CSC367 Assignment 4: GPUs and CUDA

Submitted By:
Mark Sedhom
Miguel De Vera

Implementation

Best CPU Implementation - Parallel Rows:

The first implementation achieved was a CPU version which processes the image in parallel using POSIX threads. A majority of this was already done in A2, as such we will refrain from discussing our implementation and instead focus on why we implemented this method.

Out of our A2 implementations, we used a Parallel Row Major implementation, as it was shown to have the fastest time. We took our times from A2, applying the 9x9 filter to image 1, and compared the run times we obtained. To the right are the results of running the parallel implementations at 8 threads and the work queue implementation at chunk size two. We did not consider the work queue implementation with variable chunk sizes as there were two many factors, as we had to consider both chunks and threads - we wanted to focus on a general implementation with few complications.

CPU IMPLEMENTATION TIMINGS

Implementation	Time(S)
Sequential	0.11
Parallel Rows	0.03
C-Column Major	0.07
C-Row Major	0.03
Work Pool	0.09

It should be noted that although we only showed one time result, there was a consistent trend in favor of parallel rows. The results of column-row-major appear to be the same as parallel rows, however, the results at most of the other threads counts show parallel rows with a faster time. Additionally, the work pool implementation at eight chunks and three threads was faster than parallel rows, but as stated before: we wanted to focus on a general implementation with few complications.

Helpers & Utilities:

We use three major helpers: kernels.h, reduction_kernel.h, and util.cu. Kernels.h was given to us initially and we later altered it to suit our needs, whereas reduction_kernel.h and util.cu was of our own creation. We added the other two because they were used in multiple kernels and this minimized code duplication. We created these helper files because we needed a way to synchronize thread blocks, however there was no function that allowed us to synchronize. Instead, we synchronize using kernel calls, as a kernel launch serves as a global synchronization point and has low overhead.

In kernels.h, we changed the headers for kernels 1 - 4 to take pointers to the smallest and biggest reduction arrays - otherwise we would have to memcpy them out to the host which would be unnecessary, this way we can just access smallest[0] and biggest[0]. We also added

reduce_kernel.h, call_reduce, and check_threads which are present in reduction_kernel.h and util.cu. Additionally, we include the necessary kernel5 headers, we will discuss this later.

Reduction_kernel.h stores our reduction method: reduce_kernel(), however reduce_kernel() is actually used in util.cu. Reduce_kernel() is used in our reduction step, which is necessary for the normalization. In the reduction we have two shared int arrays, since they are shared we use a value, sep, to note where one ends and one begins. We populate these arrays with the max and min values we find in each thread. Reduce_kernel() finds the largest and smallest values in the image, via the two shared arrays, through reduction by 'dividing' based on tids, which is why we need check_threads(): as we reduce we reduce the values of two threads into one step by step, until we only have a single max and min.

In util.cu we have call_reduce() and check_threads(). Call_reduce() merely calls reduce_kernel() with varying thread sizes. Check_threads() is used to ensure that the number of threads is a power of two so we could properly divide the threads as needed in the actual reduction process, otherwise we couldn't properly execute reduction.

Kernel 1 - Column Major:

This kernel applies the filter by assigning a single pixel to each thread and processing them in a column major order. Like with most of the kernels we make, we first define a way to get a minimum value, MY_MIN and the maximum number of blocks. This kernel is composed of three main functions: kernel1(), run_kernel1(), and normalize1().

Kernel1() is where a majority of the computations happen. To start, we define the row and column of our current pixel - we know this is the thread ID and block ID respectively. We calculate the offset of our current pixel using the row and col values, since our image will be traversed as a one dimensional array. Next, we begin initializing boundary values - just like our approach in A2 we are putting left and right boundaries that tell us the left and rightmost elements of the 'row' that we are currently applying our filter values to. With corner, left, and right initialized we begin calculating the new pixel value by applying the filter to the surrounding pixels. After the new value is found and stored, we change the output's value at the appropriate offset to the new value. Due to the fact that kernel1 only works on a single pixel, our helper function, apply2d(), is not needed.

Normalize1() is, as the name suggests, the normalization step. This function simply normalizes the appropriate pixel given the smallest and largest values, found in run_kernel1(). We covered this already in A2, so to summarize, it takes the smallest and biggest values we found in kernel1 and scales the target pixel in the output to an acceptable range depending on the smallest and largest values.

Run_kernel1() brings everything together. This function performs the actual application of the filter and prints the runtime of execution. A majority of the code is actually the initializing, storing and transferring of the necessary data. First, we copy the input, output, and filter from host to the device. We then call kernel1 in the device and feed it the copied arguments. Next, we find

the appropriate amount of block, threads, and shared memory size for the reduction. Additionally, we store the smallest and largest values found from the output of kernel1. After the reduction, we normalize the image and transfer the output to the host. Each individual step, transfer in, kernel1 computations, and transfer out, is timed. At the end of run_kernel1() we print the cpu time, gpu time, and the transfer times.

Kernel 2 - Row Major:

Due to the nature of kernel2, there isn't much difference between kernel2 and kernel1. Kernel1 processes the pixels in a column major order and kernel2 does it in row major order. As such, the only difference between kernel1 and kernel2 is switching the rows and columns, and the width and height. We do this in places such as the start of kernel1 and kernel2 when finding the current pixel, and in normalize Otherwise, the function is identical to kernel1.

Kernel 3 - Multiple Pixels; Consecutive:

This kernel applies the filter by assigning a multiple pixels to each thread and processing each pixel consecutively (row 1 then row 2) with the pixels covered in row major order. This kernel is composed of five main functions: kernel3(), run_kernel3(), apply2d3(), normalize_one3(), and normalize3().

Kernel3 starts by finding the start and end of the row we will work with. We introduced a macro, SPLIT, which affects the number of blocks - named as such because it decides how we split the height. Once we have the starting and ending points, we make sure they are valid, we begin traversing the pixels, in order, to apply the filters. We traverse the columns of the input image and apply the filter to the rows as needed using apply2d3(). As the filter is applied, the appropriate resulting row is stored into the output pointer, which will eventually be used in run_kernel3().

Apply2d3() is a helper function that was present in our A2. It applies the filter to a pixel and directly stores the result into the output image. Because we covered it previously in A2, we will be brief. Apply2d forms a left and right barrier to isolate the zone of appropriate data we will work on. Next, it loops through the pixels surrounding our current pixel and applies the filter to it and adds the results to the current pixel we are altering. It then copies the final resulting pixel to the output.

Normalize_one3() is actually identical to the normalize functions in kernel1 and kernel2, the main difference being that row and col are now passed into the function as arguments, from normalize3().

Normalize3() is different from the normalize functions of kernel1 and kernel2 in that it normalizes several pixels in succession and not just a single one. Despite this difference, the process is still very similar. We find our partition of work using the start and end variable, just like in kernel3, and then we simply traverse said partition, applying normalize_one3() to each pixel.

Run_kernel3() puts all of the functions together. Thanks to the structure of our code, this function is very similar, if not identical, to run_kernel1. We copy the input, output, and filter from host to the device. We then call kernel3 in the device and feed it the copied arguments, however, unlike kernel1 and kernel2, we don't use width or height for the rows or blocks. Instead, we use SPLIT and height/SPLIT respectively due to the nature of kernel3 processing multiple pixels in succession - with SPLIT being a hardcoded value of two. What this means is we decided to split it in two blocks and there is a thread for every two rows in the image.The rest of the code is more or less similar to kernel1, though. We find the appropriate amount of block, threads, and shared memory size for the reduction. We store the smallest and largest values found from the output of kernel3. We reduce and normalize the image and transfer the output to the host. The main differences lie in the helper functions, such as normalize3 working on multiple pixels unlike normalize1 or normalize2, but the overall structure is the same.

Kernel 4 - Multiple Pixels; Sharded:

This kernel applies the filter by assigning a multiple pixels to each thread and processing each pixel assigned and accessed by a stride equal to the number of threads. This kernel is composed of five main functions: kernel4(), run_kernel4(), apply2d4(), normalize_one4(), and normalize4(). Many aspects of kernel4 are similar to kernel3.

In kernel4(), you will notice that there is no 'end' variable anymore, it has been replaced with a stride variable. This is because we assign multiple threads, like in kernel3, but unlike kernel3 we do not access whole rows at once, as such we do not need to partition based on a start and end value - instead we partition based on the stride. We see this in the loop, where the loop variable is incremented by stride each time.

Both apply2d4() and normalize_one4() are identical to their kernel3 counterparts, which is to be expected as kernel4 is just kernel3 with different data partitioning. We redefine them in kernel4 because we found calling device functions from another file to be a hassle, as we would have to do things like changing the flags used - while there is code duplication we found this to be much easier and since the code for these two are small enough, we thought it would be fine.

Normalize4, like most of kernel4, is very similar to its kernel3 counterpart - however there are a couple of important changes. Firstly, you will notice the lack of 'end' variable, for the same reason as the one given in the kernel4() explanation. Next, you will notice the values of start and stride. Start's value comes from the fact that we traverse the image in a sharded manner and as such our starting location is also sharded, which thus depends on the block and threadIDs as well as the block dimension. Similarly to kernel4(), our internal loop where nomralize_one4() is applied is incremented by a strided amount.

Run_kernel4() is very similar to run_kernel3(), except for the values of the threads and blocks in the call to kernel4. Kernel3 has them as 2 and height/2 whereas kernel4 has them as 4 and 32, blocks and threads respectively. This is because kernel4 is the same as kernel3 in the sense that they both process multiple pixels, except kernel3 does it in consecutive rows whereas

kernel4 does it in sharded rows. We did this because there are 32 threads in a warp and four, we chose as a middle ground between having too little and too many blocks.

To conclude kernel4: it is very similar to kernel3, but the strided traversal proved to be quite a challenge, with the key lying in the start and stride variables in kernel4() and normalize4(). Stride, decided by the variable SPLIT, is what truly differentiates kernel4 from kernel3.

Kernel 5 - Optimized:

Throughout our code, we already kept in mind optimizations we could make to improve all of our kernels. These include:

- Changed the header to the normalize function to take in pointers for smallest and largest, which stopped us from having to memcpy to the host to access elements
- Made a header file that included a reduction kernel based off of lab10's 7th kernel. This was the fastest and most efficient kernel for reduction, as it opened all loops and we didn't have to worry about synchronization
- Made the print function available to all kernels by bringing it to util.cu, as such we did not have to spend more time recording different times i.e reported time was purely what was advertised and does not include any additional actions.
- We chose to do our reduction in the gpu, which allowed for faster computation and more accurate GPU time recording as we didn't have to subtract cpu reduction from calculation.
- It is noteworthy that points two and three improve our transfer-time and not gpu time.

There are three functions in kernel5: run_kernel5(), kernel5(), and normalize5(). In terms of code, kernel5 is very similar to kernel2. Below are the main differences.

Kernel5 was based off of kernel2, which we found had the overall best times. We stopped using memcpy as it was unnecessary - since we overwrote it later. We tried to implement striding in kernel5 by calculating current offset and adding it according to memory management. We changed the header as we no longer memcpy filter from host to device as it was unnecessary since we know what the filter is already and we can have it ready in the kernel without wasting time memcpying. We used our filter knowledge to implement a check that stated if the pixel we are working on is within 9 pixels of any border, we set it to 0 immediatly since the sum of the filter we work with is 0, therefore we know that the pixel multiplied to the sum would also be 0, ex: (ax1 + ax2 = a(x1+x2)). This means a lot less loop traversal, especially for bigger images.

Data and Analysis

Note:

- Speedup-1 = Speedup no transfer
- Speedup- 2 = Overall speedup
- There are some notable outliers, as you will see - however we were unable to retrieve more data due to time constraints

Table 1: Standard parameters with 10mb Image

Kernel	GPU Time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup-1	Speedup-2
1	2.565	1.01	1.483	24.46	12.39
2	0.310	0.890	0.432	203.02	38.46
3	23.132	0.891	0.455	2.71	2.56
4	25.267	0.887	0.575	2.48	2.35
5	0.200	0.470	0.509	314.66	53.28

- CPU time: 62.750

Table 2: Varying SPLIT values and results - Kernel4

SPLIT	GPU Time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup-1	Speedup-2
1	2810.262	25.899	36.186	0.27	0.26
2	1389.417	26.590	35.350	0.53	0.51
3	948.454	24.210	35.344	0.78	0.73
4	706.090	24.000	34.466	1.00	0.92
10	301.775	24.494	36.744	2.36	1.96
100	36.280	21.791	33.337	17.29	6.86
1000	12.479	39.134	66.474	78.25	8.27
10000	11.644	22.259	33.514	58.14	10.04
50k	9.724	22.311	33.618	64.37	9.53
max*	9.388	73.677	34.697	68.52	7.87

- Max* = 65535
- The image used was 5000x5000
- To avoid a big table, we removed the cpu times, they are included below:
- 761.215, 733.921, 739.140, 703.891, 711.117, 627.250, 976.539, 677.078, 625.963

Table 3: Kernel5 with varying images

Size*	GPU Time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup-1	Speedup-2
1	0.027	0.012	0.010	148.85	82.53
10	0.026	0.010	0.010	842.85	474.42

100	0.030	0.017	0.016	60.64	28.13
1000	0.652	1.08	1.04	121.72	28.50
5000	0.782	9.463	20.242	800.020	15.85
10000	2.508	49.510	123.731	934.34	13.34

- Size refers to the dimensions of the images, so we have 10x10 for the second row

Table 4: Kernel4 with varying images

Size*	GPU Time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup-1	Speedup-2
1	0.180	0.021	0.010	22.86	19.52
10	0.184	0.025	0.010	121.62	102.27
100	0.184	0.034	0.015	9.92	7.82
1000	0.476	0.891	0.415	166.69	44.52
10000	47.26	92.542	140.020	49.59	13.34

- Size refers to the dimensions of the images, so we have 10x10 for the second row

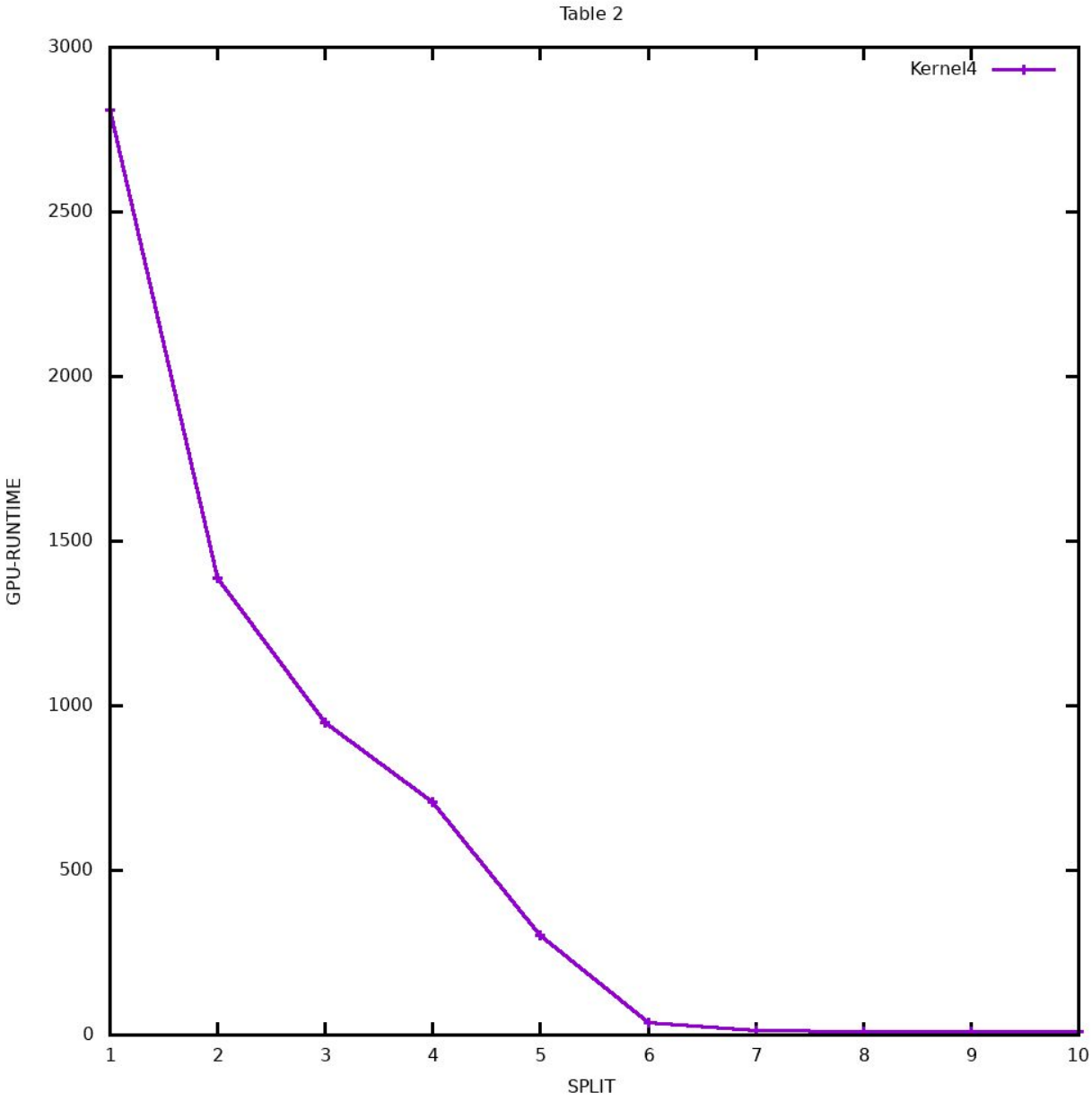


Table 2
Kernel4
GPU-RUNTIME
SPLIT
3000
2500
2000
1500
1000
500
0
1
2
3
4
5
6
7
8
9
10

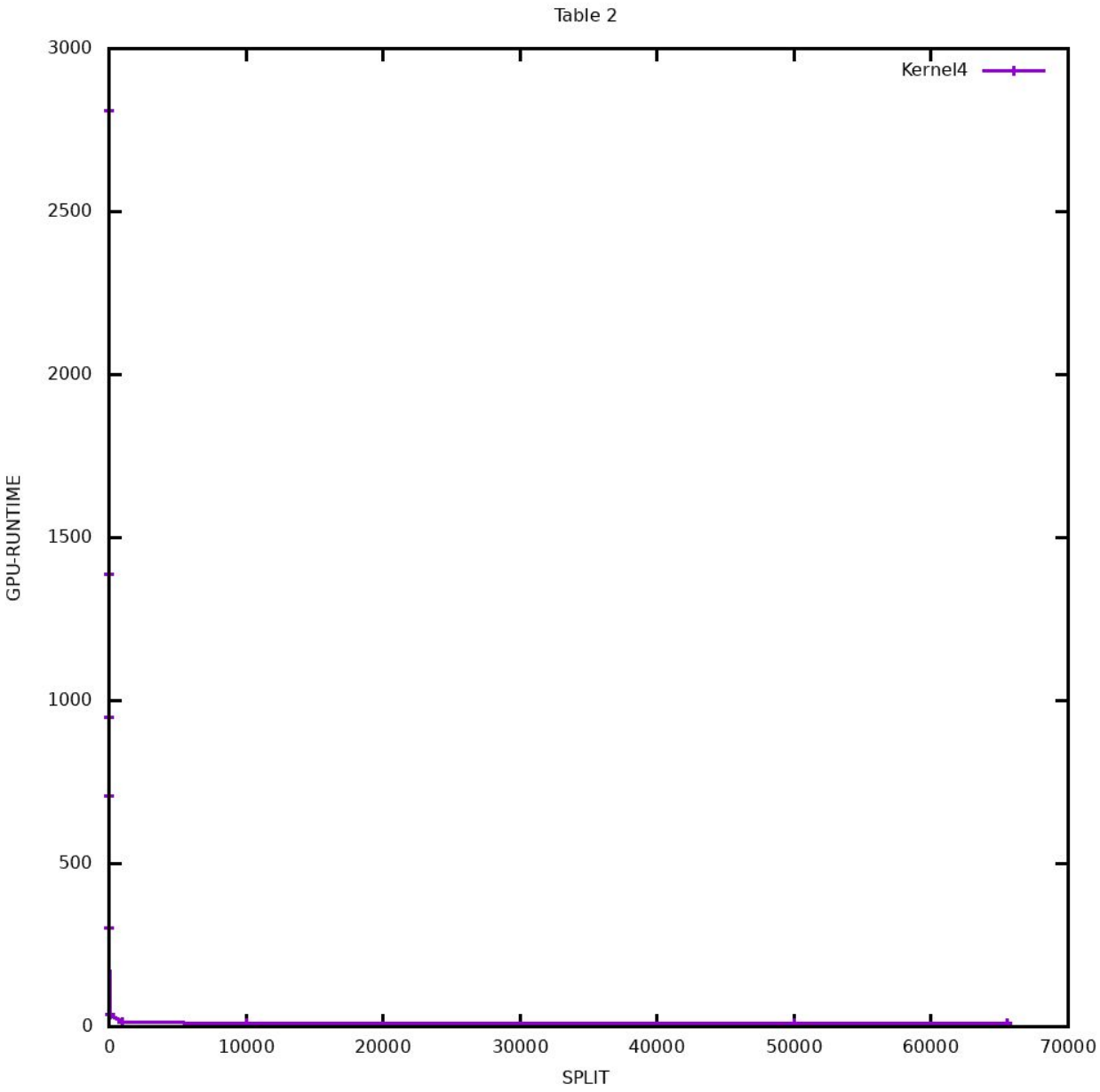
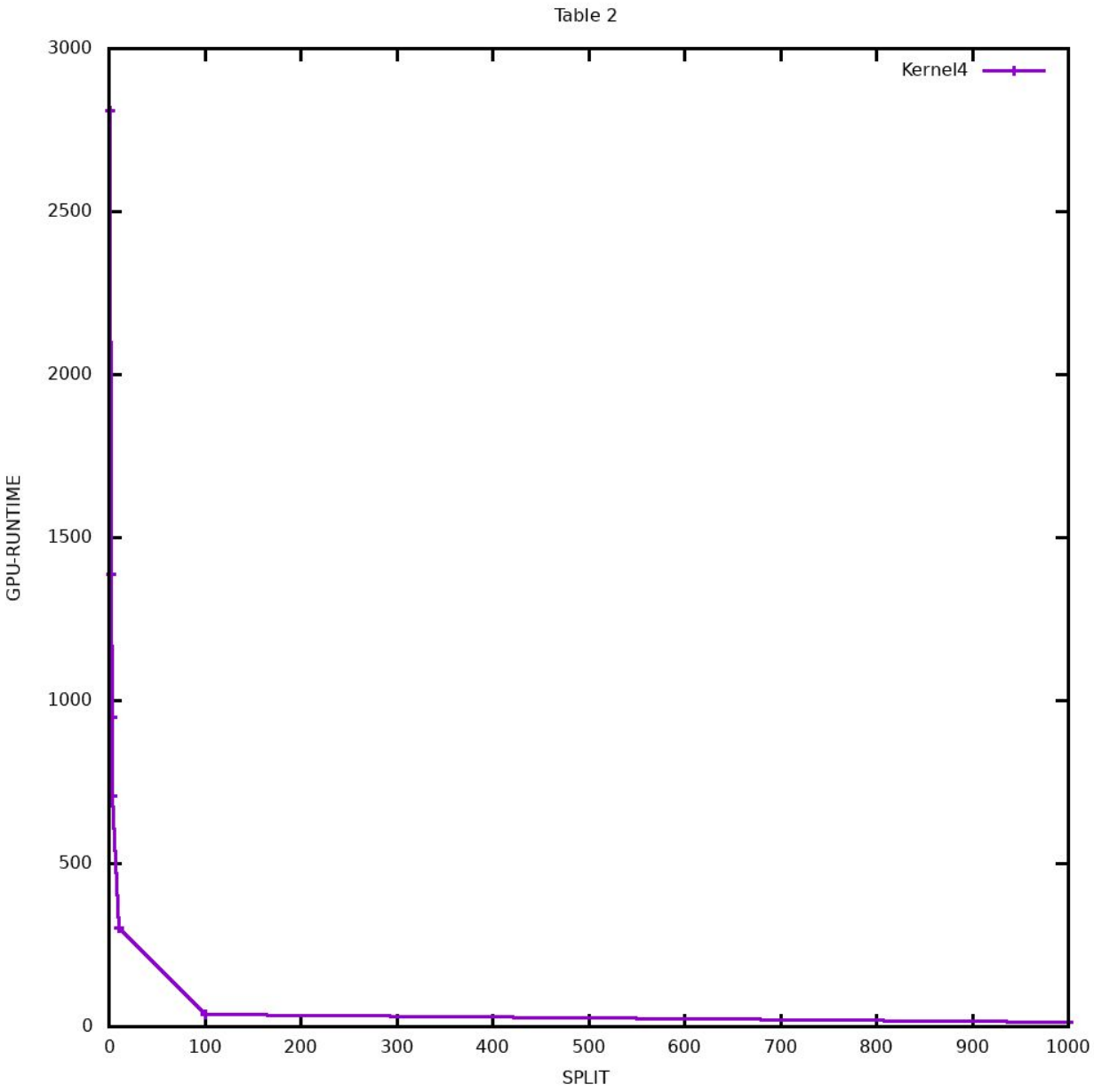
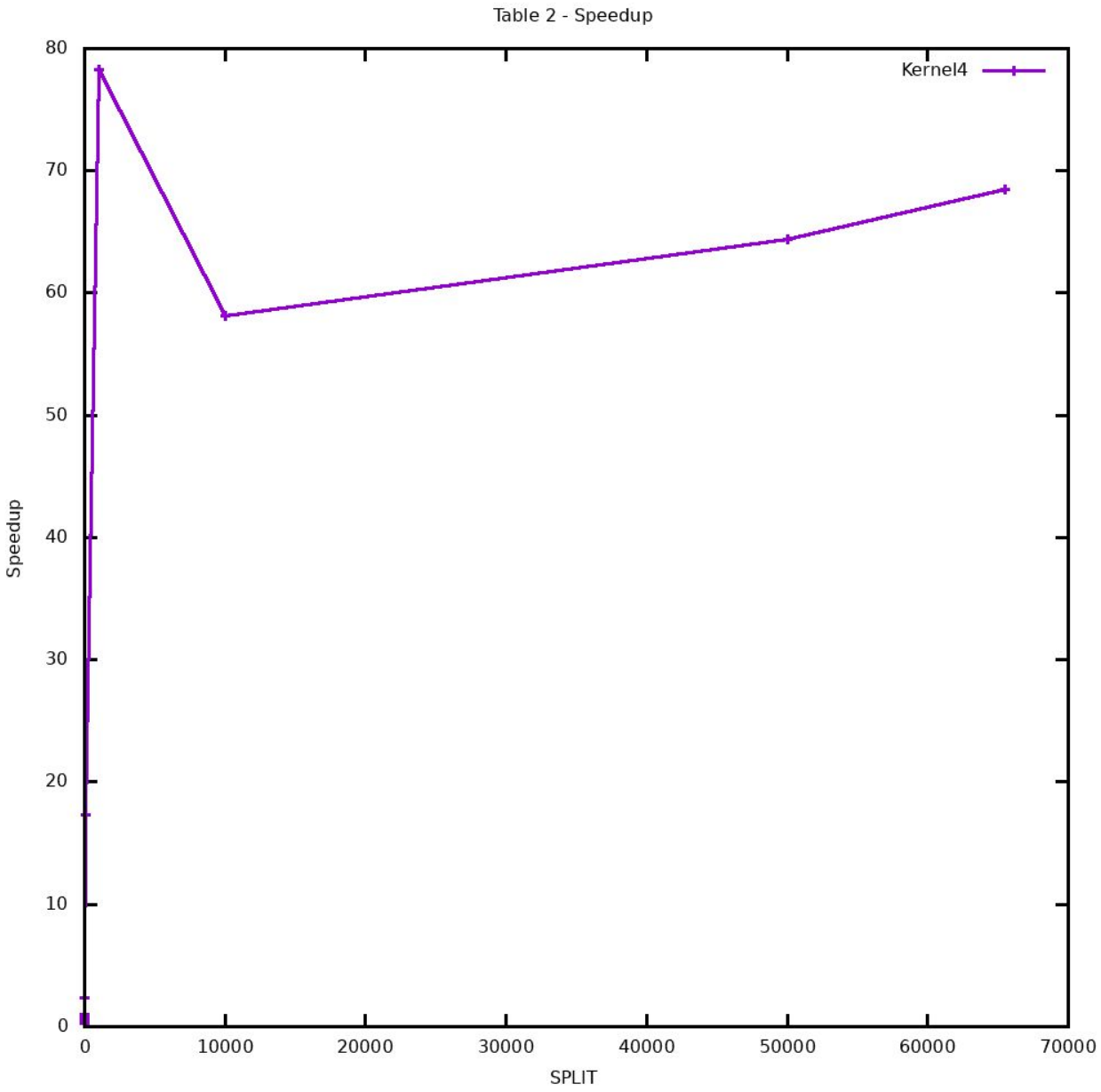


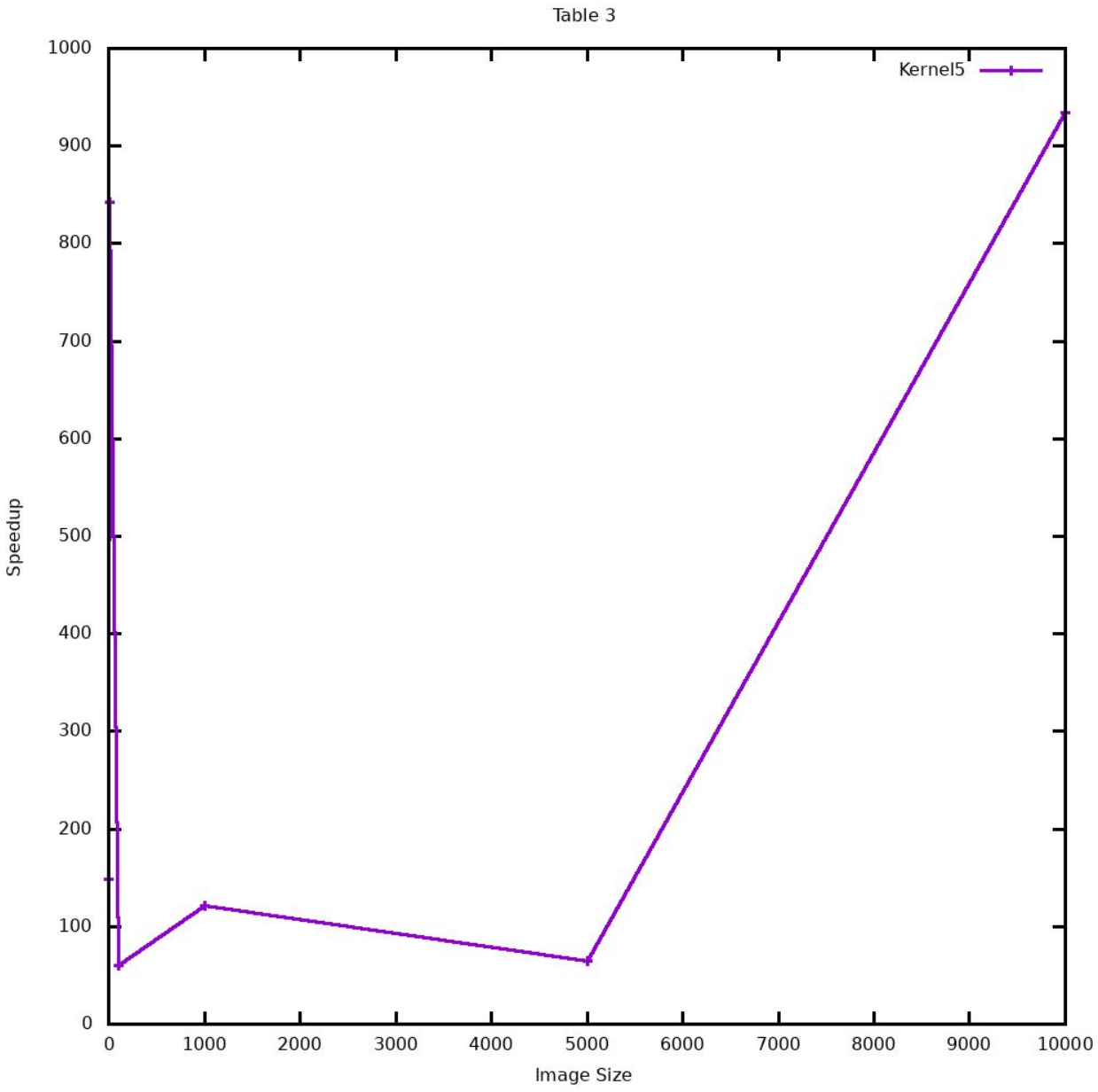
Table 2
Kernel4
GPU-RUNTIME
SPLIT
3000
2500
2000
1500
1000
500
0
0
10000
20000
30000
40000
50000
60000
70000



The above three graphs are a visual representation of the relationship of the GPU runtime of kernel 4 when used on an image with dimensions 5000 x 5000.The third graph is simply the second but without the SPLIT values that were over 1000.



The above graph is a visual representation of the relationship of the speedup (without transfer) of kernel 4 when used on a image with dimensions 5000 x 5000



The above graph represents the relationship between the image size and the speedup when tested on kernel 5 (image size referring to the dimension of a square image).

Conclusion

From the results above a few things can be concluded.

Firstly, it seems that allocating finer grain blocks seem to have a greater impact on performance. This is shown by the graphs on kernel 4 where GPU time decreases drastically and speedup increases in a similar fashion (initially). The change in runtime and speedup both seem to decrease as the number of blocks increase, probably due to the limited number of cores.

Furthermore, from the results above it can also be deduced that allocating a pixel per thread is very scalable. This is shown by the graph on kernel 5 where the speedup increases notably as the image size also increases.

From the two deductions made above it's clear that finer grained block distribution in the GPU results is an overall faster run time and is more scalable. However, there is a limit to everything, and this increase is not linear, rather the increase itself decreases as task management becomes more fine.

Final Note:

Unfortunately, we did not have enough time to fully carry out experiments and analysis. We did our best but felt it necessary to mention that this assignment, particularly the report, could have achieved better results.