1

The Power of High Performance Computation for Process Acceleration

Lucas Kabela

Abstract—The world is quickly shifting towards augmented computing. As the usage of stronger computation increases, the need to improve grows equally. Massive scale processing requires more computation than can be achieved on the CPU alone. Luckily, new models for parallel process computing are being introduced to meet this need. This projects goal was to test how much optimization can be done with the GPU and understanding how these discoveries impact computing. By passing programs to threads within the GPU, the same process that is done one at a time on the CPU can be streamlined and divided between thousands of threads. This increases the efficiency of carrying out large scale computation. When 2000 frames were sharpened on the GPU and then clocked, the results were 157 seconds while the CPUs results were 4658 seconds. This equates to the GPU being 29.669 times faster than the CPU alone and roughly 1:15:01 quicker than the CPU. While not all test proved the GPU had this high of an advantage, all test consistently demonstrated that the GPU could outperform the CPU. These results culminate to a variety of applications in areas that utilize computer vision and that can subsequently increase efficiency due to boosted performance from GPU utilization and innovation of existing technology.

1 Introduction

Technology has transformed at an unprecedented rate since its proliferation into American households in the early 1900s. The first televisions, introduced in the 1920s, were strictly in black and white. Luckily, our vision is not constrained in an equal manner. Everything seen has complex hues combining to create color that can be perceived by the human eye. In nature, color works through light being manipulated off of the three primary colors- red, blue, and green- through reflection or absorption at certain intensities. In computers however, individual points of an image called pixels have three values that correspond to the primary colors. These values can be decreased or increased in intensity to alter the colors output by the monitor. The values cannot be manipulated infinitely however; the data corresponding to the presence of each color per pixel has a minimum of 0, which is equivalent to no presence of the color, and a maximum threshold of 255, which is complete presence of the color. This means that colors comprising images and camera feeds can be manipulated to display different hues, patterns, and sizes. As photography shifts towards being primarily developed with computers, it is important to understand how the color data present can be altered to create more precise images and feeds. As such alterations increase in complexity, the limitations of modern computing become evident. Extremely complex operations performed on some images can take hours to finish. When such issues emerge, the power, speed, and efficiency of the GPU becomes a viable alternative in image processing. The question then that this study attempts to solve for is if the GPU can consistently outperform the CPU, and if so, then by what margin?

Manuscript received August 14, 2015; revised().

2 Methods

Through the utilization of Microsoft Visual Studio, Open Cv, and Cuda toolkit, 10 algorithms were tested on an image feed streamed from a Logitech Web Camera. A time read function ran at the beginning and end of each function in order to assess the length of the operation. Each individual algorithm was tested a total of

22 times. The test varied in one of two ways: the amount of frames manipulated, or the location of the operation. 11 of the test took place on the GPU where the algorithm was ran in parallel on dedicated threads while the other 11 test ran on the CPU where the process was ran in series. Out of the 10 algorithms, 8 were dedicated to color conversion and two were related to image alterations.

2.1 Color Conversion:

2.1.1 HSL:

Within the scope of the color conversions, the most complex conversion was the transformation of the standard red, blue, green into a color cylinder known as HSL, or Hue, Saturation, Lightness.

This algorithm (See Figure 1) required determining the most present hue, the intensity of this hue, and finally the brightness or darkness of each hue. The code for HSL began with converting the RGB system into a color cylinder to be used for the lightness and saturation. The algorithm runs through each pixel in series or thread in the parallel version. The algorithm starts off converting the RGB values to decimals by dividing by the threshold value of 255. The program then stores the largest and smallest value into a maximum and minimum variable.

Pseudo Code:

For Each Pixel

Convert each value, RGB, to a decimal based on the threshold value 255

Find which value is the maximum and which is the minimum for the pixel

Determine the Lightness by averaging the maximum and minimum value

Determine the saturation by checking if the value from the lightness is greater than .5

If it is,

Then subtract the max from the minimum and divide it by two minus the max and the minimum

Else if the value is less than .5,

Subtract the max and minimum and then divide it by the max plus the min

Determine the Hue by testing for the maximum value

If the maximum value is Blue

Subtract the green value from the red value and divide it by the max minus the min

If the maximum value is Red

Subtract the blue value from the green value and divide it by the max minus the min. Then add four

If the maximum value is Green

Subtract the red value from the blue value and divide it by the max minus the min. Then add two

Then add 360 to the Hue and divide the value by 60

Fig. 1. Pseudo Code for converting RGB to HSL

Then, the lightness is configured by taking the average of the darkest and lightest value utilized in the image. Next, the saturation is calculated by checking if the lightness value is higher than a median value to determine if the image is light or dark. After this is determined, the algorithm subtracts the highest and smallest value and divides it by a variety of possible numbers, depending on the highest and smallest value. Finally, the hue is determined by identifying which value has the highest presence. The other two colors are then subtracted from each other and divided by the maximum and minimum values. Errors are accounted for by adding 2 or 4, depending on which color has the most presence in the image. Finally, in order to convert the values to a cylindrical model, the values are added to 360 and then divided by 60.

2.1.2 Gray Scale

Another color conversion process was the manipulation of a 3 colored pixel to gray scale, or a one dimension color scale. This process involved averaging the 3 primary colors present in every pixel. Once these values were averaged, they could then be stored into one data point for the output to convert the image into one gray scale.

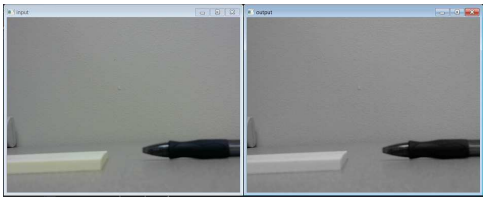

Fig. 2. Image of converting RGB to Gray Scale

2.1.3 Colored Filters

The other 6 color conversions all dealt with manipulating the primary colors of red, blue, and green to create image filters. By deliberately setting the value of a particular color to zero, the program mimicked the absorption of that color, and as a result displayed an output lacking the specified color. To create a red filter for example, all of the colors besides red were removed from the image so only red could be projected in different intensities. Likewise, to

make an additive color filter such as yellow, only the primary colors blue and green were left in the image so that their combined data would result in a yellow tint to the image. The entire color wheel combinations are detailed below with the color being displayed when the proper combination is reflected (See Figure 3)

2.2 Image Altering:

The alteration processes were the most complex, and as a result they required the most computational power. The pseudo code between the two algorithms were somewhat similar, but the primary difference between them was instead of having to check where the surrounding pixels were, the smoothing algorithm simply averaged all the pixels surrounding the origin

2.2.1 Smoothing

Smoothing utilized data points surrounding a particular pixel or thread that served as the origin. In the form of the algorithm tested, a 3 X 3 grid around an origin had its data averaged so that the average of the 9 pixels colors was stored in the data of the origin pixel. In turn, this created a blurring effect. The larger the area averaged around the pixel was, the more blurred the image would become. A count was also kept for each successful average to ensure that the program did not divide by a number larger than the amount of data added. This count was then used to divide the sum of the data points around the origin to produce the average. In theory, the count should have always been equal to the area of the grid, or 9 in the case of this algorithm, but due to some pixels being on edges, this was not true. As a result, the program required more complex logic, such as *if* loops that ensured the pixel that the algorithm was attempting to average was not outside of the image.

2.2.2 Sharpening

The sharpening function checked the values of the surrounding pixels relative to an origin. Instead of averaging all of the pixels, horizontal and vertical edges were calculated separately by adding either the value of the pixels to the right of the origin for the vertical edges, or the data below the origin pixel for the horizontal edges. All data added or subtracted was divided by the area checked, which in the case of this algorithm was 9. In an analogous manner, the pixels above the origin point and to the left of it were subtracted from the total edges. The pixels on the same row or column as the origin were neither added nor subtracted. If an edge was present, the color value of the image would suddenly shift, resulting in a large difference between the pixels above the edge and below the edge. Ultimately, the horizontal edges were added to the vertical edges and the sum was transmitted to the outputs data. See Figure 4 for more details on the function.

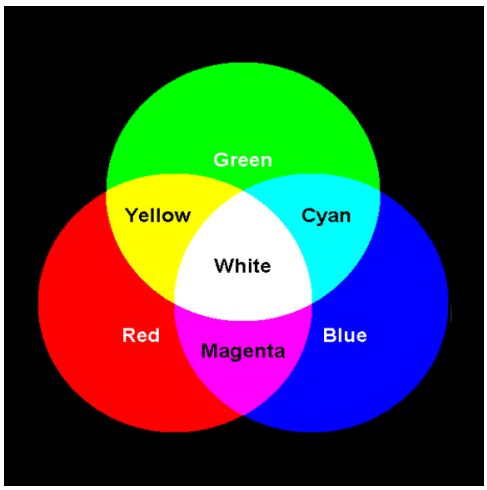

Fig. 3. Primary and Additive Colors displayed

Pseudo Code:

For Each Pixel

Check the row and column of the current pixel and save it as the origin

Go by each 3 X 3 grid around the origin

For the horizontal edges

If the temporary point is above the origin, divide by 9 and subtract from the horizontal edge

If the temporary point is equal to the origin, do nothing to the data for the horizontal edge

If the temporary point is below to the origin, divide by 9 and add to the horizontal edge

For the vertical edges

If the temporary point is to the right of the origin, divide by 9 and subtract from the vertical edge

If the temporary point is equal to the origin, do nothing to the data for the vertical edge

If the temporary point is to the left to the origin, divide by 9 and add to the vertical edge

Combine the vertical and horizontal edge together

Set the averaged sharpened data points as the data output.

Fig. 4. Pseudo Code for Sharpening an image

2.3 Differences in Parallel and Series Versions of the Algorithm:

The greatest difference in the GPU and CPU version of each algorithm was the need to go in series for the CPU compared to the ability to streamline parallel threads on the GPU. While the CPU enacted the function on each pixel, in a series format, the GPU passed each pixel to its own thread. The streamlining was not as simple as a transfer to the GPU however. The parallel threading on the GPU is only available in one dimensional storage, so the color data had to be converted over into an indexed series. This was achieved by the process of using the following equation:

$$index = threadIdx.x + (blockDim.x * blockIdx.x) \quad (1)$$

This equation added the current number of the thread which represented the pixels x position to the total number of pixels in a row times the number of rows already passed, or the current y coordinate in other terms. Once this index was discovered, the index needed to be multiplied by 3 to reach the corresponding color values for the pixel since the original 3 dimensional vector was converted to one dimensional indexes on the GPU. The data of the colors are stored in series on the GPU, so in order to reach the red, green, or blue data, simply multiple 3 for blue then add 1 for green or 2 for red. To access the data input for the threads green value, the index would read: input.data [3 *index + 1]. This indexing, though slightly longer than the CPU variant, granted the GPU clarity and efficiency that the CPU simply lacked.

3 Results

When ran on 4 gigabytes of RAM, 500 gigabytes of memory and a NVIDIA GeForce GT 620 graphics card, the results of the algorithms proved that the GPU could consistently outperform the CPU. The margin by which the GPU outperformed the CPU was dependent on the complexity of the algorithm, but regardless of the complexity or simplicity, the GPU held the advantage for all 10 algorithms. Each algorithms results are provided in the following subsections in more advanced detail.

3.1 Results of converting RGB to HSL:

Due to the complexity of converting the standard primary colors in to a color cylinder, the algorithm for converting RGB to HSL was the most complex of the color conversions. The multitude of *if* and *for* loops created one of the more logic dense functions tested. The results of this somewhat complex algorithm demonstrate that when converting from RGB to HSL, the GPU is up to 14.565 times faster.

Frames	GPUs Time (s)	CPUs Time (s)	Advantage(GPU / CP
10	2	7	3.500x
50	3	26	8.667x
100	5	51	10.200x
150	7	77 (1:17)	11.000x
200	8	109 (1:49)	13.625x
250	10	135 (2:15)	13.500x
300	12	158 (2:38)	13.166x
500	18	252 (4:12)	14.000x
1000	35	505 (8:25)	14.429x
1500	52	763 (12:43)	14.673x
2000	69 (1:09)	1005 (16:45)	14.565x

TABLE 1
Results of converting to HSL

3.2 Results of converting RGB to Gray Scale:

Due to the need to average multiple data points together, the gray scale conversion was the second most complex algorithm out of the color conversions, and this fact was demonstrated by the results of the test. While the averaging was not complex itself, the need to average such large sums, convert their data, and transfer them into one point certainly took a toll on the CPUs performance. The results prove that with a medium complexity process such as converting to gray scale, the GPU can run up to 6.745 times faster

3.3 Results of creating a Cyan filter:

The conversion to cyan was one of the simplest processes to achieve. This was the case since only one value, red, was removed from the output, resulting in two of the primary colors unchanged, and a simple process for the CPU and GPU. The data provides evidence that the GPU still outperformed the CPU by 1.841 times while processing a simple algorithm.

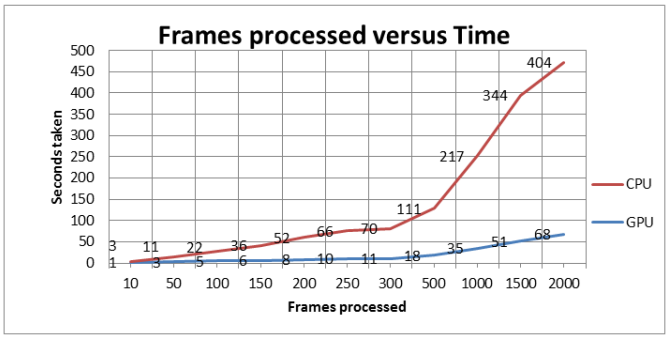

Fig. 5. Results of converting to gray scale

Frames	GPUs Time (s)	CPUs Time (s)	Advantage
10	2	3	1.500x
50	3	4	1.333x
100	5	7	1.400x
150	6	11	1.833x
200	8	14	1.750x
250	10	17	1.700x
300	12	20	1.667x
500	18	32	1.778x
1000	35	63 (1:03)	1.800x
1500	52	95 (1:35)	1.826x
2000	69 (1:09)	127 (2:07)	1.841x

TABLE 2
Results of creating a Cyan Filter

3.4 Results of creating a Yellow filter:

Similar to the cyan process, creating the yellow filter required removing only blue. Consequently, the algorithm was fairly simple and the results for the yellow filter were near identical to those of the cyan filter. The results project the fact that the GPU was up to 1.841 times quicker than the CPU in simple processes such as creating a yellow filter for a video feed.

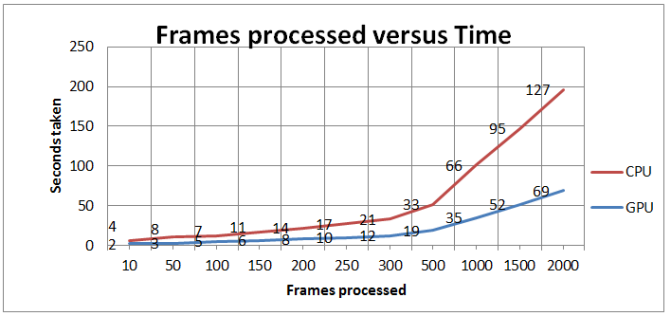

Fig. 6. Results of converting to Yellow Filter

3.5 Results of creating a Magenta filter:

Analogous to the cyan and yellow filtration, the magenta filtration required creating the additive color by removing only green from the image. However, this algorithm resulted in the smallest advantage margin out of the additive color filters. This simple process resulted in clocking the GPU 1.826 times faster in creating a magenta filter than the CPU could.

Frames	GPUs Time (s)	CPUs Time (s)	Advantage
10	2	3	1.500x
50	3	5	1.667x
100	5	8	1.600x
150	6	10	1.429x
200	8	14	1.750x
250	10	16	1.600x
300	12	20	1.818x
500	18	33	1.833x
1000	35	64 (1:04)	1.829x
1500	52	95 (1:35)	1.826x
2000	69 (1:09)	126 (2:06)	1.826x

TABLE 3
Results of creating a Magenta Filter

3.6 Results of creating a Red filter:

The process to create red is a somewhat simple process on the difficulty gradient. This is due to the factors that multiple values must be removed to simulate absorption of color. The only color that must be left is red in the image. As a result, the data shows that the when converting an image to red, the GPU was 3.269 times quicker than the CPU.

Frames	GPUs Time (s)	CPUs Time (s)	Advantage
10	2	3	1.500x
50	3	7	2.333x
100	4	12	3.000x
150	7	18	2.571x
200	8	23	2.875x
250	10	29	2.900x
300	12	34	2.833x
500	18	57	3.167x
1000	35	111 (1:51)	3.171x
1500	52	170 (2:50)	3.269x
2000	69 (1:09)	221 (3:41)	3.203x

TABLE 4
Results of creating a Red Filter

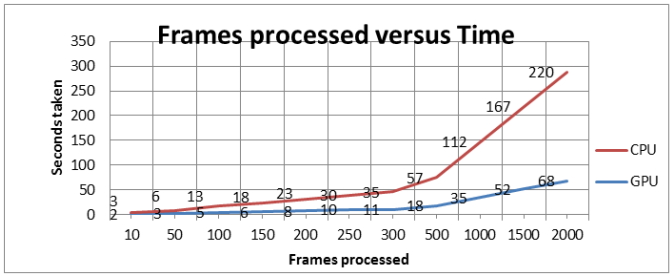

Fig. 7. Results of converting to a Green Filter

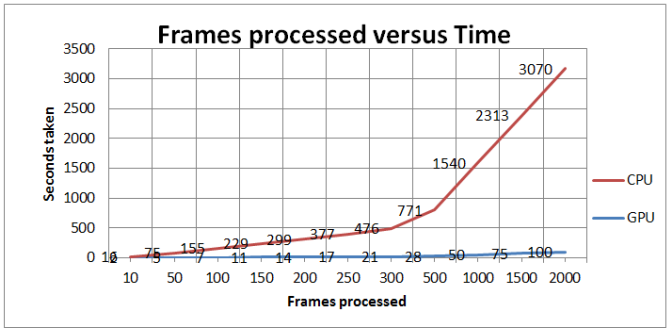

Fig. 8. Results of Smoothing

3.7 Results of creating a Green filter:

Just like the process of changing an image to display in red hues, the process to display in green hues requires the removal of two colors. Greens results were near identical to red in terms of advantage, with green lagging behind the slightest bit. This somewhat simple functions results highlight that the GPU can perform up to 3.235 times faster when creating a green filter on the image than the CPU can.

3.8 Results of creating a Blue filter:

Similar to the other two algorithms that displayed one of the primary colors, the process of changing to blue took out the other two primary colors. However, blue yielded the greatest advantage margin when compared to the other primary color filters. When creating a blue filter, the results prove that the GPU is 3.278 times faster.

Frames	GPUs Time (s)	CPUs Time (s)	Advantage
10	2	3	1.500x
50	3	7	2.333x
100	5	13	2.600x
150	7	19	2.714x
200	9	26	2.889x
250	10	30	3.000x
300	12	35	2.917x
500	18	59	3.278x
1000	36	116 (1:56)	3.222x
1500	52	168 (2:48)	3.278x
2000	69 (1:09)	226 (3:46)	3.275x

TABLE 5
Results of creating a Blue Filter

3.9 Results of Smoothing the image:

The smoothing process resulted in one of the greatest GPU advantages due to its complexity. Since the colors had to be averaged in a surrounding grid, this function was one of the most complex algorithms tested. The results certainly showed as the GPU barely slowed down compared to its previous runs while the CPU took nearly an hour to complete the task. Ultimately, the GPU was up to 29.367 times quicker handling on of the most complex processes.

3.10 Results of Sharpening the image:

The sharpening function required more nested loops, logic checks, and different kinds of addition, subtraction, and division to reach the total of the edge than any other algorithm. As a result, it is the most complex function tested and registered the longest times on the GPU and CPU as a result. This factor did not affect the advantage of the GPU as the GPU was up to 36.765 times better at sharpening edges than the CPU was.

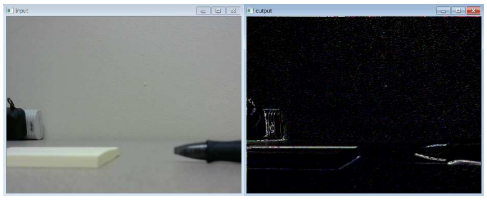

Fig. 9. Image of sharpening

4 Discussion

The data within the results provided discussion for several intriguing questions related to the studies primary question and additional observations.

Frames	GPUs Time (s)	CPUs Time (s)	Advantage
10	2	23	11.500x
50	5	115 (1:55)	23.000x
100	7	222 (3:42)	31.714x
150	10	334 (5:34)	33.400x
200	14	444 (7:24)	31.714x
250	17	625 (10:25)	36.765x
300	24	673 (11:13)	28.041x
500	41	1153 (19:13)	28.122x
1000	79 (1:19)	2313 (41:27)	29.278x
1500	118 (1:52)	3536 (58:56)	29.966x
2000	157 (2:37)	4658 (1:17:38)	29.669x

TABLE 6
Results of sharpening an image

4.1 Adressing why the GPU is consistently faster

The primary question is, why did the GPU consistently outperform the CPU? When examining the mechanics of the GPU, it makes sense as to why the GPU will always be quicker than the CPU. While the CPU is one processing unit working with all of the variables and functions in one stack, the GPU is composed of myriad threads that allow for unprecedented efficiency. The CPU must work in serial methodology due to the stack, meaning that it must handle each process on each pixel one at a time. The GPU on the other hand is able to transfer one pixel to each thread. When this occurs, the GPU can run the algorithms on each pixel in parallel, or at the same time, and return the result to the output as each pixel is finished. The CPU, on the contrary, must sort through each pixel one at a time, resulting in an exponentially increasing amount of time to complete a task as more frames are added. The GPUs results showed the opposite of this trend as the graphs are somewhat linear due to it returning each pixel that it finishes without waiting on the other pixels.

4.2 Answering why the advantage margin varied between test

When comparing the results of each algorithm, another question emerges over as to why the GPU did not consistently outperform the CPU by the same margin. This answer ties into the purpose of this study as to how much more efficient the GPU is than the CPU. The answer to this scenario is the complexity of the algorithms. When computing the simple process of changing one color value to zero, the GPU and the CPU both performed at nearly the same speeds due to how simple the process was. When the process was more complex and required increased logic, the CPUs shortcomings were increasingly apparent while the GPU continued to perform marginally better. The difference in the simple and complex functions times was a due to the fact that more complex algorithms had to be handled one at a time for each pixel on the CPU while the GPU ran the processes on each pixel simultaneously, resulting in more rapid output on the GPU. When examining the advantage in relations to the pseudo code, it becomes apparent that the more processes the function had to carry out, the larger the advantage was. This is why creating the additive colors of Cyan, Magenta, and Yellow resulted in quicker times for the CPU that creating the Red, Green, and Blue ones did. The process of creating the additive colors required removing only one color while creating the primary colors required the removal of two colors. Thus, the algorithms simplicity resulted in a faster process and a smaller margin between the CPU and the GPU.

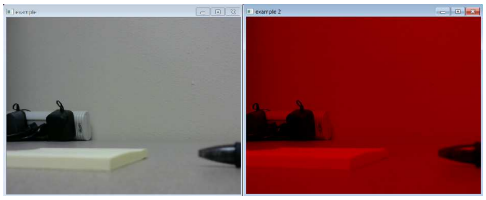

Fig. 10. Image of creating a red filter

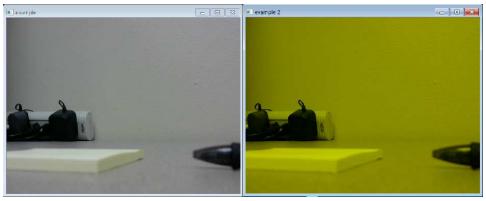

Fig. 11. Image of creating a yellow filter

4.3 Investigating why some color filters were faster than others

On the subject of the color conversion, it should be noted that some color conversions were actually quicker than the others, with the conversion to blue registering the fastest advantage out of the primary color conversions. This can be attributed to the order in which the data points are held. In the color array, blue is stored first, green is stored second and red is placed third. When altering their values and making calls to the global memory, it it much more easy to factor out numbers that are closer to the edges of the array storage points. This is the case because when calls are made, surrounding data is also brought into the temporary memory. When the call was made to change the first data point in the color array, green, blue and red were both brought over as well. Since red was already brought over, another call to memory was not required and the conversion could be done without needing to make additional calls. When converting to a filter such as green that is in the middle, the call is made to the storage of blue, and green is brought over with it, but red is not. As a result, another call must be made and the process is slowed down. Thus, some color conversions were quicker on the basis of making less calls to the global memory.

4.4 Thoughts on further GPU augmentation and test

What should be noted in every procedure is that the GPU was not fully optimized. As discussed in the methodology, each pixel was assigned to one thread, and its color values were indexed over the course of 3 separate data points. However, this process could be even faster if multithreading were implemented and shared memory were to be utilized over global memory. In their current stages, the algorithms have to make calls to the global memory to get the data points around them, such as what occurs in the sharpening or smoothing algorithms. If the GPU were to be even further optimized, shared memory could be utilized for threads that were joined together in a block, and thus less calls to the global memory would be required and an even more streamlined process could be achieved. This methodology was not attempted in this study due to the factor that the primary processes did not rely on complex image alterations. More research into the area of complex image alteration and its enhancing through GPU shared memory and multithreading could yield even more beneficial results to the image processing industry.

4.5 Closing Thoughts

While the modern world utilizes image processing on a larger scale due to technologies assimilation in culture, image processings current limitations are becoming evident. Such limitations can be overcome by utilizing the GPU of computers in order to provide for the expedition of image processing and improve the efficiency of current image processing technology. With further research, the effects of increased speed in image processing can aid the multitude of industries, home owners, and firms that are relying on image processing and cameras to keep them secure and safe. The implications can also reach into drones, autonomous machinery, and other futuristic industries that will rely on cameras.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Alireza Tavakkoli for all of his aid in developing this project and for providing direction throughout the course of this project.