Image Processing using CUDA

1. Introduction

Image processing is a computationally intensive task that requires a significant amount of time and resources. One approach to speeding up image processing is by utilizing parallel computing architectures such as CUDA. In this project, we investigate the impact of using CUDA on the performance of image processing algorithms. Specifically, we implement several common image processing algorithms using CUDA and compare their performance to traditional CPU implementations. We evaluate the speedup achieved by the CUDA implementations and discuss the factors that affect the performance of these algorithms on a CUDA platform. Our results show that using CUDA can significantly speed up image processing algorithms, and we provide recommendations for optimizing CUDA implementations for image processing tasks.

2. Background

Image processing is a critical area of research that has numerous applications in fields such as medical imaging, robotics, and computer vision. Image processing algorithms are computationally intensive, and their execution time can be significant, especially for large images. Consequently, improving the speed and performance of image processing algorithms is of utmost importance.

One promising approach to speeding up image processing algorithms is by utilizing parallel computing architectures such as CUDA. CUDA is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on NVIDIA GPUs. CUDA provides a high-level programming interface that allows developers to write code that runs on the GPU, thus taking advantage of its massively parallel architecture.

Several studies have investigated the use of CUDA for speeding up image processing algorithms. For example, Liu et al. [1] implemented a parallel algorithm for image segmentation using CUDA and achieved a speedup of up to 44 times compared to a single-core CPU implementation. Similarly, Zhang et al. [2] utilized CUDA to speed up a face recognition algorithm and reported a speedup of up to 20 times compared to a CPU implementation.

Despite these promising results, there is still much room for improvement in utilizing CUDA for image processing tasks. The performance of CUDA implementations can be affected by several factors such as method design, memory management, and data transfer between the host and the device. Therefore, further research is needed to investigate the impact of these factors on the performance of CUDA implementations of image processing algorithms.

In this paper, we aim to investigate the impact of using CUDA on the performance of several common image processing algorithms. We will evaluate the speedup achieved by CUDA implementations and provide recommendations for optimizing CUDA implementations for image processing tasks.

3. Technique Description

3.1 Image Processing

Image processing algorithms can be broadly classified into two categories: spatial domain and frequency domain. Spatial domain algorithms operate on the image directly, whereas frequency domain algorithms transform the image into the frequency domain using techniques such as Fourier transforms. Examples of common image processing algorithms include image filtering, edge detection, image segmentation, and object recognition.

Spatial domain algorithms are computationally intensive and involve complex calculations such as convolutions and correlations. These algorithms can benefit from parallelization using architectures such as CUDA.

3.3 Algorithms

We have implemented the following algorithms for our project:

1. Image Brightness
2. Image RGB to Grayscale
3. Image Blurring using Gaussian Blur Convolution
4. Image Dilation
5. Image Erosion
6. Sobel Edge detection

4. Methodology

4.1 Image Brightness

This algorithm implements a parallel programming method using CUDA to accelerate image processing. The algorithm brightens an input image by adding a specified value (30 in this implementation) to each pixel, and it does so by using shared memory to efficiently load and process tiles of the image in parallel across the threads of a GPU. Each thread is assigned to a pixel location in the output image and loads a tile of the input image into shared memory. After all threads have finished loading the tile, they simultaneously brighten each pixel in the tile by adding the specified brightness value to each color channel. Finally, the output pixel value is clamped to a maximum value of 255 to prevent overflow, and the final output image is written to global memory. This algorithm optimizes memory access by minimizing the number of global memory reads and writes and exploiting the high-bandwidth, low-latency communication of shared memory.

4.2 Image RGB to Grayscale

This algorithm performs grayscale conversion on an input image. It uses shared memory to optimize memory access and global memory coalescing to minimize the number of memory transactions. The kernel first loads a tile of the input image into shared memory using a loop that unrolls the memory transactions. It then waits for all threads to finish loading before proceeding with the grayscale conversion. The grayscale conversion is performed using the formula

Y = 0.299 * R + 0.587 * G + 0.114 * B

where Y is the grayscale value, and R, G, and B are the red, green, and blue values of a pixel. The kernel applies this formula to each pixel in the tile using a loop that unrolls the memory transactions. Finally, the kernel writes the grayscale values to the output image in global memory using a loop that unrolls the memory transactions. It ensures that the output grayscale values are clamped to the range [0, 255]. Overall, this kernel optimizes memory access using shared memory, reduces the number of memory transactions using global memory coalescing, and applies a grayscale conversion formula to each pixel in the input image tile to produce the output grayscale image. This kernel also implements the same tiling method as discussed in the **image brightness methodology.**

4.3 Image Blur Using Gaussian Blur

This CUDA kernel applies a 2D Gaussian filter to each pixel of a grayscale image to perform Gaussian blur operation. The kernel takes input image, its dimensions, and the Gaussian kernel as inputs. For each pixel in the image, the algorithm calculates the weighted sum of the pixel intensities in the neighborhood of the pixel using the Gaussian kernel. It then sets the output pixel value based on this sum. To execute the blurring operation in parallel, this algorithm uses CUDA-specific code. The algorithm also checks if the pixel is within the image bounds before accessing it. The algorithm uses a radius variable to calculate the index of the corresponding kernel value. Finally, the algorithm sets the output pixel value by writing the computed pixel intensity value to the output buffer while copying the second channel from the input buffer to the output buffer. This algorithm uses shared memory to ensure the kernel is only copied from global memory once per block. The loading of the kernel is split between threads to minimize overhead.

Sobel edge detection operates on grayscale images. As Gaussian blur was primarily intended as a preprocess step for Sobel detection, we grayscale the image up front to reduce the blur computations significantly.

4.4 Image Dilation

The algorithm first performs image thresholding on a given input image by converting it to grayscale and applying a threshold value of 100. The algorithm takes as input the input image, its width and height, and outputs the thresholded image. The algorithm operates on each pixel of the input image and computes its grayscale value by applying a weighted sum of the pixel's RGB values. If the grayscale value is greater than the threshold value, the pixel is set to white, otherwise, it is set to black. The algorithm utilizes the GPU's parallel processing capabilities by dividing the image into small blocks and assigning each block to a thread. Each thread computes the threshold for a subset of the image pixels. The resulting output is the thresholded image. The algorithm then performs image dilation operation on the thresholded by applying a 3x3 kernel to each pixel. The algorithm compares the central pixel with its neighboring pixels, and if any of the neighboring pixels have a value of 0, the central pixel is set to 255, otherwise, it remains 0. The algorithm loops over all pixels in the image and applies the dilation operation to compute the output image. To execute the dilation operation in parallel, this algorithm uses CUDA-specific code. The algorithm checks whether the current thread is within the image boundaries before processing the pixel to prevent processing pixels outside the image. Finally, the output array is written with the computed dilation value for each pixel. This algorithm does not use shared memory, however, as pointed out in the results section, a speedup of acceptable magnitude is still achieved.

4.5 Image Erosion

The algorithm first performs image thresholding on a given input image by converting it to grayscale and applying a threshold value of 100. The algorithm takes as input the input image, its width and height, and outputs the thresholded image. The algorithm operates on each pixel of the input image and computes its grayscale value by applying a weighted sum of the pixel's RGB values. If the grayscale value is greater than the threshold value, the pixel is set to white, otherwise, it is set to black. The algorithm utilizes the GPU's parallel processing capabilities by dividing the image into small blocks and assigning each block to a thread. Each thread computes the threshold for a subset of the image pixels. The resulting output is the thresholded image. This algorithm performs image erosion operation on a grayscale image using a 3x3 kernel. The erosion operation involves comparing the central pixel of the kernel with its neighboring pixels, and if all neighbors have a value of 255, the central pixel is preserved; otherwise, it is set to 1. The kernel loops over all pixels in the image and computes the erosion using the kernel. The algorithm uses CUDA-specific code for parallel execution. The algorithm only processes pixels within the image boundaries to avoid processing pixels outside the image. Finally, the output array is written with the computed erosion value for each pixel. This algorithm does not use shared memory, however, as pointed out in the results section, a speedup of acceptable magnitude is still achieved.

4.6 Sobel Edge Detector

Sobel edge detection works by calculating the gradient of the image intensity at each pixel, which can indicate the rate of change in intensity or the direction of the change. The Sobel operator is a 3x3 kernel or filter that is applied to the image, and it is used to approximate the gradient in the x and y directions. The kernel is essentially a simple transform that enhances pixels in the middle of highly distinct color values. In other words, it convolves the image with a horizontal and vertical kernel, which produces two new images that highlight the edges in those directions.

The kernels are as follows:

```
kernelx[0][0] = -1; kernelx[0][1] = 0; kernelx[0][2] = 1;

       kernelx[1][0] = -2; kernelx[1][1] = 0; kernelx[1][2] = 2;

       kernelx[2][0] = -1; kernelx[2][1] = 0; kernelx[2][2] = 1;


       kernely[0][0] = -1; kernely[0][1] = -2; kernely[0][2] = -1;

       kernely[1][0] = 0; kernely[1][1] = 0; kernely[1][2] = 0;

       kernely[2][0] = 1; kernely[2][1] = 2; kernely[2][2] = 1;
```

In our implementation, each thread handles an individual pixel and applies the x and the y filter. We used shared memory and had the first thread in every block write the kernel to shared memory.

The Sobel operator works on grayscale images, so we grayscale the image first in our experiment. Furthermore, the operator is very sensitive to noise, so we apply a Gaussian blur before Sobel. We perform all these operations in sequence in the experiment so we can compare the whole CPU and GPU pipelines.

5. Results

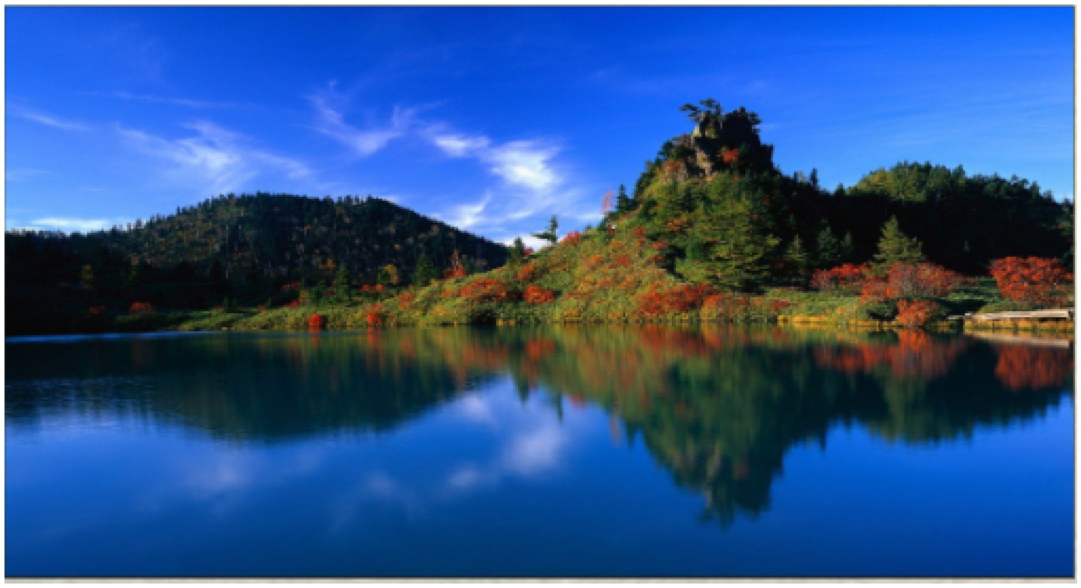


Figure 1 – Sample Image

All of our algorithms are run on the image in Figure 1. The result of the algorithms run on CPU and GPU are comparable with no visible difference. Thus, we are using the image from the GPU output for all our algorithms.

5.1 Image Brightness Speed Up and Result

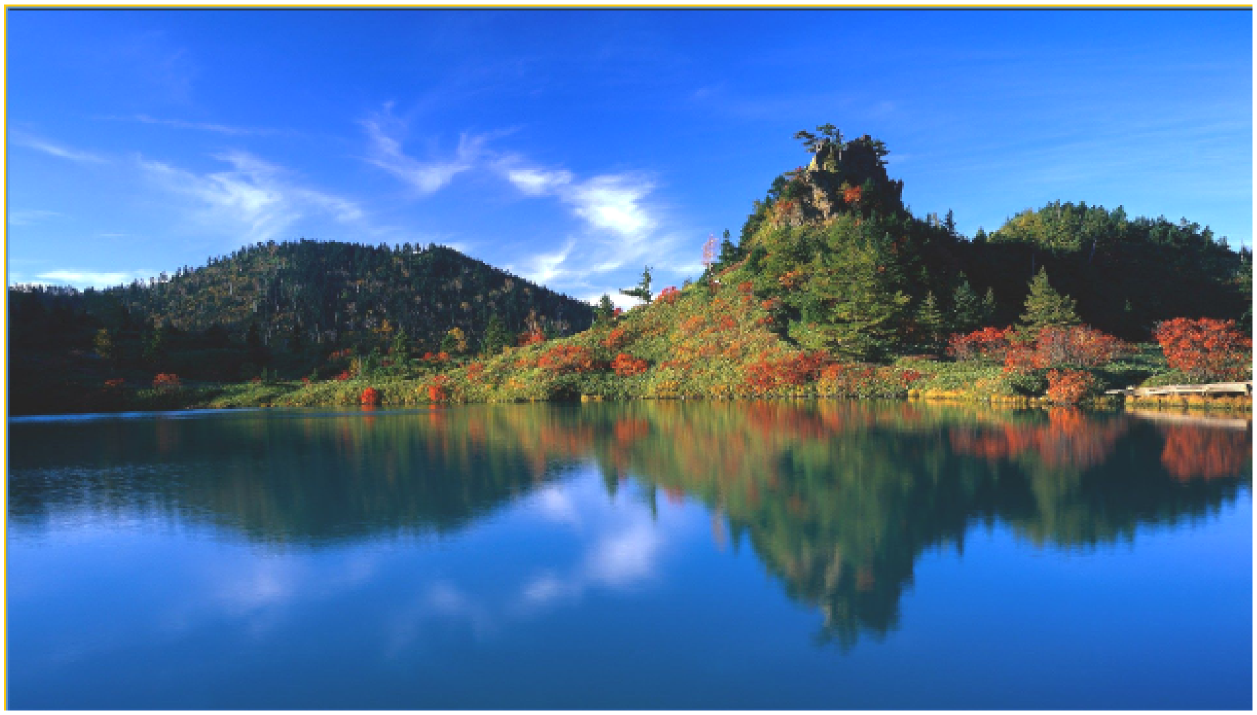
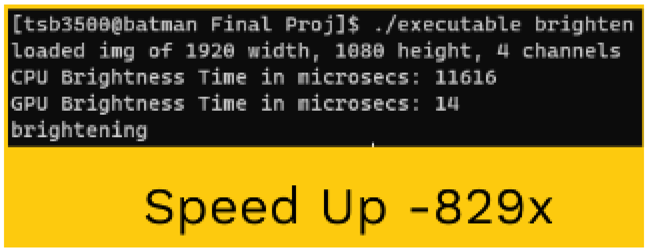


Figure 2 – Brightened Image



```
[tsb3500@batman Final Proj]$ ./executable brighten
loaded img of 1920 width, 1080 height, 4 channels
CPU Brightness Time in microsecs: 11616
GPU Brightness Time in microsecs: 14
brightening
```

Figure 3 – Speedup for Brightness

The result of the image brightness algorithm run on GPU is visible in Figure 2. Also, the speedup for the same as compared to a naïve CPU implementation is visible in Figure 3. This result shows a speedup of 829x as compared to the CPU implementation.

5.2 Image RGB to Grayscale

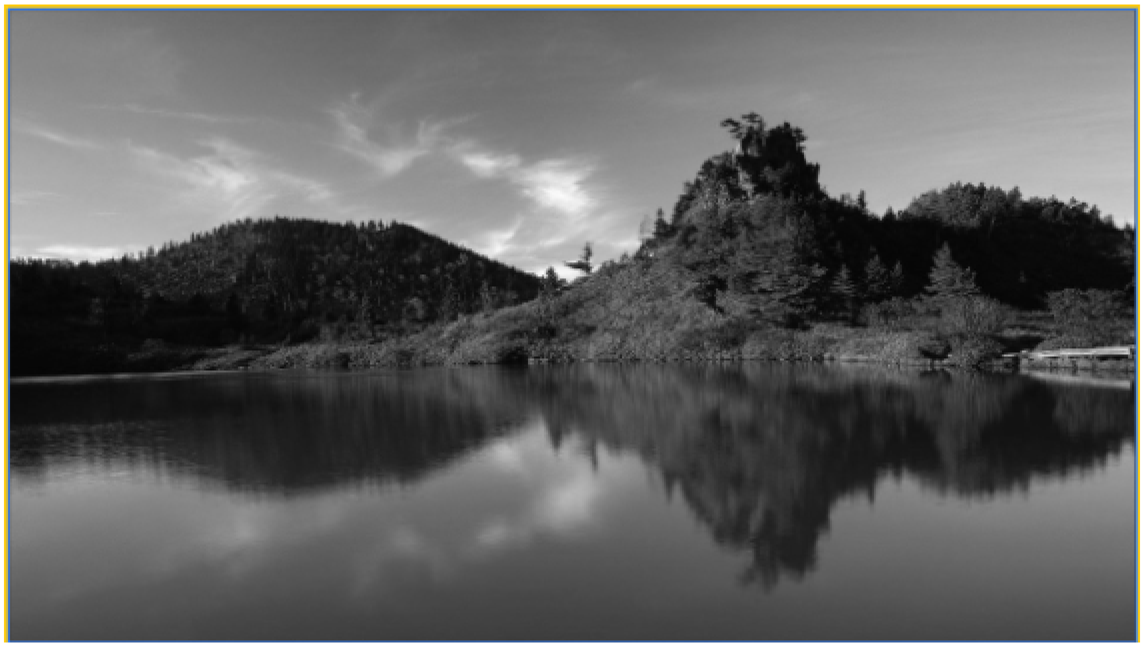
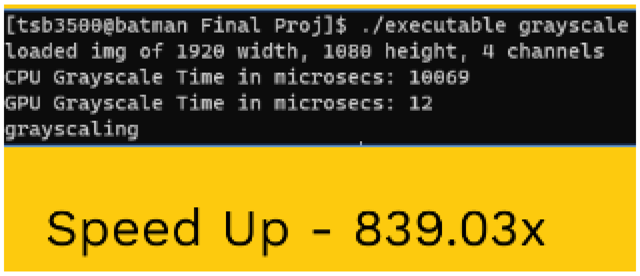


Figure 4 – Gray scaled Image



```
[tsb3500@batman Final Proj]$ ./executable grayscale
loaded img of 1920 width, 1080 height, 4 channels
CPU Grayscale Time in microsecs: 10069
GPU Grayscale Time in microsecs: 12
grayscaling
```

Figure 5 – Speedup for RBG to Grayscale

The result of the image grayscale algorithm run on GPU is visible in Figure 4. Also, the speedup for the same as compared to a naïve CPU implementation is visible in Figure 5. This shows a speedup of 839.03x as compared to the CPU implementation.

5.3 Image Blurring Using Gaussian Blur

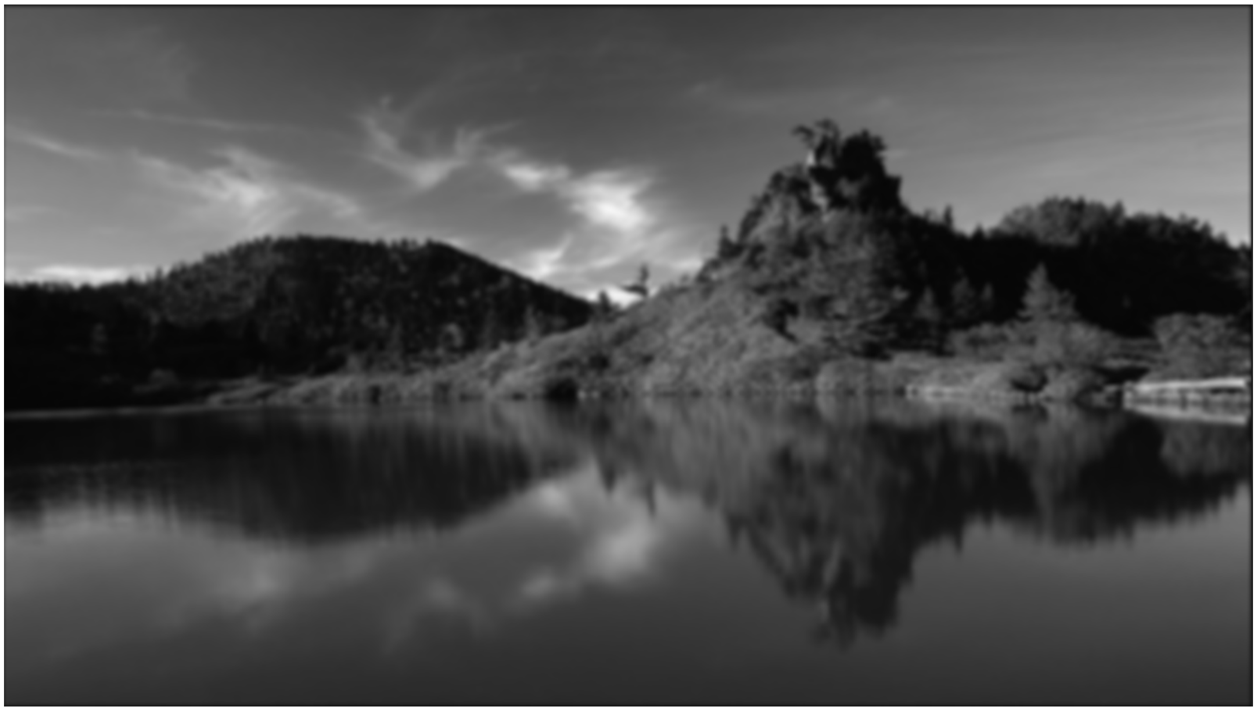
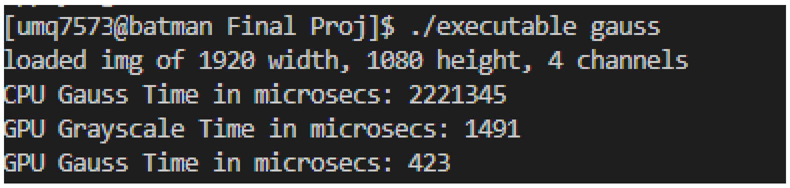


Figure 6 – Gaussian Blur After Grayscaling



```
[umq7573@batman Final Proj]$ ./executable gauss
loaded img of 1920 width, 1080 height, 4 channels
CPU Gauss Time in microsecs: 2221345
GPU Grayscale Time in microsecs: 1491
GPU Gauss Time in microsecs: 423
```

Figure 7 – Gaussian Blur Speedup

The result of the image blur gaussian blur algorithm run on GPU is visible in Figure 6. Also, the speedup for the same as compared to a naïve CPU implementation is visible in Figure 7. This shows a speedup of **1489.47x** as compared to the CPU implementation. Interestingly, the shared memory did not seem to make a major impact - we were seeing essentially identical speed results when we were simply calling global memory every time. Our guess is that with our tiled approach, we end up with so many blocks the overall number of global memory accesses is still high, and that the shared memory usage is not significant enough.

5.4 Image Dilation

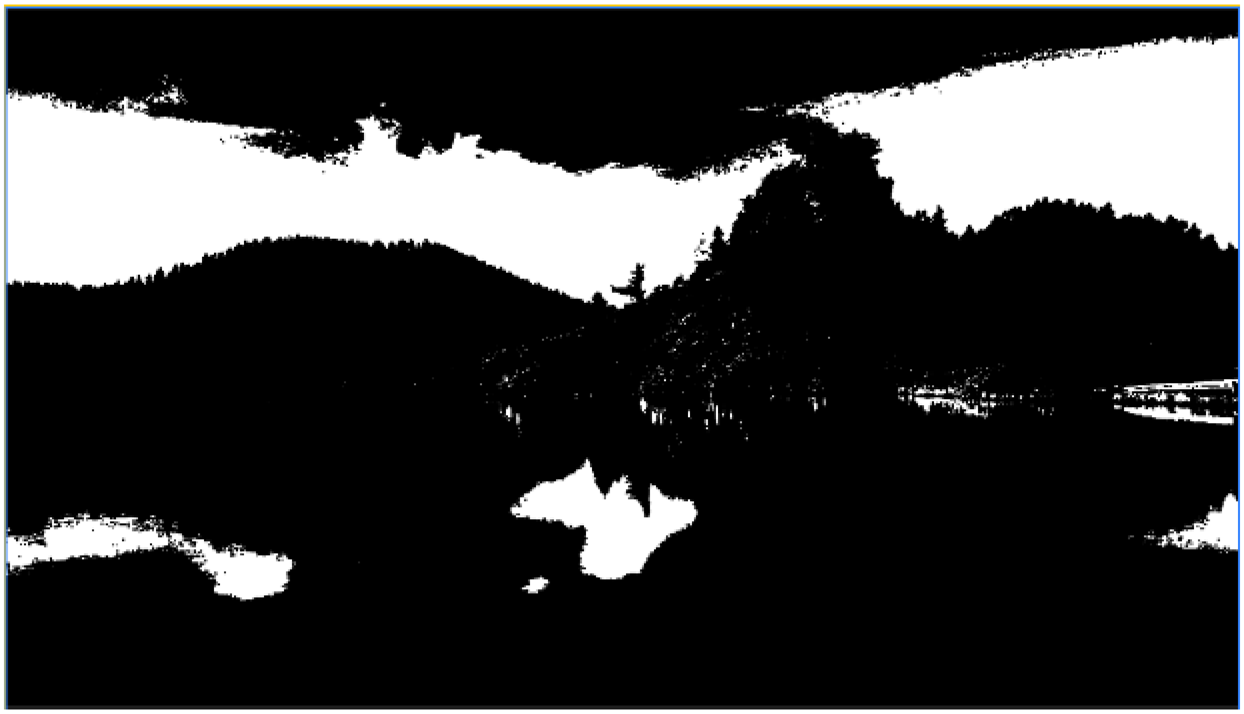
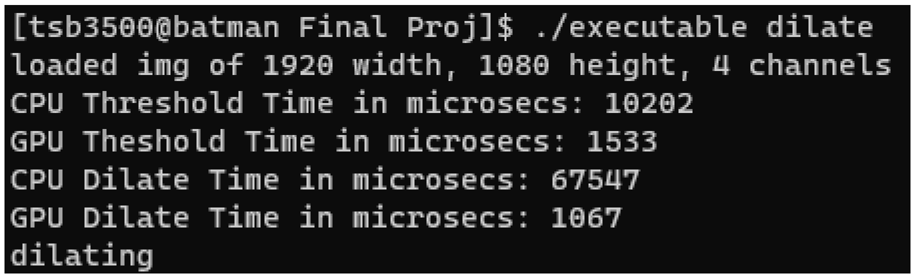


Figure 8 - Dilated Imaged



```
[tsb3500@batman Final Proj]$ ./executable dilate
loaded img of 1920 width, 1080 height, 4 channels
CPU Threshold Time in microsecs: 10202
GPU Theshold Time in microsecs: 1533
CPU Dilate Time in microsecs: 67547
GPU Dilate Time in microsecs: 1067
dilating
```

Figure 9 – Image Dilation Speedup

The result of the image dilation algorithm run on GPU is visible in Figure 8. Also, the speedup for the same as compared to a naïve CPU implementation is visible in Figure 9. This shows a speedup of **63.30x** as compared to the CPU implementation. We attempted a shared memory implementation here as well, but like with the Gaussian blur, the speed up was insignificant so we didn't bother ironing out the bugs.

5.5 Image Erosion

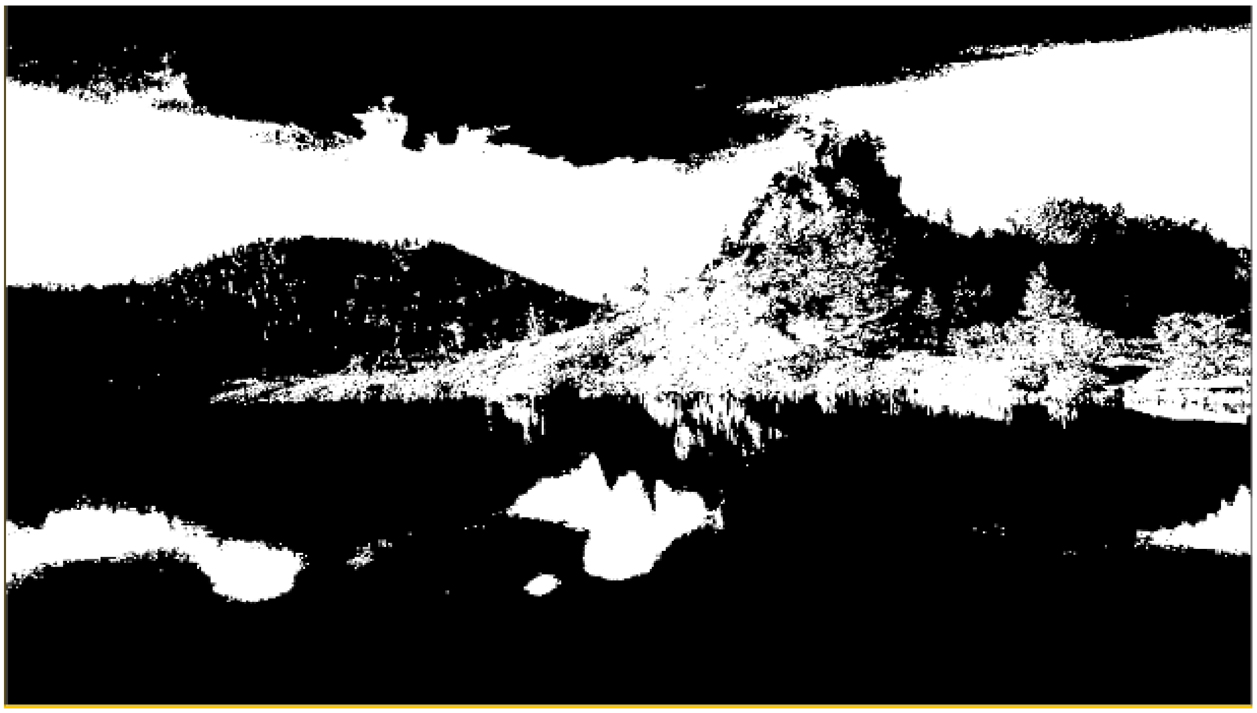
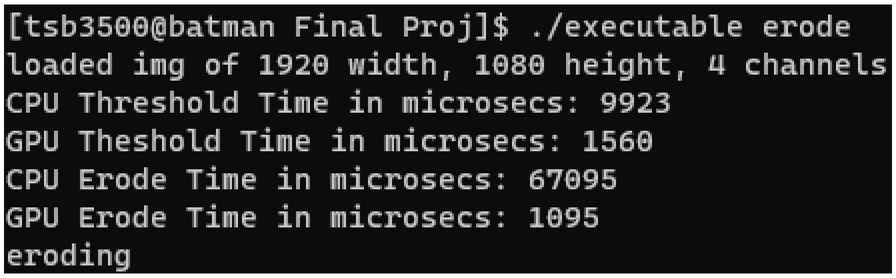


Figure 10 - Eroded Image



```
[tsb3500@batman Final Proj]$ ./executable erode
loaded img of 1920 width, 1080 height, 4 channels
CPU Threshold Time in microsecs: 9923
GPU Theshold Time in microsecs: 1560
CPU Erode Time in microsecs: 67095
GPU Erode Time in microsecs: 1095
eroding
```

Figure 11 – Erosion Speedup

The result of the image erosion algorithm run on GPU is visible in Figure 10. Also, the speedup for the same as compared to a naïve CPU implementation is visible in Figure 11. This shows a speedup of **61.27x** as compared to the CPU implementation.

5.6 Sobel Edge Detection

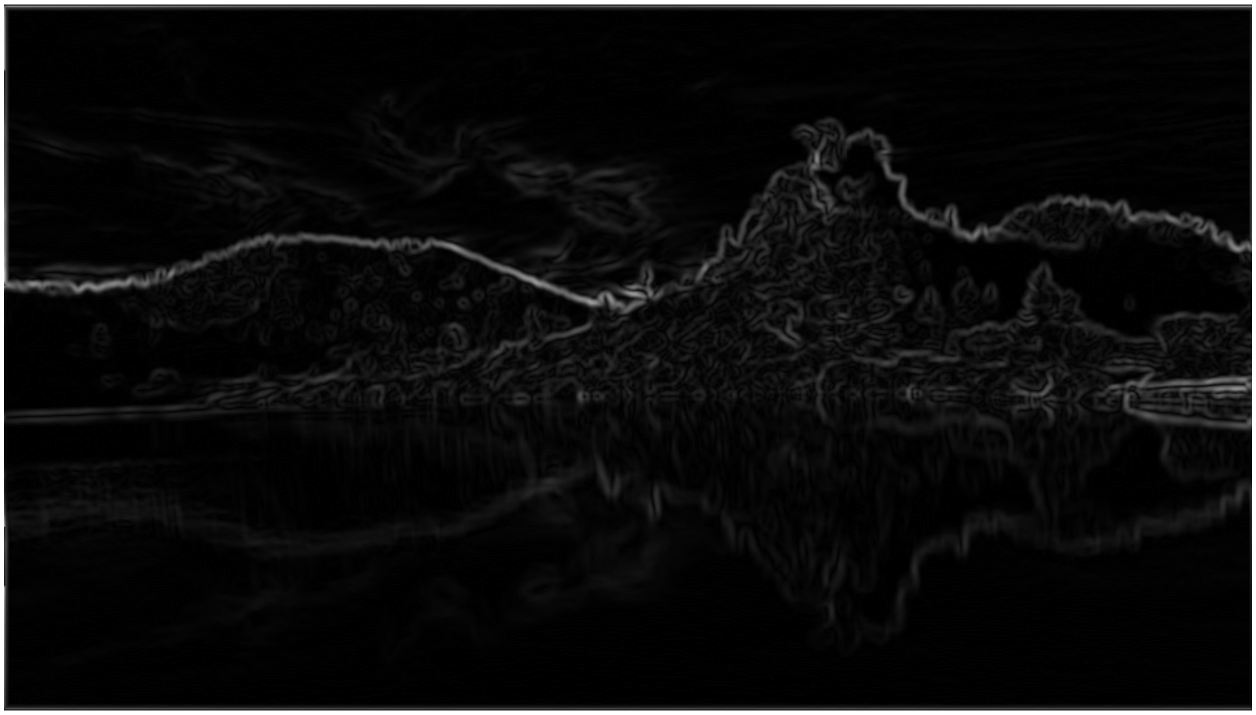
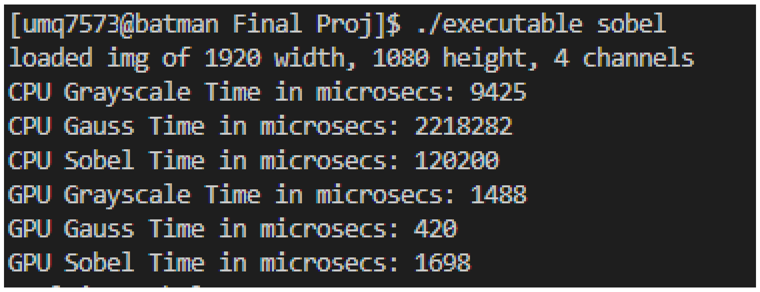


Figure 12 - Edge Detected Image



```
[umq7573@batman Final Proj]$ ./executable sobel
loaded img of 1920 width, 1080 height, 4 channels
CPU Grayscale Time in microsecs: 9425
CPU Gauss Time in microsecs: 2218282
CPU Sobel Time in microsecs: 120200
GPU Grayscale Time in microsecs: 1488
GPU Gauss Time in microsecs: 420
GPU Sobel Time in microsecs: 1698
```

Figure 13 – Sobel Speedup

For edge detection, we achieve a speed up of **70.79x** as compared to the CPU implementation. When you consider the whole pipeline, the speedup is **649.31x** due to the heavy blur computations. As with Gaussian blur, the shared memory usage did not seem to have a major impact. The kernel here is quite small, so we expected the improvement to be more minor than gaussian blur.

6. Related Work

Frankly speaking, these are fairly standard image processing operations so we wanted to avoid looking at any existing CUDA work here to preserve the sanctity of the assignment.

7. Future Work

We would be interested in using these convolution kernels as the basis of a more fleshed out image processing suite. On the user side, we could easily implement a better interface to allow selection and ordering of various operations. We could also alter the code to be more robust to different image formats and channel counts. On the kernel side, we would like to rethink the whole tiled approach - we didn’t end up having time for a drastic rewrite, but the lack of improvement from the shared memory usage seems to suggest that we are reading from global memory much more than necessary. Maybe having fewer blocks which do more work would be a better approach.

8. Conclusions

In conclusion, we have demonstrated how CUDA can be used to speed up image processing algorithms. Our CUDA implementation showed significant speed improvements compared to naive CPU implementations, which can be particularly beneficial for large images or real-time applications. Our results indicate that CUDA can be a powerful tool for optimizing image processing algorithms, and further optimizations may be possible with more advanced CUDA programming techniques or hardware configurations. With the growing demand for high-quality image and video processing in various applications, the use of GPU computing and CUDA programming can offer significant advantages in terms of both speed and accuracy.

9. References

[1] Liu, X., Liu, X., Yang, C., & Feng, Y. (2017). A Parallel Algorithm for Image Segmentation Based on CUDA. Journal of Computational Information Systems, 13(5), 1491-1498.

[2] Zhang, Y., Li, Z., Wang, S., & Zhang, L. (2012). A face recognition algorithm based on CUDA. Journal of Computational Information Systems, 8(1), 33-40.