared memory and parallel with texture memory.

Image size	512x512	1024x1024	2048x2048
Serial	11.47ms	46.32ms	173.77ms
Naive	0.10ms	0.36ms	1.36ms
Constant	0.05ms	0.17ms	0.60ms
Shared	0.04ms	0.12ms	0.55ms
Texture	0.10ms	0.32ms	7.99ms

Table 1: Mask: 3x3 Sharpen

Image size	512x512	1024x1024	2048x2048
Serial	28.13ms	116.71ms	437.36ms
Naive	0.24ms	0.93ms	3.61ms
Constant	0.10ms	0.40ms	1.48ms
Shared	0.07ms	0.26ms	1.40ms
Texture	0.20ms	0.75ms	2.68ms

Table 2: Mask: 5x5 Sharpen

Image size	512x512	1024x1024	2048x2048
Serial	54.06ms	209.20ms	820.62ms
Naive	0.46ms	1.79ms	7.11ms
Constant	0.18ms	0.75ms	2.85ms
Shared	0.12ms	0.44ms	2.26ms
Texture	0.36ms	1.39ms	5.65ms

Table 3: Mask: 7x7 Sharpen

Image size	512x512	1024x1024	2048x2048
Serial	10.89ms	44.65ms	171.68ms
Naive	0.20ms	0.35ms	1.35ms
Constant	0.05ms	0.16ms	0.60ms
Shared	0.04ms	0.11ms	0.37ms
Texture	0.10ms	0.31ms	89.69ms

Table 4: Mask: $3x3$ Average

Image size	512x512	1024x1024	2048x2048
Serial	27.79ms	107.29ms	473.10ms
Naive	0.24ms	0.91ms	3.60ms
Constant	0.01ms	0.37ms	1.45ms
Shared	0.07ms	0.25ms	1.07ms
Texture	0.19ms	0.68ms	81.59ms

Table 5: Mask: $5x5$ Average

Image size	512x512	1024x1024	2048x2048
Serial	50.99ms	204.16ms	814.90ms
Naive	0.45ms	1.77ms	7.28ms
Constant	0.18ms	0.72ms	2.97ms
Shared	0.12ms	0.43ms	1.8ms
Texture	0.35ms	1.37ms	83.77ms

Table 6: Mask: $7x7$ Average

Image size	512x512	1024x1024	2048x2048
Serial	12.09ms	48.2ms	182.49ms
Naive	0.10ms	0.37ms	1.39ms
Constant	0.06ms	0.19ms	0.67ms
Shared	0.04ms	0.12ms	0.44ms
Texture	0.10ms	0.33ms	1.24ms

Table 7: Mask: $3x3$ Edge detection

The following table shows the results of the overhead time for the GPU. This is a measure of the execution time within the host function but excluding the execution of the kernel itself(which is shown in the above tables):

Image size	512x512	1024x1024	2048x2048
Naive overhead	57.78ms	86.73ms	148.81ms
Constant overhead	60.90ms	80.42ms	169.99ms
Shared overhead	61.04ms	80.46ms	159.81ms
Texture overhead	5.40ms	21.61ms	82.16ms

Table 8: Overhead time of the GPU. Mask: $5x5$ Sharpen

Image size	512x512	1024x1024	2048x2048
Naive	0.46ms	75.29ms	137.17ms
Constant	61.11ms	80.54ms	161.02ms
Shared	72.69ms	79.94ms	169.97ms
Texture	5.46ms	20.62ms	82.31ms

Table 9: Overhead time of the GPU. Mask: $7x7$ Sharpen

3.1 Different sizes of input images

To compare the effect of different image sizes, lets look at Tables 1, 2 and 3 which all use the Sharpen mask but vary in mask dimensions between tables. Looking at Table 1, if we divide the values in the 2048x2048 column by their corresponding 1024x1024 values, we can obtain a scaling factor. We can then do the same for the 512x512 column.

Implementation	scale factor from 512 to 1024	scale factor from 1024 to 2048
Serial	4.04	3.75
Naive	3.6	3.77
Constant	3.4	3.53
Shared	3	4.58
Texture	3.2	24.9
mean	3.45	8.06

Table 10: Scaling of image sizes for 3x3

Implementation	scale factor from 512 to 1024	scale factor from 1024 to 2048
Serial	4.14	3.7
Naive	3.88	3.88
Constant	4	3.7
Shared	3.71	5.38
Texture	3.75	3.57
mean	3.9	4.05

Table 11: Scaling of image sizes for 5x5

Table 10 shows that on average, scaling the image from 512x512 to 104x1024 scales the time by 3.45. The scaling factor on average for going from 1024x1024 to 2048x2048 is 8.06. However, there seems to be high variance when it comes to shared and texture memory when scaling to higher dimensions. Table 11 calculates the same but for a higher dimensional mask but gives similar results however texture memory performs much better on the high dimensional picture. This shows that scaling an image increases the computation time most likely in an exponential way. This is coherent as the more pixels we have the more times global memory will need to be accessed and the more computations done overall.

3.2 Different sizes of sharpening and averaging convolution masks

The size of averaging and sharpening masks are easy to scale as they have a set form that is directly related to its dimension. Comparing the 3x3 and 5x5 sharpening masks, we can see a huge time increase for the serial implementation. The inner for loop will need to run many more times as the mask size increases which will increase the overall time largely. The time difference is even larger as the size of the image increases. Moving to the 7x7 sharpening mask we again see a big jump in time which is consistent with the previous observation. Shared memory seems to be the least affected by this increase in mask size. This could be due to the fact that the mask sits in constant memory and is only loaded once and read from very quickly. So this is true for when only constant memory is used as well.
The naive parallel approach for the sharpening mask scaled on average by 2.54 between 3x3 and 5x5 masks and by 1.93 between 5x5 and 7x7 masks. Comparing this to the averaging mask, we get that between 3x3 and 5x5 masks, the time scaled by an average of 2.16 and between 5x5 and 7x7 masks it scaled by an average of 1.95. The scaling factors are very similar for each mask, this shows that the values in the mask itself has little to no effect on the time, however the scaling of the dimension makes a notable difference.

3.3 Number of global memory reads

The number of global memory reads done by the kernel can be calculated by hand. The naive approach and serial approach have the same calculation. For every pixel, we access the mask its $dimension^2$ times and we access the pixels for each one of those mask elements. This gives

$$(dimension\ of\ image)^2(dimension\ of\ mask)^2(dimension\ of\ mask)^2)$$

For a 512x512 image using a mask of dimension 3 we get,

$$(512x512)(3x3x9) = 21233664\ global\ memory\ reads.$$

For the constant memory, for every pixel we do not access the mask in global memory, so for every pixel we need only access the global memory for the number of mask elements. Thus we get

$$(dimension\ of\ image)^2(dimension\ of\ mask)^2)$$

For a 512x512 image with mask dimension 3, we get,

$$(512x512)(3x3) = 2359296\ global\ memory\ reads.$$

For shared memory, the kernel accesses global memory in order to initialize the shared memory with the input data. Since the active part of the tile is what includes the input data, there are $TILE\_WIDTH^2$ global memory reads When we check if the the inactive area falls within the image, we then need to access global memory to obtain those pixels. The extreme case is when a tile is in the middle of the image and all the inactive pixels needs to be read from global memory. If our TILE_WIDTH is 14 we are using a 3x3 mask because to get to 16 we need only add 2 which is 2*1, thus S = 1, we will add one row 16 pixels to the top, as well as to the bottom. We will add on column of 14 pixels to either side as we need not account for the corners as the added rows have already. This is 60 pixels that we read from global memory. We will also need to read the mask $dimension^2$ elements of the mask from global memory. Although in this reports case that is not necessary as the mask sits in constant memory, we will include it here. Thus for a 3 x 3 mask, in the extreme case, we have

$$(14x14)(3x3) + 60(3x3) = 2304\ global\ memory\ reads.$$

When the tile fall on the edge of the image we can ignore the inactive pixels that fall outside the image. Thus for those cases we can minus the number of outside pixels times the mask dimensions from the overall calculation. In our case, we need not account for the mask accesses as it is in constant memory, thus in the extreme case, we have

$$(14x14) + 60 = 256\ global\ memory\ reads.$$

Texture mmeory contains the entire image, thus we need only read the mask elements from global memory. This is $(dimension\ of\ mask)^2$ global memory accesses. For a 3x3 mask that is 9 global memory accesses for the kernel.

3.4 Floating-point computation rate for CPU and GPU kernels

We can calculate the giga flops using this formula:

$$GFLOPS = \frac{width \times height \times dimension \times 2}{time \times 10^9}$$

where dimension is the number of elements in the mask. Using this, we will calculate the GFLOPS for the $3/times3$ sharpen mask, 5×5 sharpen mask and 7×7 sharpen mask for the serial implementation and the naive implementation for each image input.

Image size	mask size	512x512	1024x1024	2048x2048
Serial	3	$4.02x10^{-4}$	$4.07x10^{-4}$	$4.34x10^{-4}$
Naive	3	0.047	0.013	0.056
Serial	5	$4.66x10^{-4}$	$4.49x10^{-4}$	$4.8x10^{-4}$
Naive	5	0.05	0.06	0.06

Gflops calculation

3.4.1 How they scale with input size

4 Overhead costs

Tables 8 and 9 show the time taken by the GPU as an overhead. This includes operations such as cudaMalloc() to make space for variables as well as the copying of data from the host to the device and back. We can compare tables 2 and 8 as they show the same executions. We can see that in every case, regardless of image size or implementation method, the overhead takes much longer than the kernel. However, most of the overhead times are faster than the time it takes for a serial execution. Overhead costs are attributed to accessing the GPU. The overhead for copying the mask to the GPU will be constant for each mask size respectively. We note that texture memory has very low overhead costs. We can also compare Tables 9 and 3. We still see that texture memory has low overhead costs compared to the other methods. This could be that binding to texture is very fast. Again we see that despite overhead being much longer than the actual kernel, it is still faster than the serial running.

5 Different CUDA device memory

5.1 Global memory

The naive implementation only uses global memory. Every result table shows us that global memory is the slowest CUDA memory to use. We know that the naive implementation has the highest number of global memory reads so it poor performance is amplified. Global memory has the lowest bandwidth and it resides in device memory.

5.2 Constant memory

Constant memory is an optimized read-only memory. Our results show that constant memory is the second fastest memory to use, however, our shared memory implementation also uses constant memory. The increase in performance using constant memory is very notable considering that the image is not loaded into it and only the mask which is not larger in size. Since the mask never needs to be updated, it makes sense to put it into a read only memory. All the threads in a kernel would be accessing the same locations in constant memory and when this occurs, constant memory can be as fast as registers. This is reflected in the major performance improvement shown in the results.

5.3 Shared memory

Shared memory is a difficult memory to conceptualize in this case. We are loading small tiles of the image into shared memory and we need to keep track of where we are in the overall image, the mask and the tile itself as well as know what the padding of each tile entails. All threads in a block access the same shared memory and syncing of threads is very important. Our results have shown that shared memory with constant memory is the optimal approach for this computation. This approach has minimal global memory reads which plays a significant role in the performance improvement. Shared memory resides on chip as opposed to global memory so storing the pixel data in shared is beneficial as we reuse this data. It also has much more bandwidth and works with less latency. Shared memory and constant memory were quite comparable for the 512x512 images, however we start to see that shared memory scales better than constant with the size of the input image. This can be attributed to the reuse of data. This shows that although our shared memory implementation utilizes constant memory, we can still see the beneficial effects of shared.

5.4 Texture

Texture memory resides in device memory and is cached in the texture cache. This improves performance and reduces memory traffic. The results from our texture memory is comparable with our naive implementation but there is a notable improvement in terms of scaling by the image size. There were a number of times that texture memory produced anomalous data. Texture memory is optimized for 2D spatial locality. So threads that read addresses from texture memory that are close together will achieve better performance. This is something that should be evident in our case since the pixels we read are all in the same vicinity, however a major improvement is not noticed. A possible reason for this is that global memory is chached in L1 which has higher bandwidth than the texture cache, also that the 2D spatial locality is not high enough in some cases.

Something to note about all memory types is that the accesses can vary by an order of magnitude depending on the memory access patterns which is hard to judge. The synchronization of threads must be kept under check as sometimes this can result in the output not looking correct and for race conditions to occur.

6 Anomalies

- The result for 2048x2048 with a 3x3 sharpen mask for texture memory in Table 1 is not coherent. It has a much too large value.
- 512x512 image with a 3x3 sharpen filter has longer overhead times than the running of a serial execution for global, constant and texture memories
- Naive 512x512 overhead for 7x7 sharpen mask
- Texture implementation with a 2048x2048 image on any size average mask has outlying values.

Anomalies that were detected could have resulted from the test being conducted with the incorrect inputs, an incorrect address, a race condition or not coherently setting the sizes of grids and blocks.

7 Challenges

Understanding how to correctly index the different memory spaces was the most challenging part. Also understanding how different memories work and how they are different from one another.

8 Conclusion

In this assignment we have tested the different types of memory spaces offered by CUDA and the GPU using image convolution. Shared memory combined with constant memory has proven to be the most efficient in running the kernel while textured memory had the lowest overhead time. However, we can also conclude that the texture memory implementation might have had a bug as numerous anomalies were detected. It is seen that CUDA has been optimized in various ways to optimize paralellization and although complex, it offers many different approaches to dealing with a parallel problem and makes accessing the GPU and manipulating threads more efficient. Improvements to this experiment could be to run tests on more images of different sizes (in particular larger ones) as well as on rectangular images. Also, combinations of different memories could be tried.

9 References

- Naoyuki Ichimura, Accelerating Convolution Operations by GPU (CUDA), Part 1: Fundamentals with Example Code Using Only Global Memory, Accessed 02-04-2019, https://qiita.com/naoyuki_ichimura/items/8c80e67a10d99c2fb53c
- Naoyuki Ichimura, Accelerating Convolution Operations by GPU (CUDA), Part 2: Utilizing Constant and Shared Memory for Data Reuse, Accessed 08-04-2019 https://qiita.com/naoyuki_ichimura/items/519a4b75f57e08619374.
- Pietro Bonginig, CUDA-ImageConvolution, Accessed 10-04-2019 https://github.com/pietrobongini/CUDA-ImageConvolution