Accelerated Shadow Detection and Removal Method

Edward Richter, Ryan Raettig, Joshua Mack, Spencer Valancius, Burak Unal, Ali Akoglu
Electrical and Computer Engineering
University of Arizona, Tucson, AZ, USA
{*edwardrichter, raettig, jmack2545, svalancius12, burak, akoglu*}*@email.arizona.edu*

***Abstract*—Shadows can have a negative effect on the ability of computer vision techniques for object detection, tracking, and recognition. Therefore, ability to remove shadows and byproducts of illumination is an important problem to enable effective object recognition actions. As applications move into levels of higher information extraction and higher required processing speeds, efficient and sophisticated shadow detection and removal becomes even more necessary. In this study we propose a shadow removal method, parallelize using a Tesla P100 GPU, and achieve a speedup of 21.67x on an 18 megapixel (MP) resolution image compared to the same method implemented in Matlab.**

I. Introduction

Due to the vast amount of current image and video data, it is no longer feasible to manually process images. Instead, intelligent algorithms and continually improved throughput are necessary to extract meaning from this raw pixelated data. However, many computer vision solutions are sensitive to noise and conditions in ways that are unintuitive to humans due to the robustness of our own visual system. One example of this is the presence of shadows. Shadows, an area void of light caused by an object obstructing light rays, detrimentally impact the performance of common computer vision algorithms. Generally, shadows contain and/or mimic the features of objects of interests, as well as change the color profile of the objects they rest upon. While these effects rarely impact a human's ability to process a scene, they can greatly affect a computer vision algorithm utilizing pixel-wise transformations to generate meaning.

Contact sensing is a typical exercise in the greenhouse based Controlled Environment Plant Production (CEPP) systems to determine a plant's physical characteristics, a process that is burdensome, labor-intensive, and destructive. Non-contact sensing with computer vision can be applied in the greenhouse environment to determine the overall status of plants and to identify a plant's specific needs at a given time. To be able to analyze the health of a plant accurately, the captured image should be free from artifacts. For example, tip burn, which is caused by calcium deficiency, is very hard to detect when the whole or part of a plant is under shadow. In the greenhouse, overlapping shadows (structural and plant leaf) result with non-homogenous shadowing effect on the plant image. The shadow removal problem is further complicated when the environmental conditions such as the effects of clouds, changes in intensity and angles of light throughout the day, as well as air flow induced canopy movement are taken into account. As such, a computationally efficient and parallelizable methodology to remove shadows prior to extracting semantic information is necessary for the success of CEPP applications that rely on dynamically changing environmental information.

Detecting and increasing the illumination in shaded regions without creating an artifact in the original image are two challenging problems. State of the art shadow removal techniques based on RGB [5], [6] and entropy minimization and color invariance [2], [3] perform best only under a specific angle of light that they are designed for and fail to deal with the complexity of the shadow removal problem in the greenhouse environment. Furthermore, current methods proposed for shadow removal in images and photographs are computationally expensive, requiring complex mathematical operations to identify and reduce the effects of shadows created by objects within a scene.

As with most image processing applications, Graphic Processing Units (GPUs) are an attractive platform for accelerating the capabilities of this application thanks to its high levels of data parallelism. In this work we propose a chromaticity-based shadow detection and removal method and implement on a Tesla P100 GPU to to serve as a vital pre-processing step in real-time CEPP applications. To reduce the work per thread and improve computational feasibility, we utilize algorithmic simplifications to an entropy minimization-based shadow removal technique presented in [2]. This enables us to improve overall system throughput with minimal loss in output quality. As there are no other implementations of this method, however, performance comparisons with respect to other algorithmic methods fail to yield useful insights. As such, the work presented here will be evaluated solely in terms of its standalone performance, both in terms of computational results and shadow removal performance.

The rest of the paper is organized as follows. In Section II we present a literature review of the different shadow removal algorithms. In Section III we present our methodology for implementing a chromaticity based shadow removal algorithm in CUDA-C. In Section IV we present our optimization strategies applied during each stage of the shadow removal process and discuss the results. Finally, in Section V we conclude our work and present future opportunities.

978-1-7281-5052-9/19/$31.00 ©2019 IEEE

II. Related Work

Shadow detection and removal has been a topic of interest in the computer vision community for a long time due to the detriment that shadows have on common image processing algorithms. The shadow removal methods can be categorized as Chromaticity, Geometry, Physical and Texture based algorithms [9]. In chromaticity-based algorithms, it is assumed that the regions under shadows become darker, but retain their chromaticity, the measure of color independent of intensity. Most chromaticity-based algorithms involve transforming from one colorspace to another, where intensity and chromaticity are more separable such as the YUV colorspace. As most of the operations done in chromaticity-based algorithms are pixel-wise transformation, chromaticity-based algorithms tend to be the least-computationally expensive.

While chromacity based methods are one of the fastest methods available, they are susceptible to noise within an image and fail to recognize extremely dark shadows that mask the underlying chromacity of the background and surface cast underneath dark shadows. In the case of physical based methods, they fail to consistently and correctly remove shadows from objects whose chromacity is similar to the chromacity of the background. Despite these drawbacks, physical methods remain more accurate than chromacity methods. The remaining two methods, geometry based and texture based, are the most computationally complex and require the most number of calculations for removal. Another side effect of geometry methods is that they are not designed for objects or scenes with multiple shadows. Areas with many lights indoors perform poorly when compared to other shadow removal methods. Texture based methods do not have this issue, but suffer from the worst computational time complexity of all shadow removal methods since each pixel is compared to the largest neighboring area among methods.

As our main focus is to realize a low-latency and high-throughput shadow removal, we propose a chromaticity-based algorithm. This choice compliments the utilization of GPUs, as a chromaticity-based algorithm primarily consists of pixel-wise transformations that are suitable for parallelization on GPUs. In our search for possible reference shadow detection and removal algorithm, we identified chromaticity-based studies. Gudivika et al. [4], implemented a *Codebook* model on a GPU to achieve a 250x speedup on a foreground segmentation problem. Silva et al. [10], implemented an altered chromaticity-based shadow detection and removal algorithm on a GPU where they are able to achieve state-of-the-art shadow removal performance in 0.03685 seconds on a 1024x1024 image. Macedo et al. [7], proposed yet another chromaticity-based algorithm on a GPU where the image is first split into six channels. Each channel is binarized using Otsu's method [8] before being used to create a shadow mask of the image. The GPU implementation provides a 1.25x speedup on a 640x480 image and a 12.7x speedup on a 3840x2160 image. This method stops at the shadow mask step of a chromaticity-based algorithm, making this algorithm used for shadow detection but not removal.

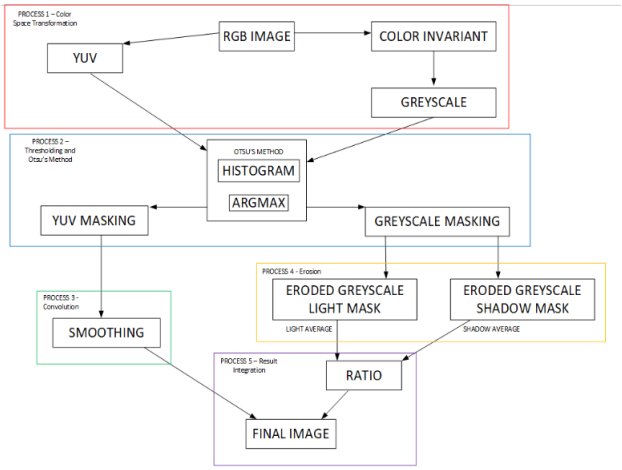


Fig. 1. Step-by-step description of the shadow detection and removal algorithm deployed in this work. We have split the steps that are similar in computation and close together.

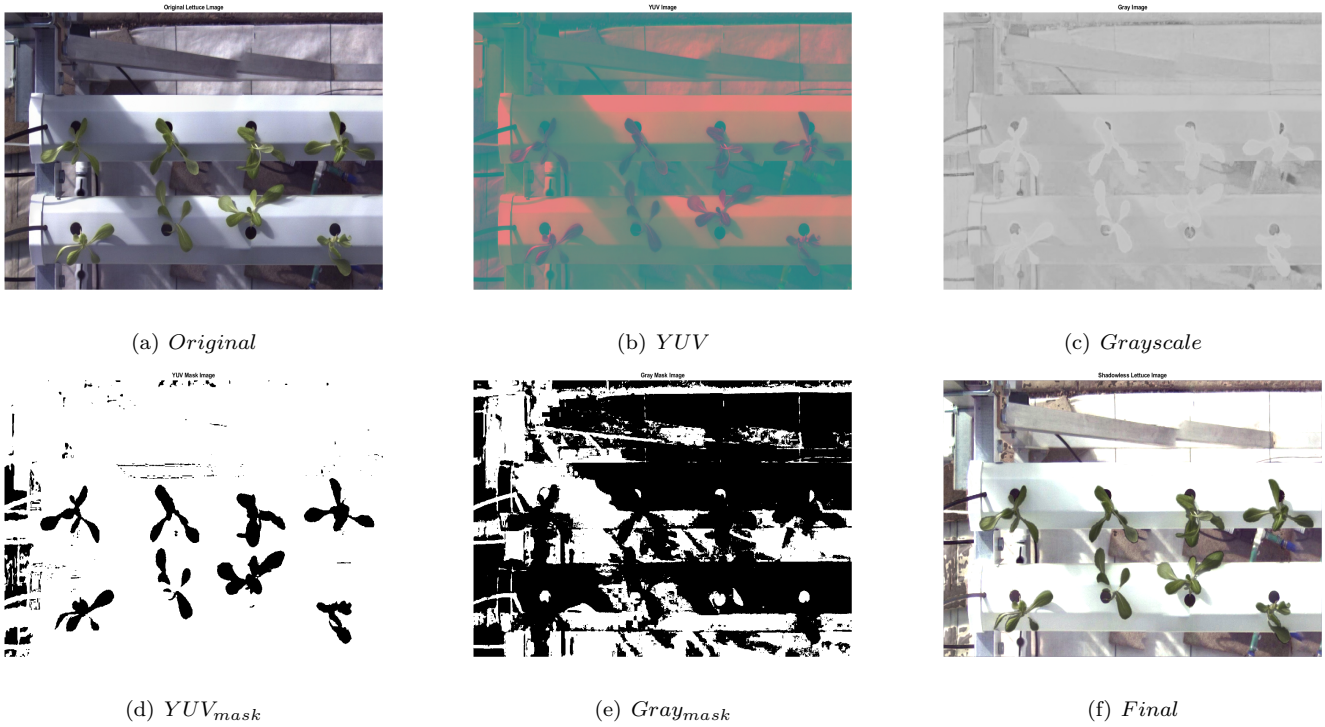
III. Methodology

In this work, we propose a chromaticity-based shadow detection and removal algorithm that involves ten intermediate steps before generating the final Red, Green, Blue (RGB) shadow-less image as illustrated in Figure 1. We partition these ten intermediate steps into five processes that we refer to as *Colorspace Transformation, Thresholding and Otsu's Method, Convolution, Erosion* and *Result Integration*. In the following subsections we describe how these processes work.

1) Colorspace Transformation: This process takes in the original RGB image, as seen in Figure 2(a), and generates YUV and Grayscale versions as seen in Figures 2(b) and 2(c) respectively. To generate the grayscale image we first generate a color invariant image from the original RGB input image.

2) Thresholding and Otsu's Method:: After performing the colorspace transformation, two masks are generated. One mask is generated from the grayscale image seen in Figure 2(c) and the other mask is generated using the U component of the YUV image seen in Figure 2(b). The generated masks of the YUV and grayscale images can be seen in Figures 2(d) and 2(e) respectively.

The masks are generated using Otsu's method [8], which is an unsupervised method of finding a threshold value to differentiate between the foreground and background of a grayscale image. The input to Otsu's method is a histogram of the pixels in the image, which is to approximate the probability density function of pixel intensities. The threshold k dichotomizes the pixels into foreground and background. After creating the histogram, the problem can be viewed through a probabilistic lens by analyzing



(a) *Original* (b) *YUV* (c) *Grayscale*

(d) YUV_{mask} (e) $Gray_{mask}$ (f) *Final*

Fig. 2. Shadow removal steps (a) Input image, (b) YUV version, (c) Grayscale of color invariant image, (d) Mask generated when performing Otsu's method on the U channel of the YUV image created in the colorspace transformation, (e) Mask generated when performing Otsu's method on the grayscale color-invariant image, (f) Final image with shadows removed.

the zeroth (Equation 1) and first (Equation 2) order cumulative moments of the image up to the threshold k.

$$\omega(k) = \sum_{i=1}^{k} p_i \tag{1}$$

$$\mu(k) = \sum_{i=1}^{k} i p_i \tag{2}$$

To determine the quality of a given threshold k, Otsu defines two variance measures: the between-class variance ($\sigma_B^2(k)$) and the total variance (σ_T^2), which are defined using Equations 3 and 4.

$$\sigma_B^2(k) = \frac{(\mu_T \omega(k) - \mu(k))^2}{\omega(k)(1-\omega(k))} \tag{3}$$

$$\sigma_T^2 = \sum_{i=1}^{L} (i - \mu_T)^2 p_i \tag{4}$$

where μ_T is the mean of the entire histogram. Intuitively, a good threshold will have a high variance between classes relative to the variance of the entire histogram. Using this intuition, a "goodness" metric of the threshold can be defined as shown in Equation 5.

$$\eta(k) = \frac{\sigma_B^2(k)}{\sigma_T^2} \tag{5}$$

As σ_T^2 does not depend on k, we can maximize $\eta(k)$ by finding the k, which maximizes $\sigma_B^2(k)$. This is the threshold that Otsu's method returns in order to create the mask.

When mapping Otsu's method to the GPU, we split up the algorithm into six kernels. The first kernel takes in the single-channel image and creates a histogram of pixel intensities. Then, we perform two scans to obtain $\omega(k)$ and $\mu(k)$ as defined in Equations 1 and 2. We then have a kernel that calculates $\sigma_B^2(k)$ for every bin in the histogram (every possible threshold). Then, we use an argmax kernel in order to find the k that maximizes $\sigma_B^2(k)$, which is the threshold calculated using Otsu's method. Finally, the last kernel takes the single-channel input image and the threshold and creates a binarized image based off whether the pixel was less than or greater than the calculated threshold.

3) Convolution: In this step we generate the convolution-pixel, which is the summation of the selected image pixel, and its neighboring pixels, multiplied and accumulated with the kernel matrix as shown in Equation 6.

$$f[x,y] * g[x,y] = \sum_{i=1}^{n} \sum_{k=1}^{m} f[i,k] * g[x-i, y-k] \tag{6}$$

In this equation $g[x,y]$ represents the image matrix we are convolving over, $f[x,y]$ represents our kernel matrix that will be sliding across each pixel of $g[x,y]$.

For the convolution process we implemented three versions: global history convolution, shared-memory 2D con-

volution and shared-memory 1D convolution. The global and 2D shared-memory convolution use a 5x5 kernel matrix that is loaded into the constant memory of the GPU. This kernel is designed such that the four corners of the kernel are 0 while all other entries are 1, as seen in Figure 3(a). The 1D shared-memory convolution still uses this 5x5 kernel in shared memory, but alters the kernel to be fully 1. This ensures the kernel is separable, as is required for a 1D convolution kernel.

For the global history convolution procedure, we implement a batch convolution. The convolution is called *batch* because each thread must bring in more than one piece of information from the global memory, into the shared memory, in a series of batches. This occurs due to the amount of area used in this shared-memory convolution implementation. In a shared-memory convolution the entire image matrix is broken up into a series of square matrices called tiles. These tiles, are where the convolutions take place and are situated in such a way that the amount of information that must be pulled in from these tiles, does not exceed a block's shared memory capacity. During convolution, if the pixel is an edge pixel then the tiles must be surrounded in an amount of padding that is determined based on the kernel's width and height.

1D convolution of the 2D image matrix requires a separable kernel and is executed the same way as the batch convolution. However, the tiles are now broken up into a series of row-tiles and column-tiles, rather than 2D tiles. The first kernel executes the row-wise convolution using shared memory as depicted in Figure 3(b). The second kernel executes the column-wise convolution as depicted in Figure 3(c). Once the column-wise convolutions have finished, the final resulting information is written back into global memory, and this resulting information is the *smooth mask* used in Result Integration process.

4) Erosion: In parallel with the 2D smoothing process of Section III-3, we take the output masks from Otsu's method and pass them through an erosion operation. The goal of erosion is similar to smoothing: produce two closely related masks, a light mask and a dark mask, that have slight overlap in the transition region between light and shadow and can be used to selectively perform operations on the light or dark regions of the image, respectively. The structure of image erosion is quite similar to that of image convolution, with the basic data access pattern being to pass a filter (structural element) over the data and reduce the value of each pixel based on interactions with the values of its neighbors as shown in Figure 4.

While the computations performed on the data are different, the data access patterns required here are essentially identical to those required by 2D convolution. In total, 8 different erosion implementations were considered, but they are primarily trading off on 3 parameters: global versus shared memory, constant versus non-constant declaration of the structural element, and a reworking of the kernel's arithmetic and memory-access operations based on NVIDIA profiler results. In moving from global to shared memory, the same tiling process was applied as described in III-3, with individual threadblocks loading in padded chunks of data from the global memory and working privately on that, only to perform a final write back to the global memory upon completion. In moving to the constant declaration of the structural element, we enabled the compiler to aggressively optimize caching of the structural element that must be read by every thread without needing to worry about the values being changed or overwritten. Finally, in optimizing based on NVIDA profiler results, we found that shared memory based around one dimensional arrays seemed to generate less memory operations than two dimensional arrays, so our two dimensional tile was flattened to a one dimensional array and all accesses were performed relative to that. Notably, in divergence from the convolution discussion, we did not implement any variant of a 1D erosion process based around a separable structural element.

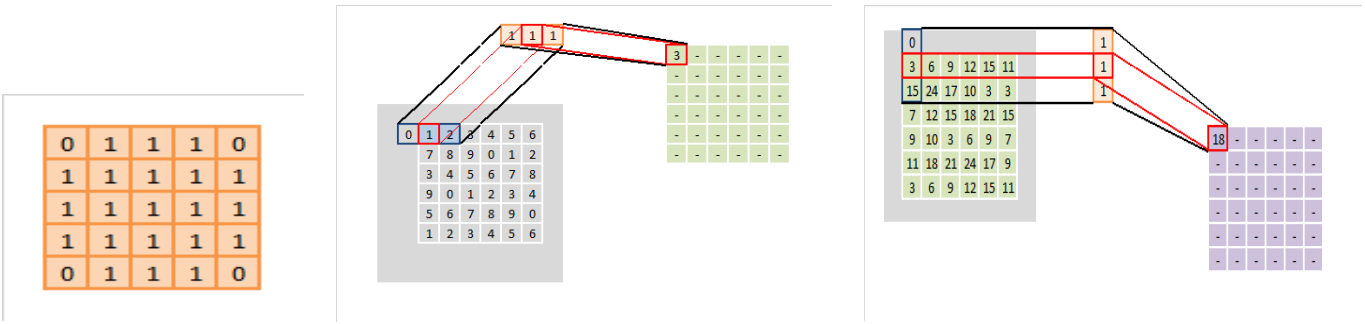
5) Result Integration: In this final step, we first create six maps of the eroded shadow and multiply light masks by the RGB of the input image. Next we accumulate these six maps along with the eroded shadow and lights masks, which requires a total of eight reductions to get the average value of each array. We then use these average values to create our RGB ratio values. Finally, we use the RGB ratio values to map with the input image one last time to remove the shadow and create our output shadow-less image.

IV. Results

In this section we present our results for each of the five processes, and their comparisons against the MATLAB implementation. In all experiments we use image sizes of 640x468, 1548x976, and 4500x4148. All GPU results were obtained using the NVIDIA Tesla P100 GPU that has 56 streaming multiprocessors each with 64 cores operating at 1190 MHz. All CPU results were obtained using Intel i7-8700 CPU operating at 3.2 GHz.

A. Colorspace Transformation

In the colorspace transformation, only the U, chromaticity, is used from the YUV, reducing writes by from 3 to 1. The color invariant (CI) image is then converted into a grayscale image. All three of these conversions translate very well to the GPU as they are close to a map with each thread independently processing a single pixel of the input image and output image. Rather than having three kernels to do each step individually, we merge them into a single kernel. This way we avoid reading input image multiple times by each kernel and increase the floating point operations per memory read from 1.78 to 5 and a speedup of 2.6x. Additionally, this solution reduced the total memory transactions for this step from 9 reads and 5 writes to 3 reads and 2 writes. Furthermore, we utilize *atan()* CUDA-C, which removes the need of a division within the *atan()* function, and leads to a speedup of 1.05x.



(a) $Non-separable$ (b) $Convolution_{Part1}$ (c) $Convolution_{Part2}$

Fig. 3. Convolution process (a) Specialized Non-Separable Convolution Kernel, (b) 1D Convolution, Part 1 - Horizontal Convolution, (c) 1D Convolution, Part 2 - Vertical Convolution.

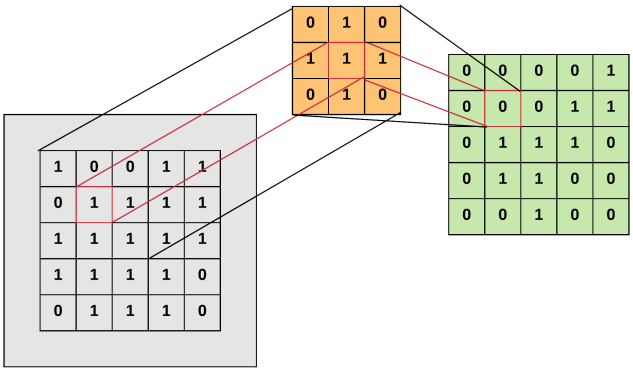


Fig. 4. Binary image erosion using 3x3 structural element

Because threads calculating the YUV, CI, and grayscale images are independent and do not share inputs, using shared memory will not see a speed increase. However, ensuring coalesced reads and writes will see a speed increase. Coalescing memory transactions reduces memory bank conflicts, which will decrease run-time. Our combined kernel, which outputs the grayscale from CI image and the YUV image, also would output the RGB values from the input image into three separate arrays Red, Green, and Blue, with the respective values. These were coalesced writes and also allowed the Result Integration process to have coalesced reads for the input RGB data creating a structure of arrays [11].

B. Thresholding and Otsu's Method

For the histogram kernel, we start with a naive implementation using global memory without coalesced memory reads as a reference implementation to quantify the impact of the optimization strategies.

The naive implementation of calculating $\sigma_B^2(k)$ reads $\mu(k)$, $\omega(k)$, and μ_T from main memory and performs Equation 3 for all 256 elements. The unoptimized argmax implementation is simply a single thread sequentially iterating through the array of $\sigma_B^2(k)$ and returns the k that corresponds to the maximum. The unoptimized mask generation implementation reads the input data in from global memory, determines whether the pixel is above or below the threshold, and writes a one if the pixel is above the threshold, and a zero if the pixel is below the threshold.

TABLE I
MATLAB v CUDA-C Performance Comparison of Otsu's Method using three different image sizes

Image Size	Kernel	MATLAB	CUDA-C	SpeedUp
624x468	histogram	251.25 μs	322.91 μs	0.778x
	$\omega(k)$	15.9 μs	5.52 μs	2.88x
	$\mu(k)$	7.9 μs	5.62 μs	1.41x
	$\sigma_B^2(k)$	28 μs	2.37 μs	11.82x
	argmax	68.2 μs	12.5 μs	5.47x
	maskGen	938.25 μs	5.84 μs	160.8x
1548x976	histogram	800.9 μs	1700.53 μs	0.471x
	$\omega(k)$	15.8 μs	5.92 μs	2.66x
	$\mu(k)$	31.05 μs	5.71 μs	5.44x
	σ_B^2	27.6 μs	2.384 μs	11.6x
	argmax	67.9 μs	12.4 μs	5.49x
	maskGen	3339 μs	29.6 μs	112.6x
4500x4148	histogram	8550 μs	21861.1 μs	0.391x
	$\omega(k)$	17.25 μs	5.24 μs	4.45x
	$\mu(k)$	31.95 μs	5.18 μs	6.16x
	σ_B^2	28.6 μs	2.37 μs	12.1x
	argmax	74.7 μs	11.7 μs	6.39x
	maskGen	37293.7 μs	34.4 μs	108.4x

Table I contains the timing for both the MATLAB implementation and the naive CUDA-C implementation using the three different image sizes. There are three interesting observations from this data: The first is that the histogram generation actually has a performance decrease on the GPU for all three image sizes. This is because the number of bins is significantly smaller than the number of pixels in the image which causes a large number of collision when performing the *atomicadd()* operations. This means that the histogram generation kernel is operating largely in a serial manner, where it is much slower than the CPU serial execution. Another interesting observation is that the majority of Otsu's method is fast on both the GPU and CPU. This is because all of the operations in Otsu's method (excluding histogram and mask generation) perform their operations on a histogram of 256 bins, which is not enough data to require the massive-parallelism of the

GPU. The third interesting observation is that the mask generation is the only kernel that has a significant speedup when mapping to the GPU (>100x) because it is the only kernel that is both largely data parallel and involves a large amount of data. Using Table I as guidance, we choose to focus our optimization efforts on both the histogram generation and the argmax kernels.

1) Histogram Generation Optimization: The first optimization was to reorganize which elements threads read, so sequential threads read sequential memory addresses. This allowed the memory reads to be coalesced, which largely decreased the amount of time waiting for memory. After optimizing the global memory reads, it was observed that due to the small number of bins and large number of pixels that the number of collisions was very large. Therefore, the next optimization was to employ privatization using each threadblocks shared memory in order to minimize the effects of collisions.

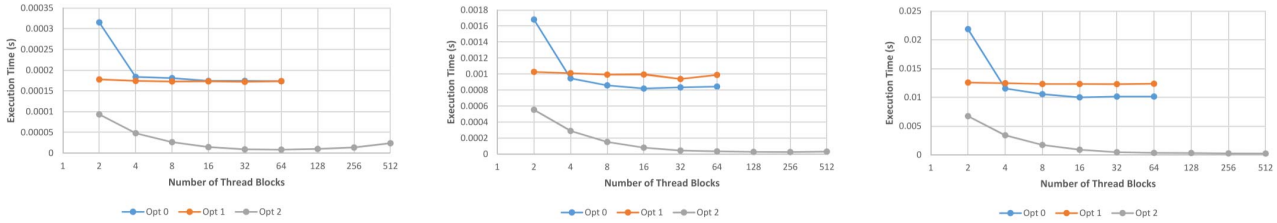
When looking at the histogram generation, there is a trade-off when increasing the number of threads. As we increase the number of threads, each thread has less workload. However, this also means that the amount of contention for the bins increases as there are more threads attempting to write to the same bins. In order to determine the optimal number of threads for each implementation, every implementation on every image size (624x468, 1548x976, and 4500x4148) was ran with a number of thread blocks ranging from 2 to 512. In order to decrease the number of variables in the experiment, we chose to keep each threadblock at 1024 threads, which is the maximum available on the P100.

The results of these experiments can be seen in Figure 5. The first observation to be made is that the shared memory implementation (Opt 2) is significantly faster than both global memory implementations. This is intuitive because privatization reduces the impact of collisions and there is a large number of collisions due to the large number of pixels and small number of bins. Another interesting observation is that the two global memory implementations observe the diminishing returns of increasing the number of threads very quickly (after 4 threadblocks on all images) This behavior differs from the shared memory implementation, which continues to see performance benefits as the number of threads increase to 128 for the smaller image and 512 for the two larger images. As stated earlier, increasing the number of threads decreases the workload of each thread but increases the contention. We have also already mentioned that contention is much less of a problem in the shared memory implementation as the vast majority of the collisions are observed at the shared memory level. The last interesting observation is that it seems that coalescing the global memory reads helps performance when the number of threads is very small (2048) but starts to perform worse than the uncoalesced version as the number of threads becomes large. When diving into this phenomenon using the NVIDIA profiler, it was found that the uncoalesced implementation achieves a higher effective memory bandwidth than the coalesced version. This behavior can be explained with the following details obtained using the profiler: First, the uncoalesced implementation generates many more memory transactions than the coalesced version, which is intuitive because without the memory accesses being coalesced more transactions are required to read the same amount of data. The profiler also shows that both implementations have very low memory bandwidth utilization, which means that despite the additional memory requests generated by the uncoalesced version, the memory system is able to serve those requests without slowing down. Then, as there are more memory transactions in the uncoalesced version, the spatial locality of the memory hierarchy caches data that would not be cached in the coalesced version, making the uncoalesced versoin slightly faster than the coalesced version as the number of threads increase.

2) Argmax Optimization: When we take a closer look at the arrays Otsu's method finds the maximum of, we obseve that the histogram is concave. This means that there is only one maximum and it is a global maximum. The maximum element is the only element that is greater than both its neighbors. Using this observation, an optimized argmax kernel was implemented by having each thread pull one bin value into shared memory and then use the shared memory to look to the neighboring values to determine if it is the maximum. This new implementation takes 2.58 μs which is a 4.81x speedup. While the speedup is not very significant, it moves the argmax kernel from being the slowest operation performed on the histogram to one of the fastest.

C. Convolution

As explained in Subsection III-3 we implement three convolution methods: global, 2D shared memory and 1D shared memory. In the naive global memory implementation, each thread is responsible for multiplying and accumulating between the entire kernel, the input image of the thread's respective focal pixel, and the pixels overlayed by the kernel. If we place the kernel matrix into constant memory, then each thread must access global memory N^2 times, where N is equal to the width of the kernel matrix, for a square kernel matrix. In the 2D shared memory implementation, each thread is responsible for pulling in their respective pixel into the shared memory prior to the actual convolution. However, because our image is much larger than the shared memory size, we tile up the input image so that the information can fit appropriately. We also take into account the padding for when we attempt to bring in information required by the convolution, but is outside the bounds of our input image. We pad these out of bounds areas with 0s. The 2D shared memory implementation is 2.5x faster than the naive global memory implementation. However, with the additional padding, total area we must bring into shared



(a) 624x468 (b) 1548x976 (c) 4500x4148

Fig. 5. Execution time versus number of thread blocks for three different image sizes. Opt 0 refers to the uncoalesced global memory version, Opt 1 refers to the coalesced global memory version, and Opt 2 refers to the shared memory version. Note that each experiment has 1024 threads in each thread block as that is the maximum supported by the P100.

memory is now greater than the number of threads we launched, but less than twice the number of threads we launched. As such, certain threads will be responsible for bringing in more than one piece of information from the global memory to the shared memory, but in doing so we eliminate the memory bottleneck of constantly accessing global memory during our kernel execution. Instead, we are bottle-necked by the calculation, which has complexity of $O(N^2)$ as each thread must iterate through all rows and columns of the kernel and the overlayed pixels.

Our final convolution methodology uses the 1D shared memory implementation. This implementation requires that our kernel be separable, requiring us to modify the kernel slightly to ensure this is true. Instead of possessing a kernel such as depicted in Figure 3(a), we make the entire kernel 1s. This allows us to separate the kernel into a *row kernel* and a *column kernel*, which are used in two separate kernel launches during the program's execution. Rather than needing to convolve around both row and column of the kernel mask, we only convolve the rows. This drops the complexity of the convolution kernel from $O(N^2)$ to $O(N)$. Once the row convolution kernel has finished execution, we send the resulting output matrix as the input image into the next kernel call, the column convolution kernel. This kernel functions exactly the same as the row convolution kernel, except that it convolves around the columns rather than the rows. This has the drawback of not using coalesced reads from the shared memory. This kernel, just as the row convolution kernel, has a complexity of $O(N)$, as such when both row and column kernels execute for the convolution our total complexity becomes $O(2N)$, which is still less than the $O(N^2)$ of the 2D shared memory implementation.

D. Erosion

Following in the convolution kernel's footsteps, the global memory version of erosion was implemented such that each output pixel was assigned a thread that gathered all relevant pixels from global memory and performed the erosion process with what it fetched, a process that placed $O(N^2)$ memory access requirements on each thread in the grid and heavily saturated the global memory. The demand for global memory decreased by moving to a 2D shared memory with tiling implementation, but unintuitively, performance actually decreased or remained the same relative to the global memory version. Analysis in the NVIDIA profiler's program counter profiling tools revealed that a fairly large amount of time was being spent at the `__syncthreads()` primitive, leading to the theory that gains in memory access times achieved in the kernel compute were being traded off with warps stuck waiting for block-level synchronization before they can proceed.

Here we note that due to the small size of the mask and its high frequency of use, we observed that it was being cached aggressively and no performance gains were seen with the placing the structural element into constant memory. Finally, the last optimization explored was related to an observation within NVIDIA's PTX intermediate representation that using shared memory as a 2D array appeared to generate extra load instructions for each of the accesses. As such, the shared array used in the erosion kernel was flattened to be accessed in a purely one dimensional manner, and the result was a nearly 2X drop in execution time relative to the shared memory kernel, with a resulting performance on the order of a 36.72X speedup relative to MATLAB.

E. Result Integration

The result integration is split into four separate kernels. The first kernel maps the input image multiplied by the shadow and light masks. The second kernel sums the light arrays, shadow arrays, and eroded arrays. The third kernel calculates the red, green, and blue ratios from the sums used to reconstruct the shadow removed image. The fourth kernel uses the red, green, and blue ratios and the input image to remove the shadow and create the output image.

The initial map and fourth kernel have independent threads and are parallelizable. The initial map and the fourth kernel benefited with coalesced reads from colorspace conversion process reordering the input image into Red, Green, and Blue arrays. This saw a speedup of 1.0012x.

The third kernel uses reduction to sum eight arrays. The data is taken from global to shared memory. Each thread adds two elements before putting it into shared memory. This initial sum utilizes all threads of the block. If a thread is out of bounds, a zero is assigned to shared

(a) *Original* (b) *Processed*

Fig. 6. The original image on the left, and shadow removed image on the right.

memory. Zero padding reduces thread divergence by allowing all the threads to participate in calculations. Shared memory indexing was implemented in a way to reduce shared memory bank conflicts and also reducing thread divergence. The last 32 threads completing the sum were in a special case where `__syncthreads()` [1] because they are all in the same warp and will not have memory scheduling conflicts. Finally, unrolling the loop optimizes the code by reducing the instructions per loop with the loop iterator and the exit conditional checks by the loop.

The third kernel, which calculates the RGB ratios, only has three calculations to complete and is not inherently parallelizable. However, transferring the data to the CPU to do the calculations and then back to the GPU would take more time than just computing it directly on the GPU and moving on to the fourth kernel. The initial CUDA-C version had one block do all three unique calculations. The optimized CUDA-C version has three blocks each doing one unique calculation. The Result Integration process saw a total of 53.938x speedup over the MATLAB version.

F. Total Speedup

In our CUDA-C implementation of the chromaticity shadow removal algorithm we are able to achieve a total speedup of 21.67x compared to the MATLAB implementation on the 4500x4148 image. A detailed breakdown of the speedups obtained from every optimization can be seen in Table II. Similar to Figure 1, in Figure 6 we show an image taken from the hydroponics based plant production testbed and the output after applying our proposed shadow removal method. Utilizing the streaming capability of the GPU allows us to overlap data transfer time behind computation. For a stream of images at the the same resolution evaluated in this study, we are able to save a total of 3ms on total kernel execution time.

V. Conclusion

In this study we implement a chromaticity-based shadow detection and removal algorithm and map this algorithm to the Nvidia P100 GPU. We discuss our implementation and parallelization approach for each step of the algorithm. We compare its execution time with respect to the MATLAB implementation of the same algorithm.

TABLE II
MATLAB v CUDA-C execution times by process, in seconds, and resulting speedup on the 4500x4148 image.

PROCESS	MATLAB	CUDA-C	SPEEDUP
Colorspace Trans.	2.0544	0.00996	206.265x
Thresho. & Otsu's	0.0914	0.00820	11.146x
Convolution	0.0768	0.07177	1.0701x
Erosion	0.339	0.06091	5.5656x
Result Integration	1.130	0.02095	53.9379x

For future work we plan to investigate altering the variable types from 32-bit float to 8-bit unsigned integers as well as utilizing CUDA-C's texture capabilities.

References

[1] CUDA Toolkit Documentation: https://docs.nvidia.com/cuda.

[2] G. Finlayson, M. Drew, and C. Lu. Entropy minimization for shadow removal. *International Journal of Computer Vision*, 85(1):35–57, 2009.

[3] G. Finlayson, S. Hordley, C. Lu, and M. Drew. On the removal of shadows from images. *IEEE Transactions on Pattern Anal. Mach. Intell*, 28(1):59–68, 2006.

[4] P. Gudivaka, N. Mishra, and A. Agrawal. Gpu accelerated foreground segmentation using codebook model and shadow removal using cuda. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 765–770, May 2017.

[5] S. Kumar and A. Kaur. Shadow detection and removal in colour images using matlab. *International Journal of Engineering Science and Technology*, 2(9):4482–4486, 2010.

[6] G. Ma and J. Yang. Shadow removal using retinex theory. *Intelligent Visual Surveillance (IVS)*, pages 25–28, 2011.

[7] M. C. F. Macedo, V. P. Nascimento, and A. C. S. Souza. Real-time shadow detection using multi-channel binarization and noise removal. *Journal of Real-Time Image Processing*, Jun 2018.

[8] N. Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, Jan 1979.

[9] A. Sanin, C. Sanderson, and B. C. Lovell. Shadow detection: A survey and comparative evaluation of recent methods. *Pattern Recognition*, 45(4):1684–1695, 2013.

[10] G. F. Silva, G. B. Carneiro, R. Doth, L. A. Amaral, and D. F. de Azevedo. Near real-time shadow detection and removal in aerial motion imagery application. *ISPRS Journal of Photogrammetry and Remote Sensing*, 140:104 – 121, 2018. Geospatial Computer Vision.

[11] J. A. Stratton, N. Anssari, C. Rodrigues, I.-J. Sung, N. Obeid, L. Chang, G. D. Liu, and W. mei Hwu. Optimization and architecture effects on gpu computing workload performance. 2012.