Color Palette Detection

Leila Adaza, Yosmel Caleo, Alexandra Daglio

Florida International University

Abstract

This paper will describe how and why parallelizing is beneficial when extracting a color palette from an image. Presently, color palettes are found using exact red, green, and blue (RGB) mapping, which makes for a complicated process and high time complexity. The proposed solution is to compare an image's original colors to a predefined collection of commonly supported colors and then parallelizing critical tasks within the Graphics Processing Unit (GPU) using CUDA, Nvidia's parallel computing platform and programming model. Our results show that the power and benefit of parallel processing becomes more apparent with greater numbers of pixels in an image. This contribution is significant for the field of digital image processing because it would allow for faster results when analyzing image's color palettes in industry.

Introduction

Color palettes are a useful tool in capturing the message of an image in a compact way. They also simplify color relationships for color-aware tasks and provide intuitive interfaces and visualizations (Tan, Echevarria, & Gingold, 2018). The most popular method of detecting color palettes in images is RGB mapping. This means that every slight variance in red, green, or blue hues are viewed as different colors even though they may look similar to the naked eye. By using a predefined color palette with distinct RGB values, a more accurate color palette can be created from the image.

Furthermore, the proposed approach to color palette detection utilizes parallelism to produce results more efficiently. The GPU is tasked with mapping each image pixel to its closest match in the predefined color palette and keeping count for each color detected, in parallel. This list is passed back to the CPU to be sorted and the top six colors are outputted to the user. Overall, this process dramatically cuts down the running time of detecting the image's color palette.

Related Work

Some related works on color palette detection can be found on Github, a code hosting service that allows developers around the world to collaborate and build or improve software. The most prominent repository related to detecting color palettes from images is called Color Thief and utilizes JavaScript and HTML (Dhakar, 2011). Since its creation in October 2011, Lokesh Dhakar's Color Thief project has had a total of thirteen contributors. It also paved the way for different implementations of color palette retrieval, including Java, Rust, and Swift. At the time of writing, the Java port is owned by Sven Woltmann and has two additional contributors (2014); the

Rust port is owned and solely developed by Evgeniy Reizner (2017); and the Swift port is owned by Kazuki Ohara and has one additional contributor (2017). Each of these versions use the original algorithm developed by Lokesh Dhakar. Since the algorithm utilizes exact RGB mapping, none of these color palette detectors rely on a predefined color palette. These works also all lack the benefits of parallelism with GPU processing, which greatly differentiates them from the new technique.

Process

Two different methods were utilized to retrieve the color palette from images. Both of these processes are explained in detail with supporting code samples and figures below. The first approach was a linear process and the second approach was parallelized for efficiency when dealing with larger images. The processes used to retrieve the color palette were developed without the use of any structs or classes. They are simply implemented with parallel arrays. Both processes began by extracting the RGB values from each pixel in an image and then separating these values into their own, individual arrays which were of a size equal to the number of pixels in the image. In addition to the COLORNAMES array which holds the string names in the predefined color palette of 143 colors, such as “Indian Red” and “Black” Both approaches also have 3 parallel arrays, each holding the values of the R, G and B values for these 143 color codes.

Linear Approach. Simply stated, after determining the closest color to which each pixel's RGB value is most closely resembling, the FREQINDEXES array is populated with the indexes corresponding to the color names. The COUNTER array kept track of how many times a particular color index appeared. With the COUNTER array, an algorithm was designed that would find the highest six counters overall. This algorithm goes through the COUNTER array to find the highest number and records that index as the index of the most common color into a variable; it ensures that this index would never be visited again, then iterates for the next highest counter value. Figure 1 is a visualization of this approach.

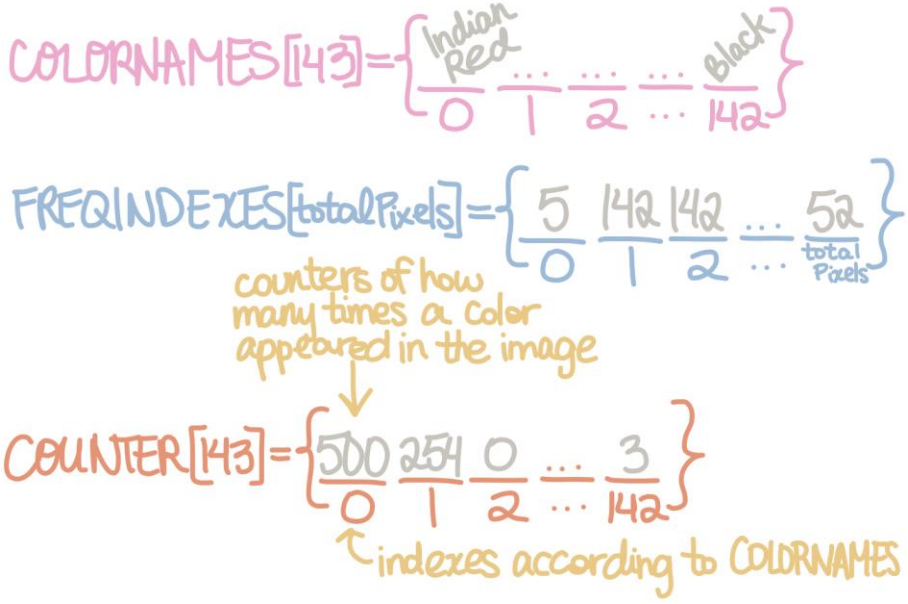

Figure 1: A visual representation of the Linear Approach that shows the parallel arrays.

Parallel Approach. Like the Linear Approach, the Parallel Approach begins by reading the pixels in the image and extracting the R, G, and B values into their own separate arrays. These arrays are then copied into the GPU along with the empty FREQINDEXES array and the number of total pixels in the image. The process of mapping each pixel to it's closest resembling color (from the predefined 143 colors) is done in parallel. Each pixel's color is determined on its own thread and the index of each pixel's color is then saved in the FREQINDEXES array. Hence, the FREQINDEXES array only holds values 0-142 representing the index of a color from the COLORNAMES array. The FREQINDEXES array is then copied back to the CPU and the COUNTER array is populated with how many times each color index appeared in the image and the most common colors are determined in the same way as they were determined in the Linear Approach.

```
__global__ void frqColorArrayBuilder(uchar* gFREQINDEXES, uchar* gpixelBArr,uchar* gpixelGArr,uchar* gpixelRArr, int totalPixels) {
    uint minIndex = 0;
    int freqIndexesSize = blockIdx.x*blockDim.x + threadIdx.x;;
    uchar avg;
    if(freqIndexesSize < totalPixels)
    {
        //initialize minAvg to the first color
        uchar minAvg = (fabsf(gpixelBArr[freqIndexesSize] - BLUE[0]) + fabsf(gpixelGArr[freqIndexesSize]) + fabsf(gpixelRArr[freqIndexesSize])) / 3;

            for (int i = 1; i < NUM_COLORS; i++) //iterate through the array of 143 color codes
            {
                //calculate the avg for the passed pixel values and the current color code (i)
                avg = (fabsf(gpixelBArr[freqIndexesSize] - BLUE[i]) + fabsf(gpixelGArr[freqIndexesSize]- GREEN[i]) + fabsf(gpixelRArr[freqIndexesSize] - RED[i])) / 3;

                //find our lowest avg
                if (avg < minAvg) //if compareNum is less than lowest min (min0)
                {
                    minIndex = i; //save the index of the color code from the COLORNAMES array
                    minAvg = avg; //save the new minAvg for future comparisons
                }
            }
            gFREQINDEXES[freqIndexesSize] = minIndex; //populate the FREQINDEXES array
    }
}
```

Figure 2: A sample of the code from Approach #2 to map each pixel using Euclidean formula.

Results

Upon testing with three trial images, it was concluded that the two approaches generate the same color palettes. The first approach, despite relying on basic data structures and lacking a sorting algorithm, proved to have a faster runtime overall.

The impact on runtime is observably affected in these results. Figure 3 shows how analysis of a smaller image's color palette can actually be determined much faster using a linear approach. This is shown in our results as a result was printed in 91.797000 ms when computed by the CPU alone in the Linear Approach as compared to the 350.416000 ms it took to print a result using the GPU in the Parallel Appraoch. Figure 4 starts to show the gains of parallelizing with a Linear Aproach result of 739.694000 ms and a Parallel Approach result of 595.089000 ms. Finally, Figure 5 further confirms that larger images are processed faster in the GPU than solely in the CPU with the Linear Approach result being reached in 8170.144000 ms and the GPU's Parallel Approach being reached in 3270.924000 ms.

Trial 1. The original image (700 × 140) is shown not to scale in Figure 3. Figures 3.1 and 3.2 show the output using Linear Approach and Parallel Approach, respectively.

Figure 3: greens.jpg 700px × 140px.

```
**********************************CPU**********************************
Now printing the color palette of this image:
1- color#67: OLIVEDRAB- 16263 pixels
2- color#32: DARKKHAKI- 16263 pixels
3- color#80: TURQUOISE- 16009 pixels
4- color#66: YELLOWGREEN- 15986 pixels
5- color#69: DARKOLIVEGREEN- 15597 pixels
6- color#61: MEDIUMSEAGREEN- 14742 pixels
CPU: 91.797000 ms
adagl001@ursula:~/Desktop/Alex/CDAProject/Project/AlexCPP 57%
```

Figure 3.1: Output of the Linear Approach showing a runtime of 91.797000 ms.

```
**********************************GPU**********************************
Now printing the color palette of this image:
1- color#67: OLIVEDRAB- 16263 pixels
2- color#32: DARKKHAKI- 16263 pixels
3- color#80: TURQUOISE- 16009 pixels
4- color#66: YELLOWGREEN- 15986 pixels
5- color#69: DARKOLIVEGREEN- 15597 pixels
6- color#61: MEDIUMSEAGREEN- 14742 pixels
GPU: 350.416000 ms
adagl001@ursula:~/Desktop/Alex/CDAProject/Project/AlexCU 40%
```

Figure 3.2: Output of the Parallel Approach showing a runtime of 350.416000 ms

Trial 2. The original image (1000 × 786) is shown not to scale in Figure 4. Figures 4.1 and 4.2 show the output using the Linear Approach and the Parallel Approach, respectively.

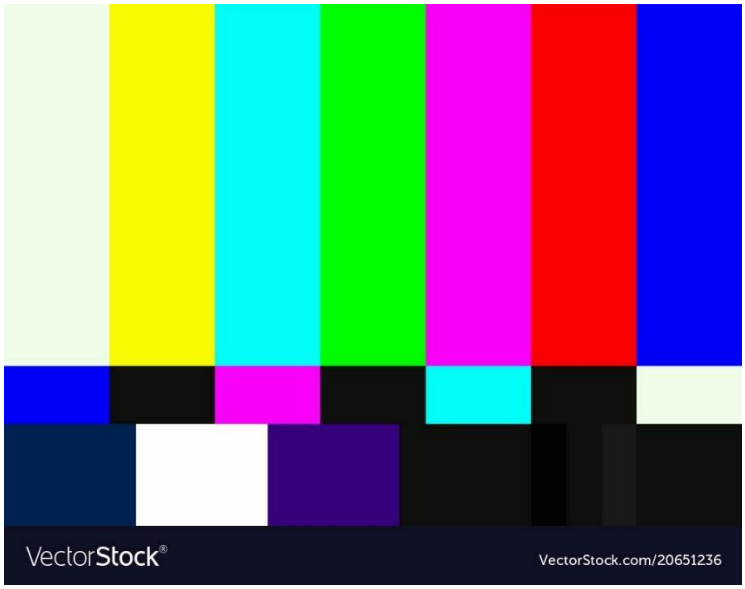

Figure 4: The tv.jpg picture clearly shows its own color palette and is a 1000px × 786px.

```
**********************************CPU**********************************
Now printing the color palette of this image:
1- color#143: BLACK- 172669 pixels
2- color#95: BLUE- 80514 pixels
3- color#75: AQUA- 80514 pixels
4- color#38: FUCHSIA- 80514 pixels
5- color#119: HONEYDEW- 79949 pixels
6- color#55: LIME- 69438 pixels
CPU: 739.694000 ms
adagl001@ursula:~/Desktop/Alex/CDAProject/Project/AlexCPP 55%
```

Figure 4.1: Output of the Linear Approach showing a runtime of 739.694000 ms.

```
**********************************GPU**********************************
Now printing the color palette of this image:
1- color#143: BLACK- 172665 pixels
2- color#95: BLUE- 80514 pixels
3- color#75: AQUA- 80514 pixels
4- color#38: FUCHSIA- 80514 pixels
5- color#119: HONEYDEW- 79949 pixels
6- color#55: LIME- 69438 pixels
GPU: 595.089000 ms
adagl001@ursula:~/Desktop/Alex/CDAProject/Project/AlexCU 46%
```

Figure 4.2: Output of the Parallel Approach showing a runtime of 595.089000 ms

Trial 3. The original image (2480 × 3508) is shown not to scale in Figure 5. Figures 5.1 and 5.2 show the output using the Linear Approach and the Parallel Approach, respectively.

Figure 5: The big.jpg picture is a 2480px × 3508px.

```
**********************************CPU**************
Now printing the color palette of this image:
1- color#117: WHITE- 6530769 pixels
2- color#23: YELLOW- 541676 pixels
3- color#143: BLACK- 530367 pixels
4- color#91: DODGERBLUE- 521497 pixels
5- color#13: DEEPPINK- 515472 pixels
6- color#72: LIGHTSEAGREEN- 9927 pixels
CPU: 8170.144000 ms
```

*Figure 5.1: O*utput of the Linear Approach showing a runtime of 8170.144000 ms

```
**********************************GPU**************
Now printing the color palette of this image:
1- color#117: WHITE- 6530769 pixels
2- color#23: YELLOW- 541676 pixels
3- color#143: BLACK- 530367 pixels
4- color#91: DODGERBLUE- 521497 pixels
5- color#13: DEEPPINK- 515472 pixels
6- color#72: LIGHTSEAGREEN- 9927 pixels
GPU: 3270.924000 ms
```

Figure 5.2: Output of the Parallel Approach showing a runtime of 3270.924000 ms

Conclusions

The results of our trials confirm that the larger the input image, the more efficient the Parallel Approach becomes in comparison to the Linear Approach. The power and benefit of parallel processing becomes more apparent with greater numbers of pixels in an image. To be more descriptive, when an image contains ~500,000 pixels or more, it is guaranteed that the GPU is faster at computing a result than the CPU.

Analyzing images in parallel can be very useful in any industry that is heavily reliant on image processing. Industries such as those in fashion, photography, gaming, video editing and others would benefit greatly from the faster processing power the GPU can offer. GPU technologies are continuing to grow in power and capabilities. NVIDIA, and their CPU-producing counterparts, are seeking to “better connect GPUs and other such coprocessors to the CPU” in order to boost the performance of this kind of processing.

References

Dhakar, L. (2011, October 30). Color Thief. Retrieved from https://github.com/lokesh/color-thief

Dixon & Moe. (n.d.). Color Names — HTML Color Codes. Retrieved from https://htmlcolorcodes.com/color-names/

DTHNews.in. (n.d.). Freedish.in Channel Image [Digital image]. Retrieved from https://www.dthnews.in/2014/12/dd-direct-plus-added-mpeg-4-hd-vacant.html

Ohara, K. (2017, February 12). ColorThiefSwift. Retrieved from https://github.com/yamoridon/ColorThiefSwift

Reizner, E. (2017, July 30). color-thief-rs. Retrieved from https://github.com/RazrFalcon/color-thief-rs

Tan, J., Echevarria, J., & Gingold, Y. (2018). Palette-based image decomposition, harmonization, and color transfer. *1*(1), 1-2. doi:10.475/123_4

Woltmann, S. (2014, August 10). Color Thief - A Fast Java Implementation. Retrieved from https://github.com/SvenWoltmann/color-thief-java

Smith, R. (2006, September 30). The GPU Advances: ATI's Stream Processing & Folding @ Home. Retrieved from https://www.anandtech.com/show/2095/3