Accelerated Shadow Removal

Fabian De La Pena, Luis Jimenez and Rishab Verma
Electrical and Computer Engineering
University of Arizona, Tucson, AZ, USA
(lcjimenez, fdlpm, rishv1995) @email.arizona.edu

***Abstract*—Computer vision algorithm continue to improve to make better sense of the image or videos it processes. In order for the computer to interpret the data it "sees" correctly and run calculation for given intend, the image must be insusceptible to noise and emerging shadows that alter pixel values. If such artifacts are not extracted in a preprocessing stage correctly, it can impact the accuracy of a computer vision algorithm. This paper presents, a real-world solution to detect and remove shadow in images from a computer vision application used for a Controlled Environment Plant Production (CEPP) system which intends to determine a plant's physical characteristics. The shadow removal detection and removal methods are to be parallelized using a Tesla P100 GPU. An expected speedup of 20x on a 18 megapixel (MP) image is expected in compared to a serial implementation in MATLAB.**

I. Introduction

The advancements in computer power have allowed computer vision algorithms to have a place in a wide variety of fields. To mention some examples, manufacturing business are able to identify product defects in real time thanks to computer vision algorithms. In the medical field, computer vision systems such as MRIs, CAT scans and X-rays are used to detect abnormalities in the body. All these accomplishments are done by using digital images along with a series of image processing algorithm steps, so that a machine can accurately locate the points of interests and operate on these accordingly to an intend. This project aims to detect and remove shadows from a greenhouse structured inside the University of Arizona's, Controlled Environment Agriculture Center (CEAC).

The CEAC serves as a research and multidisciplinary educational complex for plant sciences and agriculture studies. In there, contact sensing is a typical exercise to determine a plant's physical characteristics. Manually interacting with the plants to evaluate their development can be burdensome and labor extensive. Therefore, a proposed solution to this task is to construct a plant monitoring system using machine vision. As for an early milestone for this system, the plant image must isolate and extract any shadow source since it will impact the understanding of the plants physiology. For example, a calcium deficiency alert could be triggered due to an overlaying shadow (whether structural or clouds) that changes the color aspect of the plant's leaves. Occurrences like these could lead to a wrongful response of plant treatment, thus the ability to remove shadows prior analyzing semantic information is necessary for the success of an automating plant treatment system.

Identify applicable funding agency here. If none, delete this.

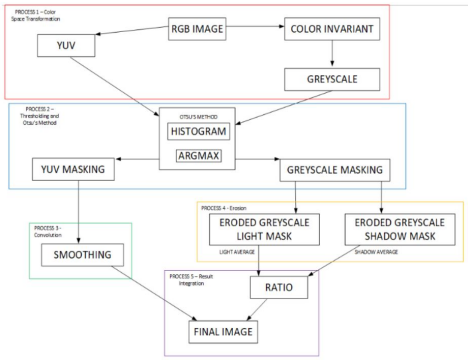

Fig. 1. Step-by-step description [1] of the shadow detection and removal algorithm deployed in this work. We have split the steps that are similar in computation and close together.

Making the shadow removal task a merely image processing solution, it will involve a repetitive iteration from pixel to pixel calculations opening the opportunity for the utilization of Graphic Processing Units (GPU) to achieve a high levels of parallel data processing. As well, it will enable a faster access to pixel values prior computation reducing the overall execution time of the shadow removal algorithm. In this work we propose a chromaticity-based shadow detection and removal method to be run on a Tesla P100 GPU. In chromaticity-based algorithms, it is assumed that the regions under shadows become darker but continue to retain their chromaticity. With this in mind, it is expected that when doing a transformation from one color space to another, the intensity and chromaticity will become separable such as in the YUV color space. Therefore, this proposed solution will not be suitable for extremely dark shadows that mask the underlying chromaticity of the background. The algorithm will be divided into five main steps: Color Space Transformation, Thresholding and Otsu's Method, Convolution, Erosion and Result Integration. These steps will be described in detail in Section III followed by optimization strategies applied in each stage in Section IV. As comparison to a linear MATLAB implementation, the parallelized implementation of this this chromaticity-based approach of shadow detection and removal is expected to be faster ($\sim 20X$).

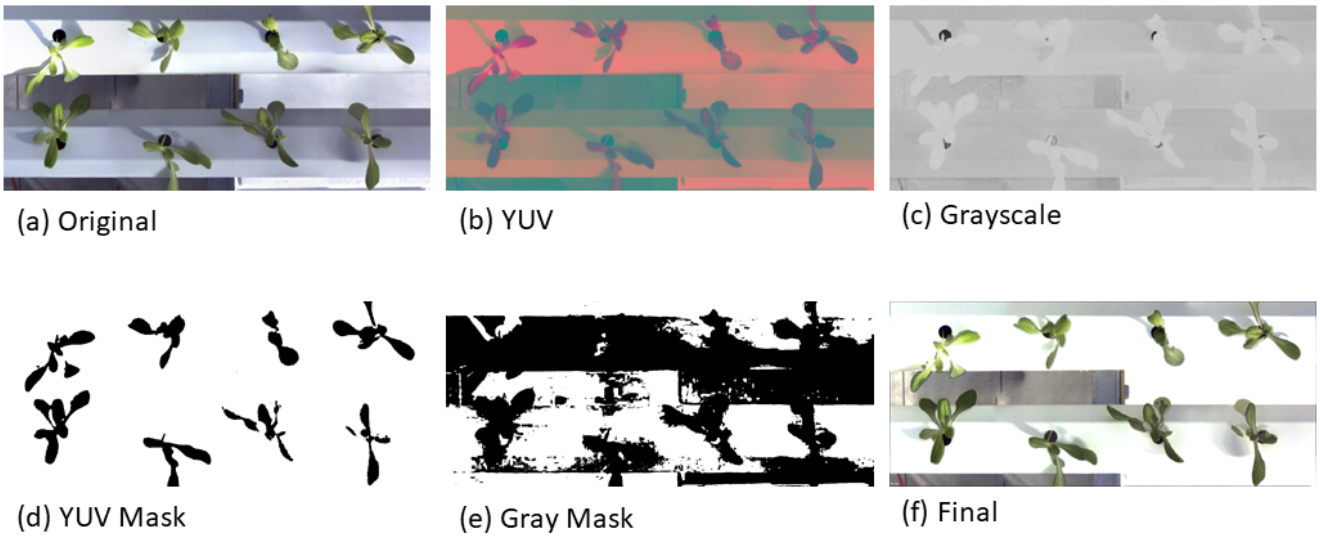

Fig. 2. Shadow removal steps (a) Input image, (b) YUV version, (c) Grayscale of color invariant image, (d) Mask generated when performing Otsu's method on the U channel of the YUV image created in the colorspace transformation, (e) Mask generated when performing Otsu's method on the grayscale color-invariant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

II. Related work

Due to our dependencies on many computer vision applications, the benefit of shadow removal from images/videos has increased value to their associated computer vision application. Successful implementation of shadow removal techniques in an accelerated manner add to the improvements of current image processing algorithms.

With recent developments in image recognition via Machine learning models, there has been significant improvements in unsupervised learning convolutional neural network based algorithms [2]. While it is important to know the machine learning self supervised implementation for shadow removal exists, our team's objective was to utilize a parallelized implementation on a Tesla P100 GPU via a chromaticity-based algorithm [1]. In chromaticity based algorithms, the assumption is that regions under shadows become darker, but retain their chromaticity, the measure of color independent of intensity.

In a chromaticity-based algorithm, most of the computation requirements arise from pixel-wise transformation, convolution calculations and color-space conversions. All these computations are not expensive, however serially they are and, as image/video quality improves the amount of pixels will increase which in turn increase the number of inexpensive computational tasks.

However, chromaticity based algorithms come with their own set of drawbacks. While these algorithms are among the fastest ones available, they are susceptible to extremely dark shadows and do not perform well when there is noise in the image. There also exist other shadow removal techniques, namely - Geometry, Physical and Texture based algorithms [3]. From research on physical based methods, while these method remains to be the most accurate, we found inconsistent shadow removal results due to the background and shadow sharing the same chromaticity[3]. Geometry based and texture based, require expensive computations which make the shadow removal on images challenging.

This paper focuses mainly on the implementation of a solution for image shadow removal in an accelerated manner via low latency and high throughput computations. We find that the utilization of GPUs for parallelizing the large number of inexpensive image pixel wise implementations produced highly acceleration results. Through the course of our implementation, and as compared to a serial implementation on Matlab, for a plt5 - 1416 x 614 pixels sized image, we found speed up to 20X in the overall implementation. Such results showcase the benefits of a GPU based parallel chromacity algorithm implementation over the computationally expensive nature of unsupervised machine learning based model or the serial chromacity based implementations.

III. Methodology

In this report, we propose a chromaticity based implementation (inspired by [1]) for image shadow detection and removal which involves 5 intermediary steps (Showcased in Figure 1) prior to producing a shadow less image output from an image with object shadows in them. Over the course of this section, we will provide details on these 5 process:-

A. Color Space Transformation:

In this process, there are mainly 3 sub tasks required - RGB input image to YUV image conversion,RGB input image to Color invariant image conversion and RGB input image to Grayscale conversion . As depicted in Figure 1, RGB input image to Color invariant and Grayscale conversions go hand in hand as compared to RGB input image to YUV conversion. All the resulting 2 color space transformation steps are showcased on processing a plt4.ppm - 1548 x 976 pixels image of plants with shadows in Figure 2 - steps a and b.

B. Thresholding and Otsu's Method:

Once we have the color transformed Grayscale and YUV images ready, we generate two masks. The first mask is generated via the grayscale image as seen in Figure 2 step e and the second mask is generated by the YUV transformed image as showcased in Figure 2 step d. These masks are generated using Otsu's method [5]. Otsu's method requires obtaining the threshold value, the key towards differentiating between foreground and background of a grayscale image. As an input to the Otsu's method, we supply a histogram bin array of the pixel values of both the grayscale and YUV images since these values correspond to the probability density function of pixel intensities of both the images. 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 the zeroth (Equation 1) and first (Equation 2) order cumulative moments of the image up to the threshold k [1].

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

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

Once we have the threshold value and the probability density function of the intensity values, Utilizing Otsu's method, we define between-class variance ($(\sigma^2)B(k)$) and the total variance ($\sigma^2 T$), which are defined using Equations 3 and 4.

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

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

In the equation above, ($\mu\tau$) is the mean of the histogram. We further define the metric of the threshold by Equation 5 as the assumption here is that the K value will have a variance between classes relative to the variance of the entire histogram .

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

We then split this entire process of Thresholding and Otsu's method in 6 different kernels. The first kernel takes a grayscale and YUV image to create a histogram of pixel intensities. Utilizing scan operations, we obtain $\omega(k)$ and ($\mu(k)$) as defined in Equations 1 and 2.

In order to calculate ($\sigma^2 B(k)$) of every bin in the histogram, we assign another running sum kernel. Post this kernel, we pass in the outputs to a kernet which calculate the argmax to find the threshold value which maximizes the $\sigma^2 B(k)$, 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 [1].

C. Convolution:

The goal for the convolution step is to filter the YUV binary mask obtained from the previous process with the smoothing kernel show Figure 2(a). A convolution operation works by performing an elementwise multiplication with the kernel and a window of the source image with the same dimension as the kernel. Then the results are added into a single output pixel as seen in Figure 2(b). The mathematical model can be seen in Equation 6 where $g[x,y]$ representing the source matrix is convolving over the kernel matrix $f[x,y]$. There were two

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

implementations for the convolution, a naïve based approach where only global memory was involved and an optimized approach that utilized shared memory. Both of these methods read the kernel entries from the GPUs constant memory. For the naïve approach, each thread had to access from global memory all the information needed to convolve over the kernel and return the new value to the pixel it was mapped. On the other hand, the shared-memory approach consisted on allowing each individual thread to bring information from global memory so that other threads that required access to the same information will now read from shared memory at a low latency access time. In general, the source image is broken down into tiles where these are stored in shared memory. Convolution thus now take place in shared memory tiles rather than in global memory. As a return, threads are now collaborating to reduce global memory reads allowing for a much higher throughput of data (meaning more floating-point operations per data access). The result of the convolution

step yield a smooth mask image used in the Result Integration process.

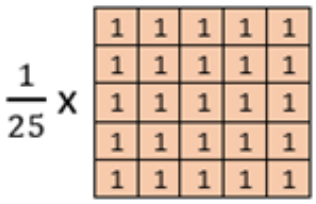

Fig. 3. Kernel used to smooth out YUV binary mask

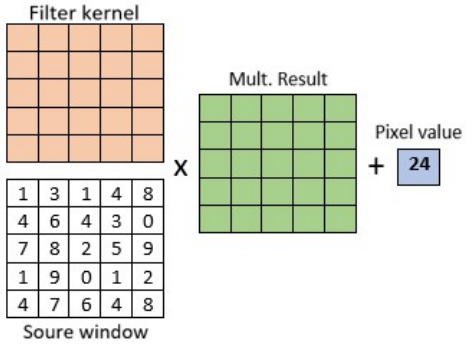

Fig. 4. Basic convolution processes done per thread to return a new pixel value

D. Erosion:

In the convolution step, we utilized the output masks from Otsu's method and passed them into the erosion kernel. Similar to smoothing, the goal of the erosion kernel is to produce 2 closely related image masks , one which is light and the other - dark. These masks help in identifying the overlap in the transition region between light and shadow and which is used to selectively perform operations on the light or dark regions of the image.

The output results from the erosion kernel are similar to that of the convolution kernel images, with the key difference being the basic data access pattern in this kernel is being passed as a filter over the data and reduces the value of each pixel based on the neighbor pixel interaction. While the computations performed on the data are different, the data access patterns required here are essentially identical to those required by 2D convolution

E. Result Integration:

The final step of the shadow removal GPU implementation project requires creation og 6 maps of the output images from the erosion kernel followed by a multiplication and weighted sum with the RGB pixel inputs of the original input image. Post this multiplication,we average the values to obtain RGB ratio values. These averaged RGB values are then mapped out to produce a shadow less image.

IV. Results

Through the course of this section, we will present the results for each of the 5 processes in the problem scope of shadow removal through a chromaticity based algorithm. In all experiments we use image sizes of 1416 x 614 (plt5.ppm), 1578 x 976 (plt4.ppm), and 4500 x 4148 (plt.ppm).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 [1].

A. Color Space Transformation:

In this process, there are mainly 3 sub tasks required - RGB input image to YUV image conversion, RGB input image to Grayscale conversion and RGB input image to Color invariant image conversion. As depicted in Figure 1, RGB input image to Color invariant and Grayscale conversions go hand in hand as compared to RGB input image to YUV conversion. For all three implementations, in order to optimize, we combined the computations into one CUDA kernel since each thread is responsible for image pixel wise calculation. 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 [1].

Image size	Matlab (serial implementation) Kernel Exec. Time (ms)	Naive Individual (Speed of #1a+1b+1c) Kernel Exec. Time (ms)	Naive combined kernel (Speed of #2) Kernel Exec. Time (ms)	Combined coalesced reads (Speed of #3) Kernel Exec. Time (ms)
1416 x 614	108.03	0.499008 (Ci)+ 0.148256 (g) + 0.188224 (yuv) = 0.827	0.548192	0.43213
1578 x 976	177.5	0.844608 (CI) +0.245792 (g) + 0.310624 (YUV) = 1.39	0.917376	0.71287
4500 x 4148	2450	10.066720 (CI) + 2.878752 (g) + 3.630880 (YUV) = 16.56	11.037024	9.87642
Speedup for the largest size image (4500 * 4148)	**Baseline**	**247X on baseline**	• **1.5X on previous implementation** • **~248X on baseline**	• **1.1X on previous implementation** • **~250X on baseline**

Fig. 5. Timing analysis of each implementation of the color space transformation step

For RGB input image to color invariant and grayscale conversion, the GPU parallelized implementation translates well as each thread processes a weighted sum for each pixel of the RGB input image and outputs the corresponding output image. We utilize atan() CUDA-C, which removes the need of a division within the atan() function, and leads to a speedup of 1.05x [4].

For RGB input image to YUV conversion, Similar to the CI to Grayscale conversion, the GPU parallelized implementation translates well as each thread processes a weighted sum for each pixel of the RGB input image and outputs the corresponding output image.

With regards to optimizations, we utilized coalesced memory read and writes by creating Red, Green and Blue pixel value arrays from the input image from the host side. This optimization technique helped us obtain a 1.2X speed up. A key important point to note is that shared memory implementation does not have a speedup benefit as there is no sharing of data between different threads in the color space transformation process.

Looking at Figure 5, we notice parallelized implementation of the color space transformation to have an overall speedup of 250X as compared to the serialized matlab implementation. Within the individual parallelized implementations, we can see a consistent speed up in the optimization strategies utilization as well.

B. Thresholding and Otsu's Method:

For all the kernels in this process of the implementation, we only implemented the naive versions of each kernel.

For the first kernet of step 2, histogram kernel, we only implemented a global memory based naive implementation. THis kernel does not have coalesced memory reads and as a result, there are areas of improvements for this kernel. 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 naive implementation requires reading all elements from the global memory. The unoptimized argmax implementation is simply a single thread sequentially iterating through the array of σ^2 B(k) which returns k that shows the maximum.

For the following mask generation kernel, in the naive implementation, we read the input data from the global memory to determine whether the pixel is above or below the threshold, then we write one if the pixel is above the threshold or zero if the pixel is below the threshold into an array.

C. Convolution:

As mentioned in Section III there were two implementations for the convolution method: A naïve based and a shared memory approach. In the naïve approach each thread access global memory N^2 times, where N is equivalent to the width of the kernel matrix. When shared memory was utilized, convolution data was now been access at a low penalty access time. Although in theory this should have improved the execution time the timing measurements dictated otherwise. The shared memory implementation appeared to add more overhead in the process and utilize a greater amount of threads in order to retrieve all the needed information from global memory. That said, a revise implementation is suggested where idle threads and coalescence memory accesses is closely review.

D. Erosion:

For erosion, a total of three kernels were implemented. Version 1 was a naive implementation that implemented a convolution working in global memory. Version 2 was a modified version which made use of shared memory and tiling in order to attempt to improve performance. Version 3 was similar to version 2, but, it added the use of constant memory for the structuring element, noting that it is never modified by any of the threads. After testing each version on all available images, the best version of the erosion kernel was the first, naive version. The other versions were slower. It is hypothesized that the reason this occurred is that the additional overhead of transferring data from global memory to shared memory along with thread synchronization was larger than any benefit derived from the optimizations for the data sizes being worked on in this project. However, all three implementations were significantly faster than the Matlab implementation, with the naive implementation being 4x faster than the Matlab equivalent. The figures below show the speed of all versions on a 4K and a non 4k image to illustrate relative performance.

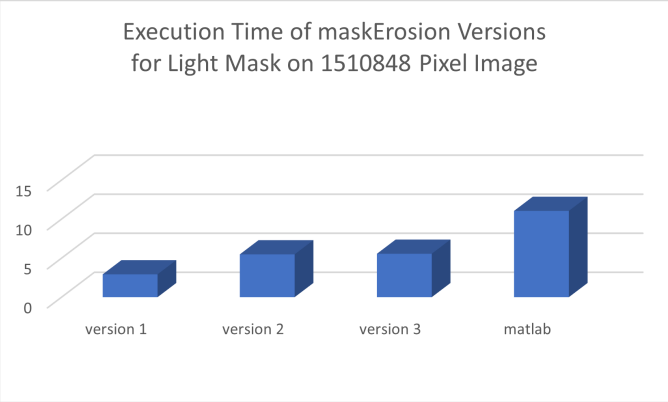

Fig. 6. Erosion kernel and Matlab performance on non 4k image.

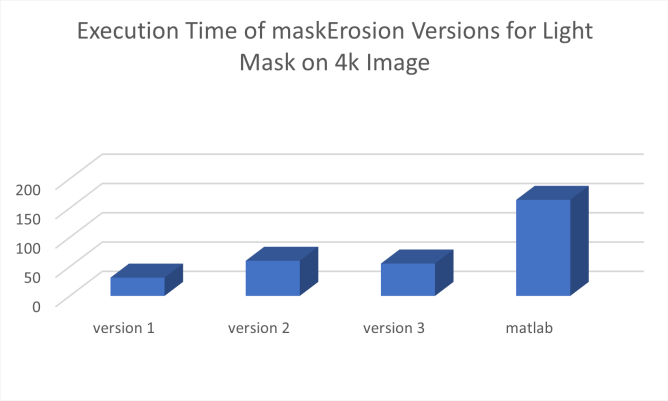

Fig. 7. Erosion kernel and Matlab performance on 4k image.

E. Result Integration:

For result integration, a total of 4 kernels were used.

The first one generated, per image channel, 2 arrays: an array of the channel values multiplied by the corresponding eroded shadow mask value and an array of the channel values multiplied by the eroded light mask value. This kernel processed one pixel per thread.

These arrays were then fed to the second kernel, which was used to perform one reduction per array to get their total sum. This kernel was designed such that it would perform as many sums in shared memory as possible. Sums would be performed on each block and then the partial sum of a block would be added to obtain the final sum.

These sums were then passed to the third kernel, which calculates the RGB ratios. The ratio for each channel is calculated in each thread since it is a simple calculation. As a result, only three threads are used by this kernel.

The final kernel builds the final resulting image, pixel by pixel, with each thread assigned to a pixel. Each channel for each pixel is calculated using the corresponding channel ratio and the corresponding value of the smooth mask.

The speed of all kernels in this process proved to be significantly faster than the equivalent implementations in Matlab, providing a 22x speedup.

V. Total Speedup

plt4 - 1548 x 976 pixels			
Process	MATLAB (ms)	CUDA-C (ms)	Speedup (X factor)
Colorspace Trans	262.632	1.166336	225.2
Greyscale mask generation	5.919	4.055008	1.5
Erosion	19.369	4.402176	4.4
YUV Mask Generation	6.466	4.315872	1.5
Smoothing	4.963	3.059712	1.6
Result Integration	83.173	3.71952	22.4
Total Speedup	382.522	20.718624	18.5

Fig. 8. Timing analysis on image 1 - plt4.ppm - 1578 x 976 pixels

plt5 - 1416 x 614 pixels			
Process	MATLAB	CUDA-C (ms)	Speedup (X factor)
Colorspace Trans	192.049	0.688896	278.8
Greyscale mask generation	4.176	2.533024	1.6
Erosion	10.932	2.521344	4.3
YUV Mask Generation	4.575	2.683968	1.7
Smoothing	6.99	1.814784	3.9
Result Integration	40.888	2.350272	17.4
Total Speedup	259.6	12.6	20.6

Fig. 9. Timing analysis on image 2 - plt5.ppm - 1416 x 614 pixels

For our parallelized implementation on the TestP100 GPU via CUDA-C, our implementation of the chromaticity based shadow removal algorithm achieved a total speed up of 18X for plt4.ppm - 1578 x 976 pixels image and 20X for plt5.ppm - 1416 x 614 pixels image. A detailed breakdown of the kernel wise speed up can be seen in Figure 8 and Figure 9.

VI. Conclusions

In this paper, we implemented a CUDA-C based parallelized chromacity algorithm for shadow removal on images on 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. For future work, we'd like to explore following optimizations :-

- Optimization strategies for Thresholding and Otsu's method (Step 2)
- Altering the variable types from 32-bit float to 8-bit unsigned integers
- Explore CUDA streams
- Utilize padded and masked images to avoid thread divergence

References

[1] E. Richter, R. Raettig, J. Mack, S. Valancius, B. Unal and A. Akoglu, "Accelerated Shadow Detection and Removal Method," 2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA), Abu Dhabi, United Arab Emirates, 2019,

[2] Florin-Alexandru Vasluianu, Andres Romero ´ ETH Zurich, Luc Van Gool ETH Zurich, Radu Timofte ETH Zurich, "Self-Supervised Shadow Removal"

[3] 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.

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

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

[6] Huang, Chunyan, et al. "AN OTSU Image Segmentation Based on Fruitfly Optimization Algorithm." Alexandria Engineering Journal, Elsevier, 17 July 2020