Preparation of Papers for ece569 Project Report

First A. Author, , Second B. Author, and Third C. Author

* Introduction

Machine vision systems are becoming an important part of technology today. All these systems must interpret images like humans. Although humans can recognize shadows and real objects, many machine vision systems have trouble recognizing these differences.

Self-driving systems, for example, will need to interpret shadows and actual cars to provide an adequate driving experience for the user. A system that fails to recognize shadows could cause issues to the system which can affect the performance and even the safety of the user.

As with most applications, the efficiency and execution time of the shadow removal algorithm is critical to the performance of computer vision system. The use of GPUs (Graphic Processing Units) is critical to increasing performance metrics such as execution time.

GPUs contain many threads that can execute instructions in parallel, therefore decreasing the execution time substantially. Each thread in a GPU executes a single kernel which can utilize optimization techniques such as shared memory, constant memory, and memory coalescing which also increases the performance of the implementation.

There are a total of five sections in this report. In Section II, related works that implement shadow removal algorithms are presented. Section III describes the algorithm and CUDA C implementation used to perform shadow removal. In Section IV, we examine the performance of our CUDA C version in comparison to a serial MATLAB version and validate the effectiveness of our implementation. Then in Section V, we describe the impact of our analysis and describe what future work can be done to improve our implementation.

* Related Work

Our implementation of the shadow removal algorithm is very similar to the implementation used in \cite{Akoglu}. Richter et al. \cite{Akoglu} had processes such as color space transformation, thresholding, Otsu’s method, convolution, erosion, and integration. They had also used CUDA C to implement their algorithm and used various optimization methods such as shared memory, tiling, and reduction.

Another similar implementation is the concept proposed in Wu et al. \cite{Wu2020}, where the background and foreground are extracted using Otsu’s Method. Like the concepts proposed in Richter et al. \cite{Akoglu}.

In Qu et al. \cite{Qu2023} a progressive attention mechanism concept was proposed for shadow removal, although this study did not use CUDA C to implement its algorithm.

Besides \cite{Akoglu}, the rest of the methods do not use CUDA C to implement their algorithm. Also, there are no execution time benchmarks provided in those two articles. The methods specified also appear to be very compute-intensive and it may be difficult to accomplish a CUDA C implementation within a reasonable amount of time.

* Methodology
* *Colorspace Transformation*

This is the first process of the shadow removal algorithm, where the original RGB image is converted to a color invariant using (2), (3), and (4). The U component image is part of a YUV image and is calculated using (1). The color invariant image is then converted to a grayscale image using (5). In (2), 0.5 is added to the formula so that the U value never goes negative. When the image is first read, the pixel values only have a range from 0.0 to 1.0.

(1)

(2)

(3)

(4)

(5)

The original input image with its color invariant, U, and grayscale images are shown in Fig. 2 below.



Fig. 2 (a) Original Image

* 

Fig. 2 (b) Color Invariant Image



Fig. 2 (c) U Image



Fig. 2 (c) Grayscale Image

* *Histogram*

The next process in the algorithm is to calculate the histogram for the grayscale image. In this process, there are 255 bins to represent each pixel intensity in the histogram. Each pixel intensity bin is the number of pixels in an image that have that same pixel intensity. The histogram is then used in Otsu’s method to calculate the threshold.

(7)

In our CUDA implementation, shared memory was utilized so that the kernel could employ privatization. Each thread block contained a private histogram with 256 bins and would iterate throughout the image using a stride amount calculated using (7). The kernel would then atomically increment each private histogram bin based on the pixel intensity read. When the thread has iterated through all valid pixels it will wait for all other threads to finish their iterations. Afterwards the first thread in the thread block will transfer the private histogram values from shared memory to global memory.

* *Thresholding and Otsu’s Method*

Otsu's method would then be implemented in order to determine an optimal threshold values for the histogram outputs. Initially, the implemented algorithm calculated the total number of pixels and the weighted sum of pixel intensities using the histograms. The total pixel count and the weighted sum were obtained by iterating through the 256 histogram bins.

Our algorithm then evaluated the between-class variance for each threshold value. For each threshold, it would separate the histogram into background and foreground, calculating their weights and cumulative sums. The mean intensity for both background and foreground would then be calculated by dividing the cumulative sums by their weights. The variance for each threshold was then computed using the equation for between-class variance (??).



The between-class variance in (??) involved the product of the background weights, foreground weights and the square of the difference between the background and foreground means. Our algorithm would then search for the threshold that maximizes the between-class variance. The threshold that was determined to be the optimal value would now be returned to main and would be used when implementing the binarization kernel. The optimal threshold would be determined for both the Grayscale histogram and the YUV histogram.

In our binarization kernel, the algorithm is implemented so that each CUDA thread was set for processing a single pixel of the image. The global thread index is computed using the block index and thread index, which would correspond to the pixel location in the image.

Our implemented binarization kernel involved in comparing the pixel intensity against the threshold. If the intensity of a pixel was less than or equal to the threshold, it was labeled as background. If the intensity of a pixel was greater than the threshold, it was labeled as foreground. This labeling in our binarization kernel assisted in distinguishing between shadowed regions and lit areas of the input images. The binarized images would then be returned into global memory for further processing. Both the Greyscale and the YUV image would be binarized using our implemented kernel. After the binarization kernel execution. The binarized images would be produced for both the Greyscale and the YUV Image.

The output images resulting from the binarization kernel execution are shown below in Fig ???



Fig. ??? (a) Grayscale Binarized Output

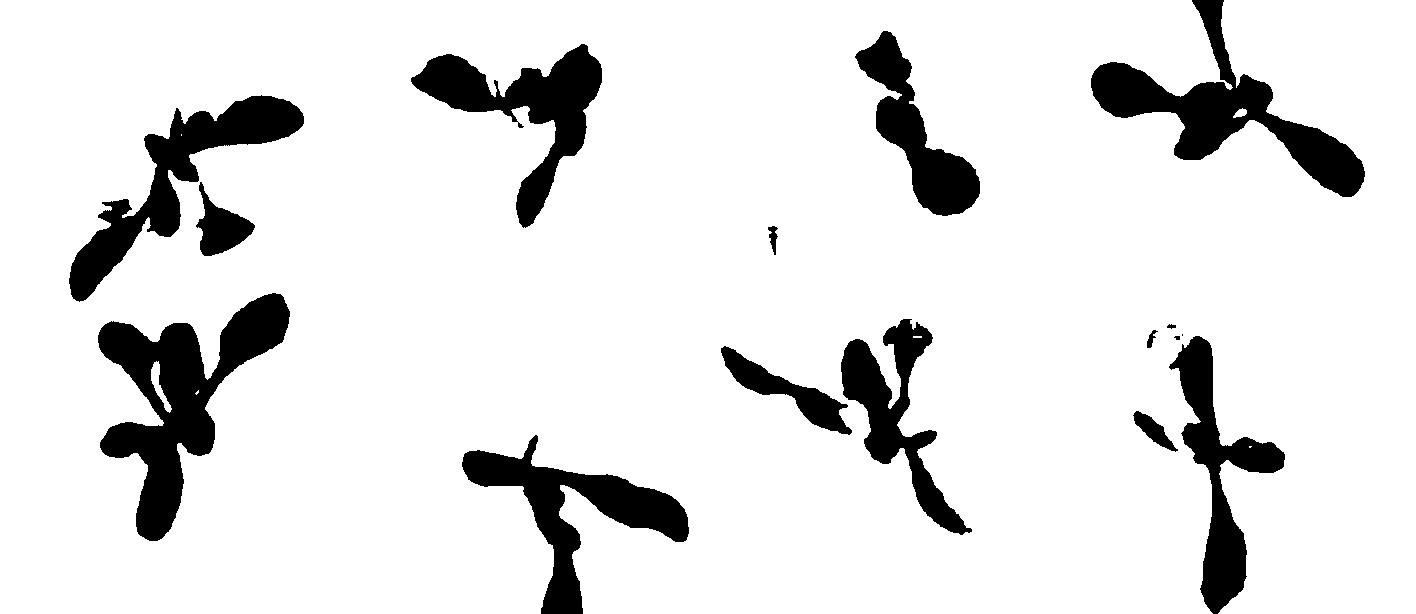


Fig. ??? (b) YUV Binarized Output

* *Convolution*

TBD

* *Erosion*

TBD

* *Integration*

The last process of the algorithm is integration, where the eroded shadow and light mask are used to calculate the RGB ratios. To calculate these ratios, the pixel intensities for the original input image and eroded mask are multiplied and then accumulated, which provides the numerator for the ratio. Then the sum of the pixel intensities for eroded mask images is also calculated, which provides the denominator for the ratio.

(8)

(9)

(10)

Afterwards the RGB ratios, smooth mask image from convolution, and input image are used to generate the final output image with shadows removed using (8), (9), and (10). In the equations RR is the red ratio, RG is the green ratio, and RB is the blue ratio. The IR, IG, and IB variables are the original input RGB pixel values. The S variable is the pixel value for the smooth mask image from the convolution process.

In our CUDA implementation, there are three kernels used in the integration process. The first kernel is used to calculate the numerator value for the RGB ratios. In the kernel, a private sum value is declared in shared memory to calculate the sum of the input image multiplied by the eroded grayscale mask. Each thread uses the stride amount shown in (7), to iterate through the image and atomically accumulate the sum in the private sum values in shared memory for the red, green, and blue channels. The first thread in a block then transfers the private RGB sum values to global memory.

The second kernel is like the first kernel, where a private sum is declared in shared memory. The kernel calculates the sum of pixel intensities for the eroded grayscale shadow and light mask which is used as the denominator for the RGB ratio values.

The results from the first and second kernel are then divided in the host code to generate the RGB ratio values which are used by the third kernel to generate the final output image with shadows removed.

In the third kernel, only global memory is used to calculate the result image, which uses the equations in (8), (9), and (10) to generate the RGB values for the final output image.

* Evaluation and Validation

The results in this section were obtained using a computer that had an Intel Core i7 12800H CPU with an operating frequency of 4.8 GHz. The GPU used to obtain the CUDA executions times was the NVIDIA RTX A2000 Laptop GPU with 20 streaming multiprocessors and a total of 3,328 Ampere-based CUDA cores.

TABLE I

Execution Times Per Kernel for 1548 x 976 Image



TABLE II

Execution Times Per Kernel for 4500 x 4148 Image



* *Colorspace Transformation*

In our colorspace kernel, we were able to achieve a speedup factor of 124x. Each thread in the kernel performed the transformation on one pixel in the original image. Also like in \cite{Akoglu}, only the U component in the YUV transformation was calculated since the Y and V components were not needed. Also like \cite{Akoglu}, we used the atanf() function from the CUDA library which reduced the calculation time. Since each thread works on subsequent pixel, the memory accesses also work on consecutive locations so there is a form of memory coalescing which increases the performance of the kernel. Some improvements that could have been made to our CUDA source code were to put a type specifier on the floating point literals, which we observed having around 3x to 4x speedup compared to our implementation based on results from the NVIDIA profiler tool. The results from the NVIDIA profiler tool also showed the memory throughput being 31.62%. The reason for this low throughput was the large amount of global memory accesses which take a longer time in comparison to shared memory or constant memory accesses. In the kernel, there were 3 global memory reads for the input image and 5 global memory writes for the color invariant, U component, and grayscale image.

* *Histogram*

Our histogram kernel performed worse than the MATLAB serial implementation. This was also the case in Richter et al. \cite{Akoglu} as well. Our kernel performed an average of 0.65x of the MATLAB serial implementation. In our kernel, each loop has 1 global memory read, 1 atomic addition to shared memory, and 1 integer addition. The number of loops is dependent on the image size. Each thread also transfers the private histogram bin values to global memory. According to the NVIDIA profile tool, the histogram kernel has a compute throughput of 45.63% and a memory throughput of 31.64%. The reason for the low throughput may be due to the large number of writes to shared and global memory. Also, the memory accesses are not coalesced, which also affects the performance of the kernel.

* *Thresholding and Otsu’s Method*

Our Otsu’s method was implemented using a CPU bound function while the binarization kernel was implemented using CUDA architecture. In our binarization kernel, we were able to achieve a speedup factor of 14.90x for the Grayscale binarization and a 23.90x factor for the YUV Image binarization for the 1548x976 dimensions. For the 4500x4148 images, we were able to achieve a speedup factor of 13.75x for the Grayscale binarization and a 25.34x factor for the YUV Image binarization. Each thread in our kernel performed binarization on each pixel in the inputting images. Compared to our MATLAB execution, for 1548x976 dimensions the execution time for Grayscale binarization was 2360us on MATLAB and 79us on CUDA. For YUV, our MATLAB execution time was 1863us and 79.33us on CUDA. As observed, MATLAB execution was significantly slower than the CUDA implementation. For 4500x4148 dimensions the execution time for Grayscale binarization was 26413us on MATLAB and 959us on CUDA. For YUV, our MATLAB execution time was 24323us and 959us on CUDA. As observed, MATLAB execution was significantly slower than the CUDA implementation. Our CUDA implemented kernel had a significantly fast execution time primarily due to the parallelism that was offered by our GPU, while our MATLAB code processed the binarizations in a sequential manner on the CPU. For our CUDA kernel, the architecture performs the binarization operation concurrently across many pixels where each of the threads can binarize a pixel simultaneously, which significantly reduces the time to process the images. Comparing the different dimensions, CUDA processing is more advantageous for larger scale images since the high throughput of image data would need to be processed.

* *Convolution*
* *Erosion*
* *Integration*

The three kernels that make up the integration process are the image mapping, image mask combination, and the result image kernel. The total speed up for the integration process was approximately 4.5x, with the result image kernel performing the best out of the three. Despite the result image kernel performing the best compared to MATLAB, there were still improvements that could be made. The NVIDIA profiler tool showed that a 59% speedup could occur if there were no uncoalesced memory accesses. The profiling tool also revealed a 24.23% compute throughput for the result image kernel as well, which is due to the various multiplications, additions, and division operations having to be used. In the kernel there were 6 floating point additions, 6 floating point multiplications, 3 floating point subtractions, and 1 floating point division which contributes to the low compute throughput. There were also 9 global memory reads and 3 global memory writes that contributed to the performance.

Our image mapping kernel also showed a speed up factor of around 3x in comparison to MATLAB. The kernel had 6 global reads and 3 atomic additions to shared memory for each loop iteration. There were also 3 floating point multiplications that contributed to the execution time of the kernel. The first thread of the thread block would also transfer the data from shared memory to global memory using the atomic addition operation. The NVIDIA profiler tool also showed a 73% compute and memory throughput for the image mapping kernel. Since the kernel is always reading the same image mask value from memory, the number of global reads could of have been reduced to 4, which would help with the memory throughput.

The image mask combination kernel performed the worst compared to MATLAB, where the CUDA kernel ran slower at around a factor of 0.1x. The MATLAB implementation seemed to perform well in which it only took around 180 us to compute the sum of the image mask pixel array compared to our CUDA kernel which took an average of 2000 us. The compute and memory throughput were around 70%. There is only 1 global read and 1 atomic addition to shared memory for each loop. Also, only the first thread in a block uses an atomic addition to transfer the private sum value to global memory. The speed up factor did increase slightly with a larger image size, since the MATLAB serial implementation has to execute more operations and the CUDA program only launches more threads which are running in parallel, which does not increase the execution time.

* *Total Speedup*

As can be seen in Table I, our CUDA program had a total speedup of 12.57x for a 1548x976 image. If we were to add type specifiers to our floating-point literals, we would have a speed up factor of 15.73x which is shown in Table II. Both tables show the execution times and speed up factors relative to the MATLAB serial implementation for each kernel. Some kernels were executed multiple times but with different inputs, which is accounted for in Table I and Table II.

A close-up of a plant growing

Description automatically generated

Fig. ??? (a) Original Input Image

A close-up of a plant growing

Description automatically generated

Fig. ??? (a) Final Output Image

* References

M. Wu, R. Chen, and Y. Tong, “Shadow Elimination Algorithm Using Color and Texture Features,” Computational Intelligence and Neuroscience, vol. 2020, p. e2075781, Jan.

H. Qu, C. Tong, and W. Liu, “Image shadow removal algorithm guided by progressive attention mechanism,” Signal, Image and Video Processing, vol. 17, no. 5, pp. 2565–2571, Jul. 2023, doi: 10.1007/s11760-022-02473-z.

Wikipedia Contributors, “Y′UV,” Wikipedia, Apr. 22, 2024. <https://en.wikipedia.org/wiki/Y%E2%80%B2UV> (accessed Apr. 30, 2024).

Dynamsoft, “Image Processing 101 Chapter 1.3: Color Space Conversion,” Dynamsoft Document Imaging Blog, Sep. 13, 2021. <https://www.dynamsoft.com/blog/insights/image-processing/image-processing-101-color-space-conversion/>

‌

‌