Assignment 4
Optimize Convolutions
Syed Hassan Jalil - 11161604
Trisha Anand - 11434104

General Convolution

Code

For the first part of the assignment we were supposed to implement a simple general convolution algorithm in C which would be able to work on both 3x3 and 5x5 kernel. A framework was provided to us to help us get started. The started code was made for 5x5 kernel and we had to adopt it to make it general and work for both 3x3 and 5x5 kernel.

This was done using the following code :

```
void convolution(data_t *inData, data_t *outData, const int width, const int height,data_t
*filter) {
    for ( int y = 0; y < height; y++ ) {
        for ( int x = 0; x < width; x++ ) {
            unsigned int filterItem = 0;
                for ( int fy = y - STARTOFFSET; fy < y + ENDOFFSET; fy++ ) {
                   for ( int fx = x - STARTOFFSET; fx < x + ENDOFFSET; fx++ ) {
                       if ( ((fy < 0) || (fy >= height)) || ((fx < 0) || (fx >= width)) ) {
                           filterItem++;
                           continue;
                       }
                 outData[(y * width) + x] += inData[(fy * width) + fx] * filter[filterItem];
                 filterItem++;
             }
         }
     }
}
```

Here for the inner loops (fy and fx) instead of running from y-2 to y-3, we created values STARTOFFSET and ENDOFFSET. The values of this depends on which kernel we are using (passed as an input parameter). Also, the value of the filter array also depends on the kernel

```
if(kernel==3){
               printf("inside kernel\n");
               filter=(data_t *)malloc(9*sizeof(data_t));
               filter=filter3x3;

       STARTOFFSET=2;

       ENDOFFSET=1;
               printf("Start offset: %d\n",STARTOFFSET);
```

```
	}
	else if(kernel==5){
		filter=(data_t *)malloc(25*sizeof(data_t));
		filter=filter5x5;


	}else{
		printf("Wrong Value for kernel, exiting\n");
		return -1;
	}
```

This way we made it a more general convolution program.

<u>Evaluation</u>

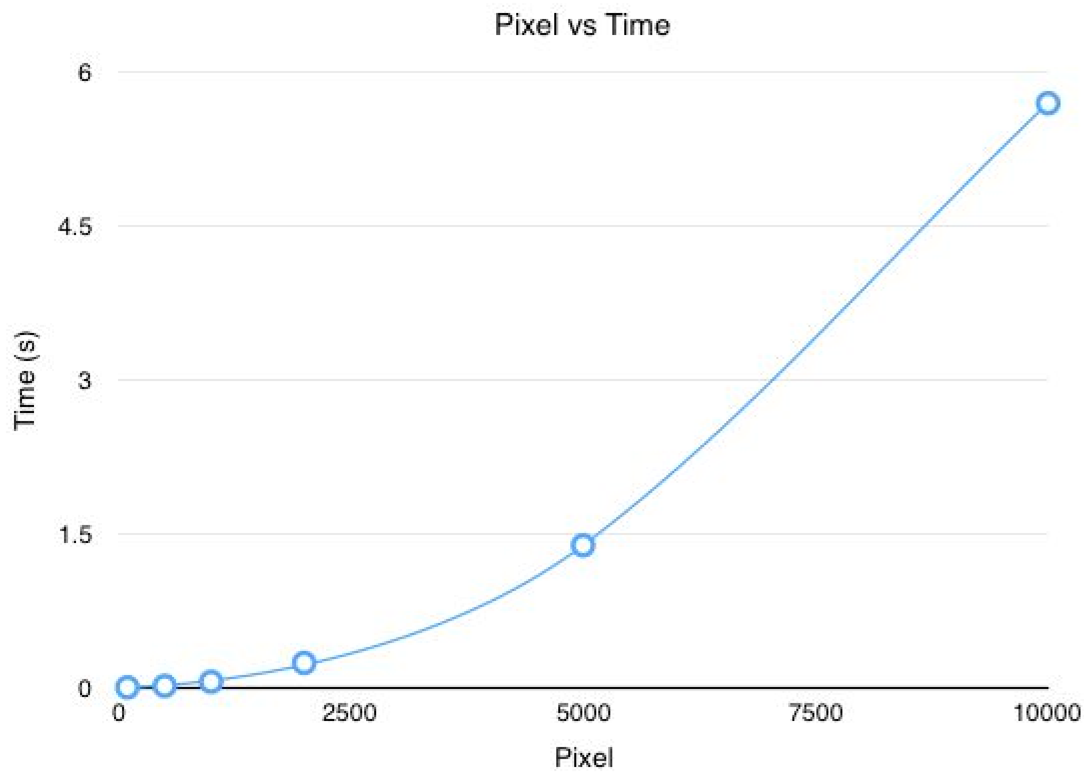
The code was evaluated on our local machine, that is running an Intel Core i5-6360U, 8GB of RAM and MacOS Sierra version 10.12.4.

All the test were carried out for 5x5 kernel.

The image were generated randomly in code using the math rand function and we did not use actual images for the tests.

Image Size	Time (s)
100x100	0.000657
500x500	0.015052
1000x1000	0.055844
2000x2000	0.239176
5000x5000	1.384061
10000x10000	5.688193

We can visualize this in the following scatter chart



Optimizing General Convolution

After this, we had to optimize the general convolution for 3x3 and 5x5 kernels. We applied the following optimizations :

Optimization 1 : Code Motion

The first optimization we applied was by moving some of the code outside the loop, we did this for index of certain arrays and other values.

Optimization 2: Move If outside of the loop

We decided to split the if condition inside the fx loop into two parts, one that checks if fy<0 OR fy>=height and moved it outside the loop, and the other that checks for fx

```
for ( int fy = y - STARTOFFSET; fy < y + ENDOFFSET; fy++ ) {
    int innerIndex=fy*width;
    if(fy<0||fy>=height){
        printf("Inside if\n");
        filterItem=filterItem+STARTOFFSET+ENDOFFSET;
        continue;
}
    for ( int fx = x - STARTOFFSET; fx < x + ENDOFFSET; fx++ ) {
       if ( ((fx < 0) || (fx >= width)) ) {
             filterItem++;
```

```
                continue;
        }
        outData[outterIndex] += inData[(innerIndex) + fx] * filter[filterItem];
        filterItem++;
}
```

Optimization 3: Removing If conditions

So far, we have made simple optimizations that have shown small amount of improvements. The if condition we moved out is rarely true and hence does not provide us with enough benefits. To get better speed ups, we should remove the if conditions from the main kernel. The if condition checks if current index is outside the bounds of the image. We only need to do these checks when our kernel is running along the edges of the image. If we split the kernel into three parts, the center, the upper edge, and the lower edge, we can remove the if conditions from the center part which accounts for the most amount of iterations.

We do this for both width and height. We will use the following image to explain how we divided our image

We first calculate the blue part which is the center, and hence there is no need for if conditions because we are away from the edges and our index will never be out of bound from the images. Next we calculate the red parts, here we only need to do a check for fx as we are away from the y edges and hence won't be out of bound. Finally we calculate the green parts where we have to have the if conditions for both fx and fy.

Since the Blue part forms majority of our calculation, we get good speed ups by removing the if conditions. The Green and the red part are very small, so having if in them does not negatively affect us that much

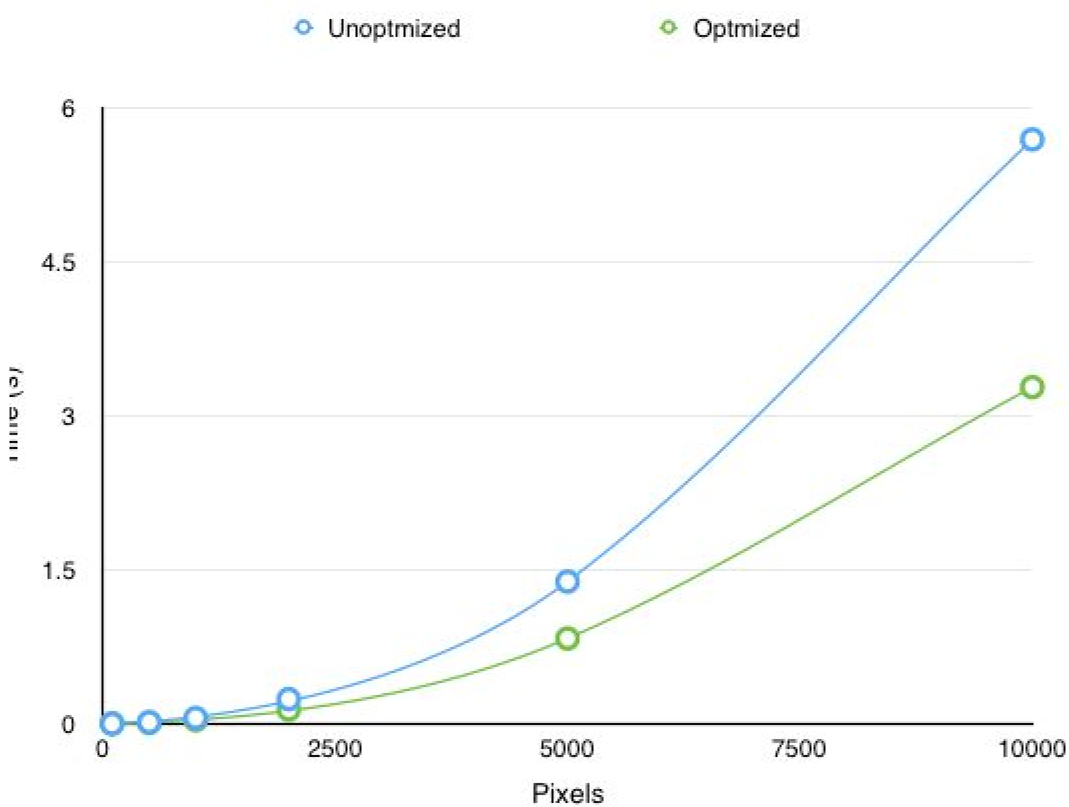
Evaluation

Again, we are evaluating on the same machines

Size	Time opt1 (s)	Time opt2(s)	Time opt3 (s)
100x100	0.000647	0.000645	0.000410
500x500	0.014757	0.014435	0.008887
1000x1000	0.055645	0.054754	0.034983
2000x2000	0.227232	0.219147	0.138092
5000x5000	1.320397	1.284438	0.830404
10000x10000	5.365361	5.298825	3.277270

Size	Speedup opt1	Speedup opt2	Speedup opt3
100x100	1.01545595054096	1.01860465116279	1.60243902439024
500x500	1.01999051297689	1.04274333217873	1.69370991335659
1000x1000	1.00357624224998	1.01990722139022	1.59631821170283
2000x2000	1.05256301929306	1.09139527349222	1.73200475045622
5000x5000	1.04821580176265	1.07756154831919	1.6667320966662
10000x10000	1.06016966985073	1.07348195118729	1.73564979388332

As we can see, we got almost 2 times speedup by our last optimization. We think it's quite a good result, we can compare the performance of our unoptimized code and optimized code from the following chart



Performance Model for optimized convolution

Roofline Model :

Innermost loop, the number of computations = 4, and the number of memory accesses = (1 + ¼ + ¼) * 4 bytes = 6 bytes

Arithmetic Intensity, AI = Total computations / Total memory access = 4/6 = ⅔ = 0.67 FLOPs/byte

We will now use the values from DAS4 to check if its compute bound or memory bound
MIN(4.8, (0.67 * 25.6)) = MIN (4.8,17.152) = 4.8
Based on the roofline model, this is compute bound, and hence we can get speed ups by parallelizing it.

Basic Model :

For the inner part of the image, the following performance basic model applies :
T = T_compute + T_memory + T_communication

In this case, T_communication = 0 because of no outside communication.

T_compute = T_addition + T_multiplication

T_addition = ((height - start_offset - end_offset) * (1 + (width - start_offset - end_offset) * (2 + (start_offset + end_offset) *(1 + (start_offset + end_offset) * 4)))) * t_addition

T_multiplication = (height - start_offset - end_offset) * (width - start_offset - end_offset) * (1 + (start_offset + end_offset) * (1 + (start_offset + end_offset))) * t_multiplication
T_memory = (height - start_offset - end_offset) * (width - start_offset - end_offset) * (start_offset + end_offset) * (start_offset + end_offset) * (4 + 4/16 + 4/16) * 4 * t_memory

Therefore, T = T_compute + T_memory

T = ((height - start_offset - end_offset) * (1 + (width - start_offset - end_offset) * (2 + (start_offset + end_offset) *(1 + (start_offset + end_offset) * 4)))) * t_addition + (height - start_offset - end_offset) * (width - start_offset - end_offset) * (1 + (start_offset + end_offset) * (1 + (start_offset + end_offset))) * t_multiplication + (height - start_offset - end_offset) * (width - start_offset - end_offset) * (start_offset + end_offset) * (start_offset + end_offset) * 18 * t_memory

Triangular Smoothing

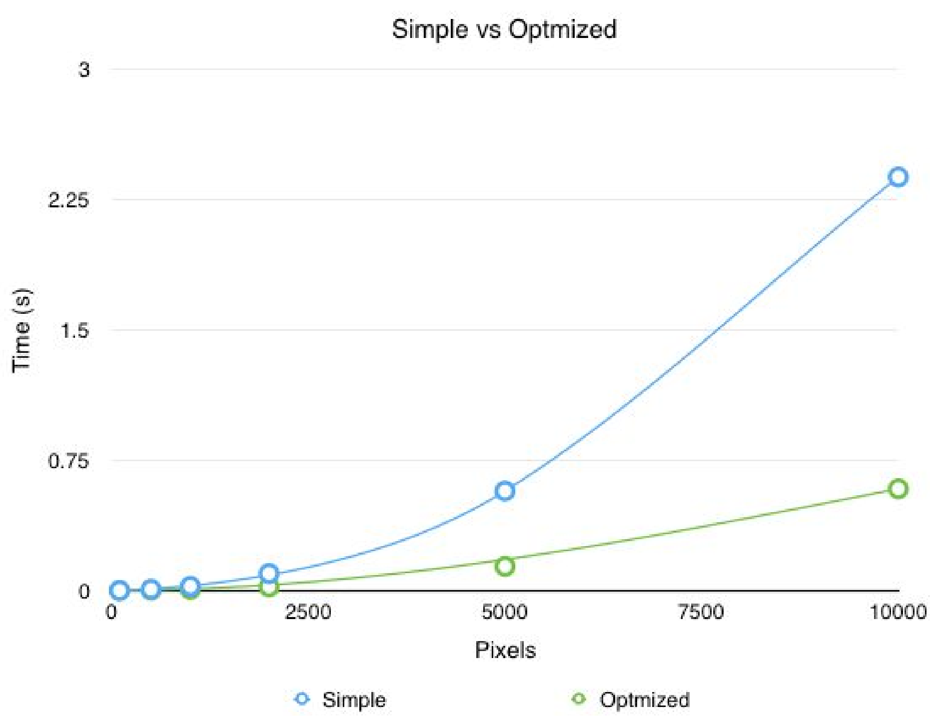
Triangle smoothing is a 5x5 kernel, which is quite similar to our general convolution. There is only a small difference in calculation, otherwise, reading the image remains the same. For this reason we used the same code as that of general convolution, moreover we applied the same optimizations. Hence we won't discuss all the details again here and just share the results of the evaluation.

Evaluation

For this we will just compare the results of unoptimized version and the fully optimized version (all three optimizations applied). We are again using the same machine and same OS

Size	Time simple (s)	Time opt (s)	Speedup
100x100	0.000266	0.000088	3.02272727272727
500x500	0.007351	0.001495	4.91705685618729
1000x1000	0.024136	0.005817	4.14921780986763
2000x2000	0.097827	0.021180	4.61883852691218
5000x5000	0.571970	0.138391	4.13299997832229
10000x10000	2.376158	0.584336	4.06642411215465

As we can see, we are getting better speedups than general convolution, even though the optimizations are same. The reason for this is that we have more memory access in general convolution than we have in triangle smoothening. In general convolution outData is written in the innermost loop while we write to it only inside the second loop. For this reason the memory overhead of general convolution is higher, making the effect of optimization less. This can also be seen by comparing the AI of both these algorithm
We can visualize this better using the following scatter plot



Performance Model for optimized triangular smoothing

Roofline Model

In the innermost loop, total number of computations = 5, total number of memory accesses = ¼ + ¼ + ¼ = ¾

Arithmetic Intensity = Total number of computations / Total number of bytes accessed
= 5 / ¾*4 = 5/3 = 1.67 FLOPs/byte

We will now use the values from DAS4 to check if its compute bound or memory bound
MIN(4.8, (1.67 * 25.6)) = MIN (4.8,42.752) = 4.8

This is compute bound program and would benefit from being parallelized.

Basic Model

T = T_compute + T_memory + T_communication

In this case, T_communication = 0 because of no outside communication.

T_compute = T_addition + T_multiplication

T_addition = ((height - start_offset - end_offset) * (1 + (width - start_offset - end_offset) * (2 + (start_offset + end_offset) *(1 + (start_offset + end_offset) * 5)))) * t_addition

T_multiplication = (height - start_offset - end_offset) * (width - start_offset - end_offset) * (1 + (start_offset + end_offset) * (1 + (start_offset + end_offset))) * t_multiplication

T_memory = (height - start_offset - end_offset) * (width - start_offset - end_offset) * (start_offset + end_offset) * (start_offset + end_offset) * (4/16 + 4/16 + 4/16) * 4 * t_memory

Therefore, T = T_compute + T_memory

T = ((height - start_offset - end_offset) * (1 + (width - start_offset - end_offset) * (2 + (start_offset + end_offset) *(1 + (start_offset + end_offset) * 5)))) * t_addition + (height - start_offset - end_offset) * (width - start_offset - end_offset) * (1 + (start_offset +

end_offset) * (1 + (start_offset + end_offset))) * t_multiplication + (height - start_offset - end_offset) * (width - start_offset - end_offset) * (start_offset + end_offset) * (start_offset + end_offset) * (3) * t_memory

Sobel Operator

The **Sobel operator**, sometimes called the **Sobel**–Feldman **operator** or **Sobel filter**, is used in image processing and computer vision, particularly within edge detection algorithms where it creates an image emphasising edges. Sobel is 3x3 Kernel. We were supposed to implement sobel operator and optimize it in this part of the assignment.
We first had a simple sobel implementation. Since sobel does not cater to the edges of the images, there was no need for any if conditions.

In sobel, you have a horizontal filter and a vertical filter, you apply both to the image, take the square of both the results, add them and then take the square root. This gives you the final resulting pixel from the filter.

NOTE: This is what we understood the sobel implementation is, we found this in a number of different sources that explained it the same way. In case it is wrong, it should not really effect the assignment, as our task is to optimize it and improve the performance, not to implement a the perfect sobel operator. As long as the results of simple and optimized sobel operator is same, this should be good enough (We did do a sanity check to make sure the results are the same)

Our simple implementation is given below

```
const data_t filterX[] = {-1.0f, 0.0f, 1.0f, -2.0f, 0.0f, 2.0f, -1.0f, 0.0f, 1.0f};
const data_t filterY[] = {-1.0f, -2.0f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 2.0f, 1.0f};

for ( int y = 1; y < height-1; y++ ) {
            for ( int x = 1; x < width-1; x++ ) {
                    int outterIndex=(y * width) + x;
                    unsigned int filterItem = 0;
                    float magX = 0.0f;
                    float magY = 0.0f;

                    for ( int fy = 0; fy <  3; fy++ ) {
                            int innerIndex=fy*width;
                            for ( int fx = 0; fx <3; fx++ ) {
                                    int xInd = x+fx-1;
                                    int yInd = y+fy-1;
                                    int index= xInd+yInd*width;
                                    magX+= inData[index]*filterX[fx+(fy*3)];
                                    magY+= inData[index]*filterY[fx+(fy*3)];
                            }
                    }
                    outData[x+(y*width)]=sqrt((magX*magX)+(magY*magY));
            }
    }
```

```
}
```

Optimization 1: Code Motion

Like previously, the first optimization we made was to have some code motion out of the loops.

Optimization 2: Loop Unrolling

Since we know exactly how many times the inner two loops are executed, and what exact values are calculated in them, we can easily fully unroll this loop.

```
magX=inData[(x-1+((y-1)*width))]*filterX[0]+inData[(x+((y-1)*width))]*filterX[1]+
inData[(x+1+((y-1)*width))]*filterX[2]+inData[(x-1+((y)*width))]*filterX[3]+
inData[(x+((y)*width))]*filterX[4]+inData[(x+1+((y)*width))]*filterX[5]
+inData[(x-1+((y+1)*width))]*filterX[6]+inData[(x+((y+1)*width))]*filterX[7]+
inData[(x+1+((y+1)*width))]*filterX[8];

magY=inData[(x-1+((y-1)*width))]*filterY[0]+inData[(x+((y-1)*width))]*filterY[1]
+inData[(x+1+((y-1)*width))]*filterY[2]+inData[(x-1+((y)*width))]*filterY[3]+
inData[(x+((y)*width))]*filterY[4]+inData[(x+1+((y)*width))]*filterY[5]
+inData[(x-1+((y+1)*width))]*filterY[6]+inData[(x+((y+1)*width))]*filterY[7]+
inData[(x+1+((y+1)*width))]*filterY[8];
```

Optimization 3 : Splitting The Unrolled Calculation

After unrolling the loop, we perform a lot of additions and multiplication, we decided to further split them so they can be carried out in parallel

```
float magX1= inData[(x-1+((y-1)*width))]*filterX[0]+inData[(x+((y-1)*width))]*filterX[1]
+inData[(x+1+((y-1)*width))]*filterX[2];

float magX2=inData[(x-1+((y)*width))]*filterX[3]+inData[(x+((y)*width))]*filterX[4]
+inData[(x+1+((y)*width))]*filterX[5];

float magX3=inData[(x-1+((y+1)*width))]*filterX[6]+inData[(x+((y+1)*width))]*filterX[7]
+inData[(x+1+((y+1)*width))]*filterX[8];

magX=magX1+magX2+magX3;

float magY1= inData[(x-1+((y-1)*width))]*filterY[0]+inData[(x+((y-1)*width))]*filterY[1]+
inData[(x+1+((y-1)*width))]*filterY[2];

float magY2=inData[(x-1+((y)*width))]*filterY[3]+inData[(x+((y)*width))]*filterY[4]+
inData[(x+1+((y)*width))]*filterY[5];

float magY3=inData[(x-1+((y+1)*width))]*filterY[6]+inData[(x+((y+1)*width))]*filterY[7]+
inData[(x+1+((y+1)*width))]*filterY[8];

magY=magY1+magY2+magY3;
```

```
outData[x+(y*width)]=sqrt((magX*magX)+(magY*magY));
```

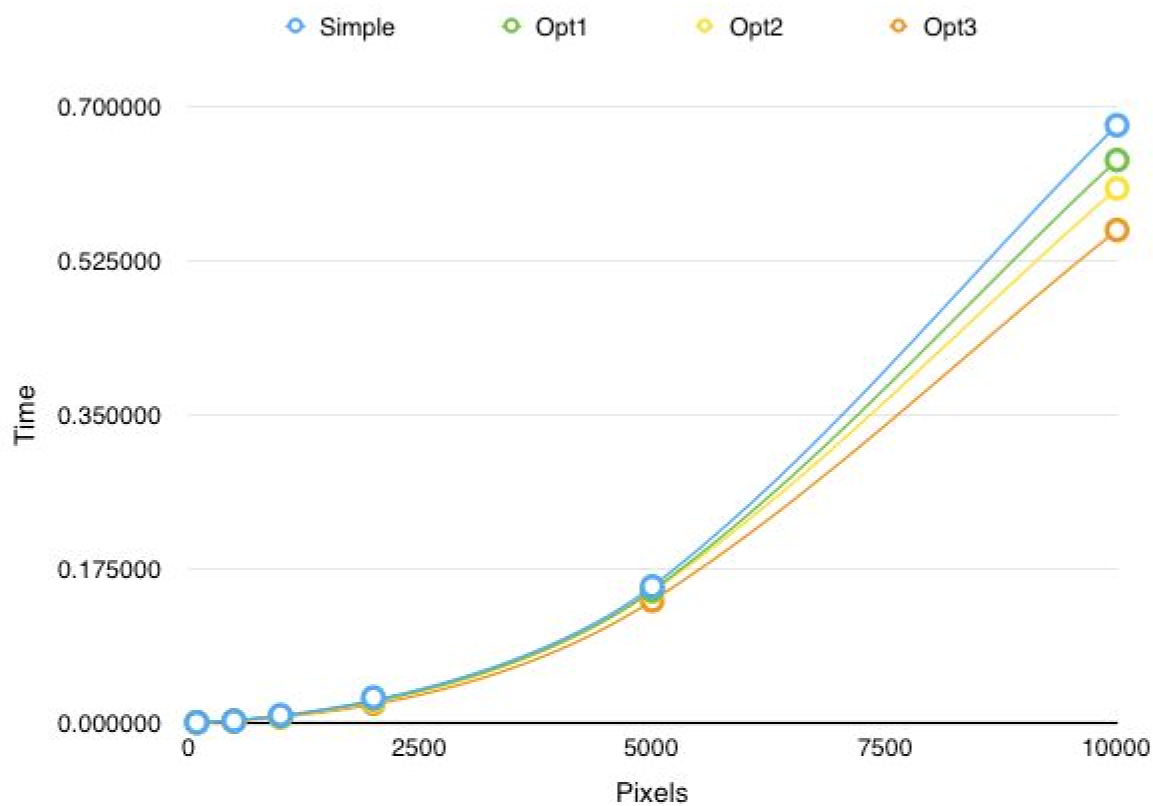
Evaluation

Our setup for evaluation remains the same as last time

Size	Simple (s)	Opt1 (s)	Opt2(s)	Opt3(s)
100x100	0.000070	0.000069	0.000068	0.000062
500x500	0.001792	0.001704	0.001750	0.001469
1000x1000	0.008387	0.007126	0.006555	0.005757
2000x2000	0.028521	0.026837	0.024229	0.021362
5000x5000	0.154532	0.150082	0.148434	0.138290
10000x10000	0.678017	0.638495	0.606223	0.559101

Size	Speed up Opt1	Speed up Opt2	Speedup Opt3
100x100	1.01449275362319	1.02941176470588	1.12903225806452
500x500	1.05164319248826	1.024	1.21987746766508
1000x1000	1.17695761998316	1.27948131197559	1.45683515719993
2000x2000	1.06274918955174	1.17714309298774	1.33512779702275
5000x5000	1.02965045774976	1.04108223183368	1.11744883939547
10000x10000	1.06189868362321	1.11842836711903	1.21269144573163

As we can see, the speed ups are not as good as the previous time, we believe this is due to the heavy computation nature of the sobel operator. There is a lot of arithmetic computation, and the overhead avoided by removing if conditions, is not as big hence the speed ups are less.



<u>Performance Model for optimized Sobel Operator</u>

<u>Roofline Model</u>

For the innermost loop, total number of computations = 103 (we ignore the square root for roofline model)
Since 9 elements of the array are accessed at a given time and 6 of them should already be in cache from the previous iteration, each iteration, only one element is read which also loads the other 2 elements. So number of memory accesses = (1 + 1)*4 bytes = 8 bytes

Arithmetic intensity = (103) / 8 = 12.875

This is clearly heavily compute bound and would perform incredibly better with parallelization.

Basic model

$$T = T_{compute} + T_{memory} + T_{communication} + T_{sqrt}$$

In this case, $T_{communication} = 0$ because of no outside communication. T_{sqrt} is total time spent on calculation square root because we are unsure of how many computes it takes to calculate the same.

$$T_{compute} = T_{addition} + T_{multiplication} + T_{sqrt}$$

$$T_{addition} = (height - 1) * (1 + (width - 1) * 62 * t_{addition}$$

$$T_{multiplication} = (height - 1) * (width - 1) * 40 * t_{multiplication}$$

$$T_{sqrt} = (height - 1) * (width - 1) * t_{sqrt}$$

$$T_{memory} = (height - 1) * (width - 1) * 2 * 4 \text{ bytes} * t_{memory}$$

$$T = T_{compute} + T_{memory} + T_{communication} + T_{sqrt}$$
$$= (height - 1) * (1 + (width - 1) * 62 * t_{addition} + (height - 1) * (width - 1) * 40 * t_{multiplication} + (height - 1) * (width - 1) * t_{sqrt} + (height - 1) * (width - 1) * 8 * t_{memory}$$

Parallelizing General Convolution

We used OpenMp to parallelize our code. We used the optimized version of general convolution where we removed the if condition. We only parallelized the central part of the image that has no if conditions. We wanted to also try and implement it on GPU, but due to shortage of time we could not do it.

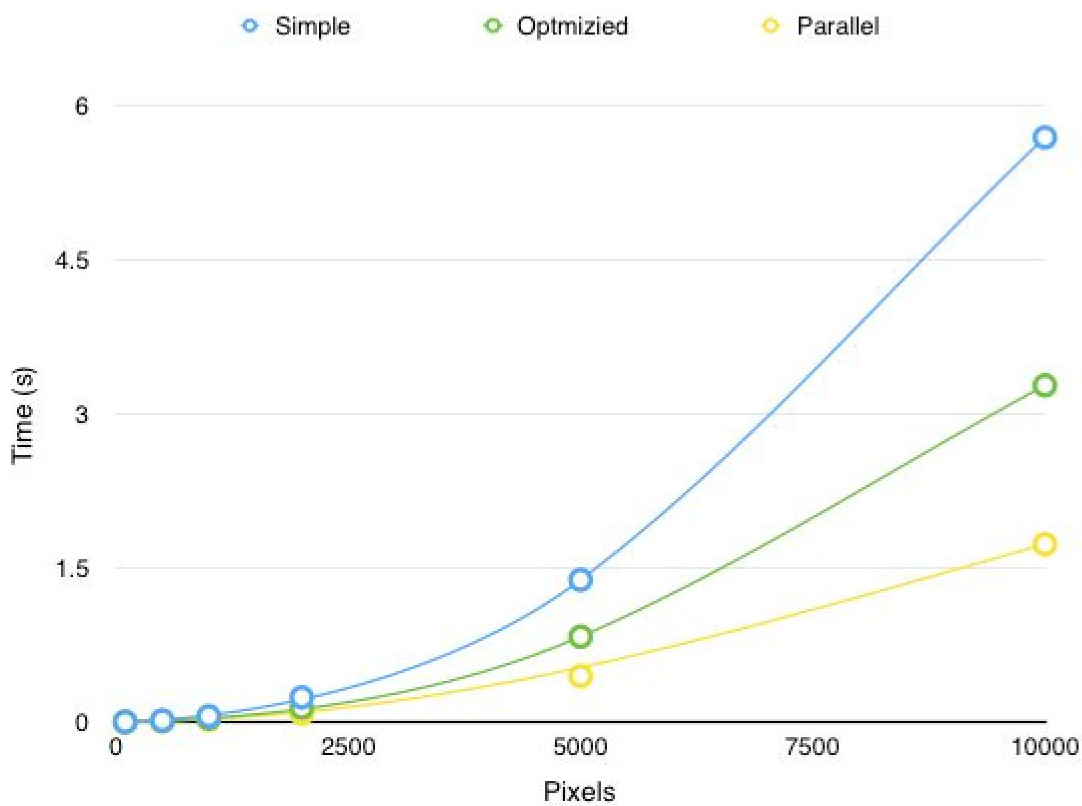
```
#pragma omp parallel for shared(filter,outData,inData)
       for ( int y = STARTOFFSET; y < height-ENDOFFSET; y++ ) {
            ......
            ......
            ......
}
```

Evaluation
Again we use the same machine for evaluation, since its a dual core CPU, we use 2 threads in our openMP implementation. We compare our results against the simple and the optimized implementation

Size	Simple (s)	Parallel (s)	Speedup
100x100	0.000657	0.000581	1.13080895008606
500x500	0.015052	0.005348	2.81451009723261
1000x1000	0.055844	0.020412	2.73584166176759
2000x2000	0.239176	0.078113	3.06192311138991
5000x5000	1.384061	0.447110	3.09557155957147
10000x10000	5.688193	1.731259	3.28558176448469

Size	Opt (s)	Parallel (s)	Speedup
100x100	0.000410	0.000581	0.705679862306368
500x500	0.008887	0.005348	1.66174270755423
1000x1000	0.034983	0.020412	1.71384479717813
2000x2000	0.138092	0.078113	1.76784914162815
5000x5000	0.830404	0.447110	1.85727002303684
10000x10000	3.277270	1.731259	1.89299810138171

As we can see above, compared to the simple approach, we get very good speedups, and even compared to our optimized approach, we are getting close to 2 times the speed up, which is almost linear. For 100x100 we actually have a slower implementation, because the overhead of creating threads is higher than the benefit we get from parallelizing the code.



Lastly, we also decided to run our parallel code on DAS4 since it has more number of cores, so we ran it on DAS4 with number of threads set to 8.

Size	Simple (s)	Parallel (s)	Speedup
100x100	0.000990	0.000284	3.48591549295775
500x500	0.022970	0.005775	3.97748917748918
1000x1000	0.084220	0.007929	10.6217681927103
2000x2000	0.348940	0.028514	12.2374973697131
5000x5000	2.155172	0.178272	12.0892344282894
10000x10000	8.265273	0.696125	11.8732598312085

Size	Opt (s)	Parallel (s)	Speedup
100x100	0.000477	0.000284	1.67957746478873
500x500	0.011626	0.005775	2.01316017316017
1000x1000	0.050806	0.007929	6.40761760625552
2000x2000	0.182768	0.028514	6.40976362488602
5000x5000	1.199755	0.178272	6.72991271764495
10000x10000	4.579893	0.696125	6.57912443885796

As you can see, by increasing the number of threads our speedups also increase, compared to our base for parallelization, the optimized version, we are getting close to 6.6 times the speedup on DAS4