Parallel Image Processing with CUDA

Austin Bond , Thomas (TJ) Davies , Andrew Fleming , and Justin Newman

James Madison University Department of Computer Science, Harrisonburg, VA 22801, United States

Github Repository, Trello Board

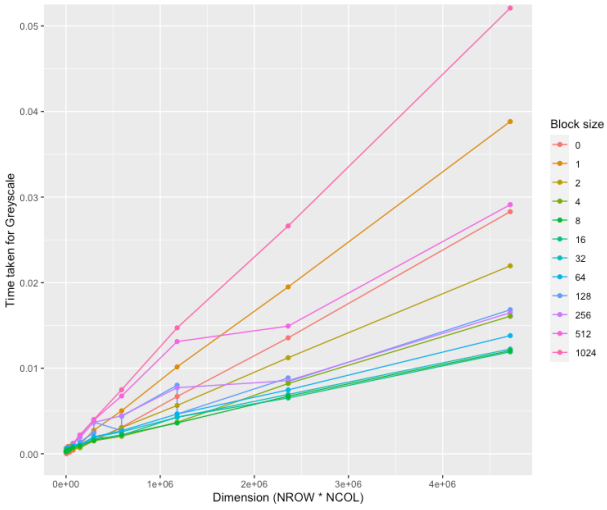
We have spent this semester developing CUDA implementations of many popular image processing techniques. CUDA, or Compute Unified Device Architecture, is a parallel computing collection of libraries, tools, and technologies developed by NVIDIA. It allows software developers to harness the power of Graphics Processing Units for general-purpose computing tasks, not just graphics rendering.

Introduction

Our final project includes both serial and parallel implementations for many image processing techniques. For our project we have chosen to implement greyscale, rotation, targeted color extraction, and pixel sorting. We were able to see significant speed up on all of our image processing techniques and tested them on a variety of images. "Correctness" with image processing is often in eye of the beholder, so we used "Image Magick" to compare images before and after filters were run in addition to manual checking. Our one-touch testing script can be run using the command "bash test.sh" from within the project directory. This script begins by testing all of our deterministic options for correctness. "Deterministic Options" are options for our program that will result in the same image every single time. The script will then run our program using every option and a variety of thresholds. The timing results for each option-threshold pair will be printed to standard out and the images will be stored in a "test" directory.

Greyscale

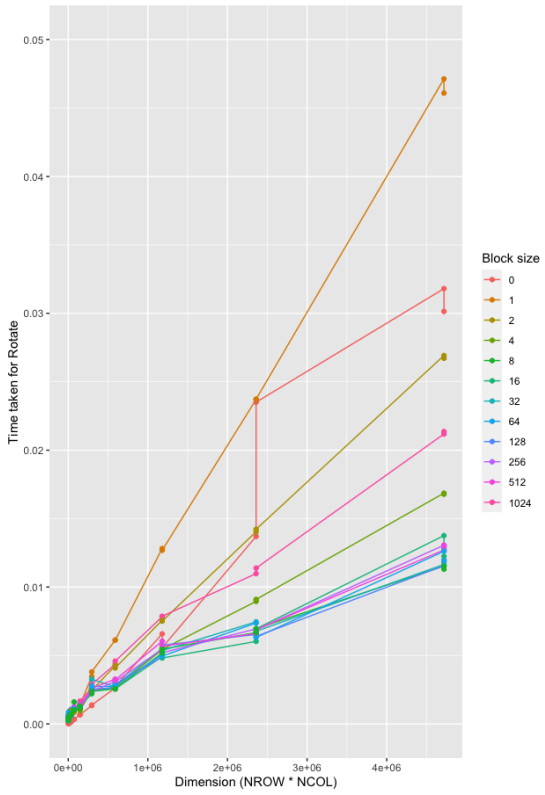
This greyscale function takes an input image represented by row pointers and converts it to a greyscale image. The dimensions of the input image are given by width and height. The function iterates over every pixel in the input image using nested loops. The outer loop iterates over the height (rows) of the image, while the inner loop iterates over the width (columns) of the image. For each pixel, it calculates the average of the red, green, and blue channels to obtain a greyscale value. This greyscale value is then assigned to the red, green, and blue channels of the corresponding pixel in the output image. The alpha channel of the input image is preserved, meaning that the transparency of each pixel remains unchanged in the output image. The pixel is then stored in a new image buffer named "out_pixel" which corresponds to the memory addresses in the parameter out_row_pointers. After this function is executed, out_row_pointers contains the greyscale version of the input image.



Taking an existing serial function and converting it to a working CUDA kernel was, a huge obstacle. Careful consideration of memory allocation was needed to move data correctly to GPU memory in order to perform GPU computations. Researching how libpng allocated memory for its read function was crucial to figuring out a CUDA implementation. Tinkering with the original libpng read function proved to be too difficult, so we compromised with a basic copying operation into new CUDA synced memory. Coupled with trial an error, we successfully transferred the necessary png image information onto synced CUDA memory. Afterwards, modifying the serial imaging processing functions to account for the parallel nature of GPU computation was a chore. Establishing the technique of distributing work across grid and threads is conceptually tough. Nonetheless, we built working solutions for some of our basic filters. The next major roadblock that we encountered was that we experienced very slow saving speed. It slowed down our testing procedures to a crawl. Upon further investigation, we discovered that the issue was in the write utility. After reviewing documentation of libpng, we discovered that a medium amount of image compression was enabled by default in the write function. For the time being we have disabled this compression, which immensely improved the write speed of massive input files. Testing will be far faster as a result.

Rotate

The rotate90 function turns an RGBA image 90 degrees to the right. You might see this used in applications like image editing apps, computer vision, or just to show a picture the right way up. The function needs four things to work: two sets of row pointers (one for the input image and one for the output image) and the width and height of the image. Inside the function, there are two loops that go through all the pixels in the input image. The outer loop iterates through the rows (y-axis) of the image, while the inner loop iterates through the columns (x-axis).



In order to find the pixel's new position, our function switches the x and y positions and then subtracts the original y position from the height of the image minus one. After that, it moves the RGBA values of the input pixel to the output pixel in the new spot. The rotate90 function is a simple and efficient way to rotate images and can be added to bigger programs or libraries very easily. It's a helpful starting point for more complicated image tasks, like flipping, mirroring, or resizing pictures. By using this function, people can make cooler image editing tools and algorithms to improve their apps and fix all sorts of image-related problems.

The rotate90 function benefits from its CUDA implementation quite a bit. As the block size increases, the time taken decreases up until a size of around 64. After that all times are around 1/3 of the serial implementation and continues to scale as the image size increases. This is one of our deterministic functions which means that every single time this runs it will result in the same output image. In order to test this we used ImageMagick's compare command to compare the serial and parallel implementations. We would also do manual checking to determine whether or not the image was rotated correctly.

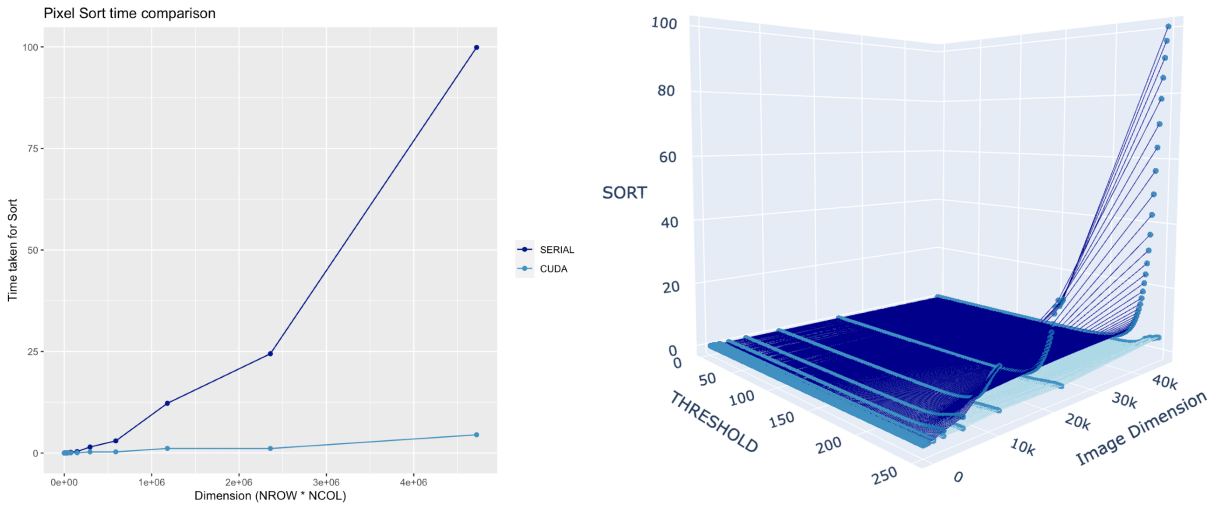
Pixel Sort

Pixel sorting is an artistic method where pixels in an image are sorted based on their brightness or color values, creating visually appealing patterns or effects. In this specific implementation, the sorting is done row by row, and only pixels below a certain brightness threshold pixel segments are sorted. The getBrightness which was remaned computeBrightness in the parallel code is a helper function which calculates the brightness of a pixel by averaging its red, green, and blue components. We denote the brightness as ΔA.

$$\Delta A = \frac{(R+G+B)}{3}$$

The algorithm can be broken down into the following steps. The pixelSort function iterates through each row of the image, sorts the pixels below the threshold in segments and writes the sorted pixels to the output image. It sorts the unsorted pixel array based on the brightness using a bubble sort algorithm. The algortihm uses helper functions returns the x-coordinate of the first dark pixel in a row by checking if $\Delta A <$ threshold, starting from a given x-coordinate. If all remaining pixels are brighter than the threshold, it returns -1.The algortihm uses another helper function to return the x-coordinate of the next bright pixel in a row by checking if $\Delta A >=$ threshold. If there are no more bright pixels in the row, it returns the last x-coordinate in the row.

The main loop in the pixelSort function iterates through each row in the input image. It calculates the length of the pixel segment to be sorted (sortingLength). Then the function allocates memory for the unsorted and sorted pixel arrays and fills the unsorted array with the current segment of sorted pixels. This function benefits from a 20x speed up when compared to the serial version which grows as the image size increases. After processing all the rows, the output image will have sorted darker pixel segments in each row based on their brightness. The sorting creates visually interesting patterns that can be used for artistic purposes or other image processing tasks.



CUDA shown in light blue.

Color Extraction

Color extraction is an incredibly common and useful tool identifying one feature of an image or removing a uniform background of a distinct color. The problem is the method of identifying a color and comparing it to others. Simple formulae for calculating a color's feature such as luminosity or averaging is insufficient for complex images with many gradients across its color palette. Additionally, we wanted to set transparency across vacant space after extraction.

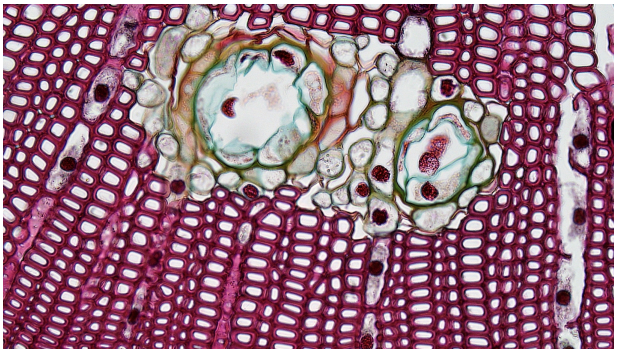
$$\bar{r} = \frac{C_{1,R}+C_{2,R}}{2}, \quad \Delta R = C_{1,R} - C_{2,R}, \quad \Delta G = C_{1,G} - C_{2,G}, \quad \Delta B = C_{1,B} - C_{2,B}$$

$$\Delta C = \sqrt{\left(2 + \frac{\bar{r}}{256}\right)\Delta R^2 + 4\Delta G^2 + \left(2 + \frac{255-\bar{r}}{256}\right)\Delta B^2}$$

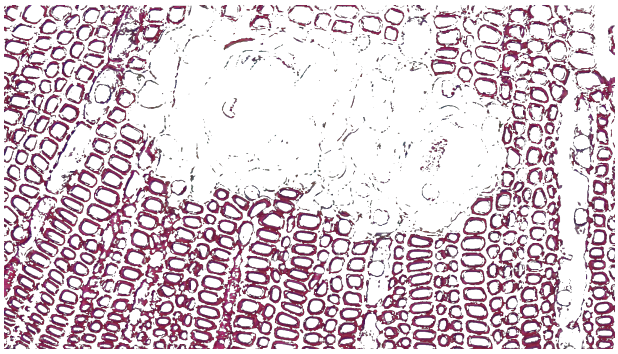
We opted for a color diff function as described by Thiadmer Riemersma of CompuPhase [5]. It uses a a weighted euclidean distance formula with a red mean and color diffs to calculate a general difference between two colors. It's an approximate calculation that superimposes a CIELUV color model onto existing RGB values.

Filtering by color wasn't enough. A lot residual information remained after screening for color, so some kind of noise filtration was necessary to create a coherent image. This is when the median filter comes into play. An image median filter is a post process effect that finds the average pixel color around a middle pixel and inserts the median of those colors into the target coordinate [6]. It's good at removing excess digital noise along with salt and pepper noise. For our purposes, it smoothed out areas of the image that required uniform transparency. The problem with this filter is it's run time. Median filtering uses a sorting technique with $\mathcal{O}(n^2)$ run time, which is terribly slow in serial.

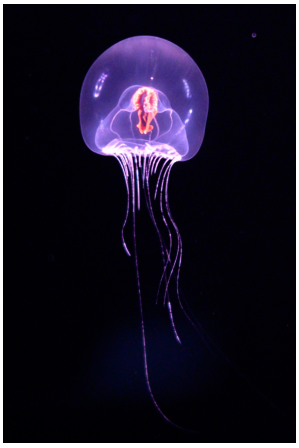
As seen in the photos of the train, the grainy image on top is run through a CUDA accelerated 5x5 median filter. The end result is substantially more coherent while taking less than 50 milliseconds to compute at approximately 720p resolution. This same process is more than 30x slower in serial. Edges have been preserved to a reasonable degree. What's particularly interesting is how the pixel order has been shifted substantially and yet a more coherent output is obtained. Color perception is deceptive. It plays a crucial role when complex filtering techniques are combined.



Tissue under a microscope



Extracted membrane structure (170 threshold)



Original

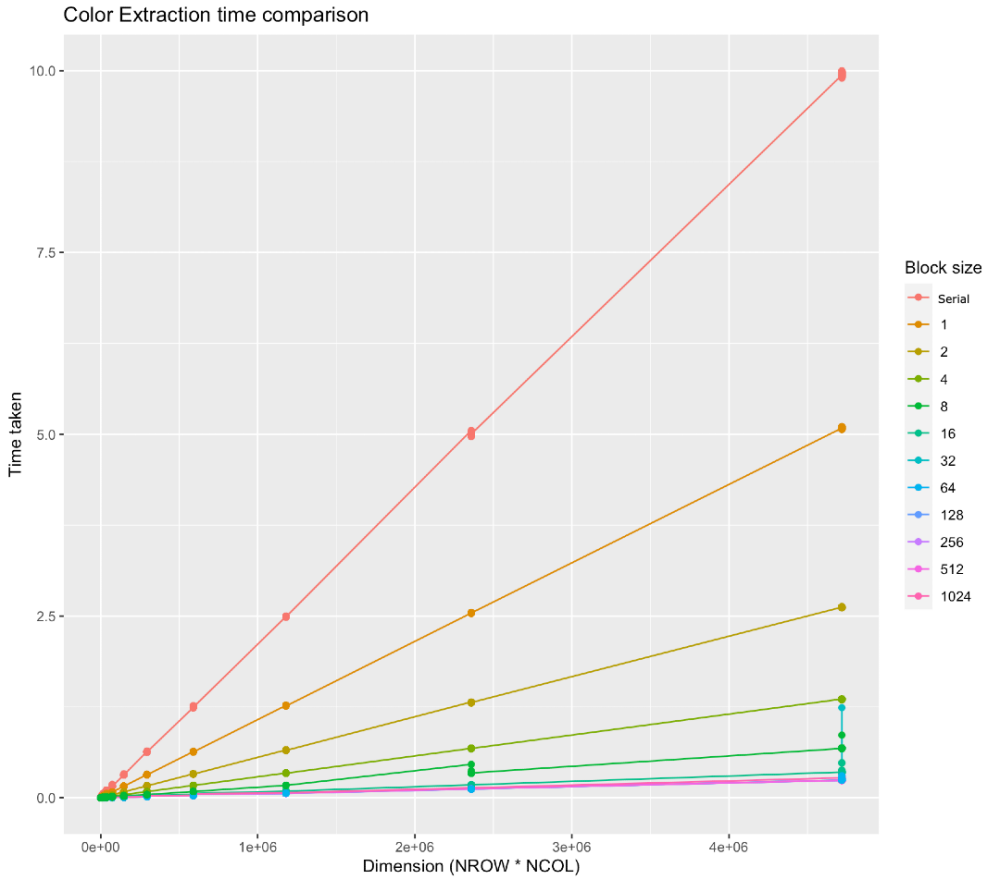
Serial (1.7s)

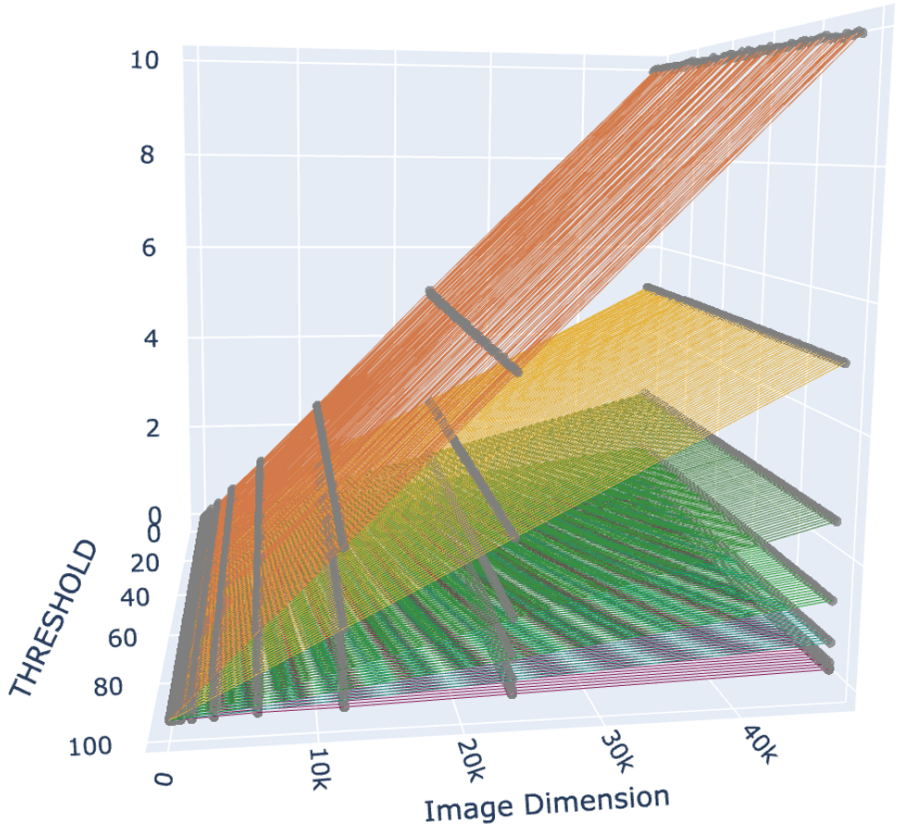
CUDA (65ms)

Diff

After color diffing and median filtering, we arrive at a visual paradox after processing an image. The extracted jellyfish from serial and CUDA removal appear the same to the human eye, yet are entirely different according to (apart from the transparent background) an ImageMagick diff . How?

The answer lies in the median filter, CUDA's non-determinism, and human perception. Since median filtering requires surrounding pixel information, each CUDA thread will make it's judgement of the median in that frame of time, with that instance of color information. Operations are asynchronous. In contrast, serial mode sorts in order and will judge differently because preceding pixels have been filtered. We argue that a CUDA implementation would be wrong (and much slower) if the diff was the same. Comparisons prove that CUDA threads are filtering independently in parallel. It explains the 25x speedup we obtained. Visually the results are comparable, so who would wait 1.7 seconds to process an image of modest resolution when the GPU does it in 65 milliseconds? Consider processing an entire HD video. It's the difference between having real-time filtration vs. a lengthy post-production edit. There is no right or wrong when removing heavy noise or applying creative effects. Color gradient shifts are minute when they are at the pixel level. These kinds of discrepancies between serial vs. GPU image processes is a theme that continues in pixel sort.


Color Extraction time comparison
Time taken
10.0
7.5
5.0
2.5
0.0
0e+00
1e+06
2e+06
3e+06
4e+06
Dimension (NROW * NCOL)
Block size
Serial
1
2
4
8
16
32
64
128
256
512
1024


THRESHOLD
0
20
40
60
80
100
0
10k
20k
30k
40k
Image Dimension
10
8
6
4
2
0

Future Work

For our future work on this project we hope to add a social media style interface as a front end. We wish to include more features, and ways for users to store and share the images they create with our project. Friends would be able to add filters and edit each other's images, while having a board where they could pin their favorites. We would also include real-time editing software so that users could test different thresholds for our various filters before selecting their favorite. We would also like to add more video functionality. Where we could edit a video one frame at a time with our existing set of features, and recompile the video at the end. We would also like to experiment with using gigapixel images to see how our code's run-time changes.

Acknowledgements

Special thanks to Dr. Lam for introducing us to CUDA and its benefits. Learning about this powerful tool has been an eye-opening experience that has greatly enhanced our understanding of parallel programming. His guidance and expertise has been instrumental in helping us understand the intricacies of this technology. Through his clear and concise explanations, we have gained a deep appreciation for CUDA and will apply this knowledge to our future projects.

Images are courtesy of Pexels. Color extraction techniques were inspired by Ian Che Te and CompuServe.

References

[1] An even easier introduction to Cuda. NVIDIA Technical Blog. (2022, August 21). Retrieved May 5, 2023, from https://developer.nvidia.com/blog/even-easier-introduction-cuda/

[2] ASDFPixelSort. (2022). PixelSorting [Code repository]. GitHub. https://github.com/kimasendorf/ASDFPixelSort

[3] Ianchute. (2018, February 22). Background removal (CIELUV color thresholding). Kaggle. Retrieved May 5, 2023, from https://www.kaggle.com/code/ianchute/background-removal-cieluv-color-thresholding/notebook

[4] Pexels. (n.d.). Retrieved May 6, 2023, from https://www.pexels.com/

[5] Riemersma, T. (2019, May 23). Colour Metric. CompuPhase. Retrieved May 5, 2023, from https://www.compuphase.com/cmetric.htm

[6] Wikimedia Foundation. (2023, April 20). Median filter. Wikipedia. Retrieved May 5, 2023, from https://en.wikipedia.org/wiki/Median_filter