05/04/2022

15-418 (S22) Final Project Report:

Implementing and Parallelizing Pixelated Image Abstraction using OpenMP and CUDA

Anthony Meza & Kevin Xie

1. Project Summary

We implemented an algorithm to pixelate images while emphasizing unique features to mimic pixel art, then compared using CUDA on the GPU and OpenMP on the CPU on the GHC machines to parallelize the steps taken in each iteration of the algorithm.

2. Background

“Pixelated Image Abstraction” is an algorithm published by Gerstner et al., which describes a method of combining SLIC superpixel segmentation and MCDA clustering to downsample an input image in a way which preserves its unique features. The goal of such an algorithm would be to generate an output which resembles manually made pixel art, which is a type of illustration where the artist’s palette and image fidelity is limited.

Simple Linear Iterative Clustering (SLIC) is a method for segmenting an image’s pixels into larger “superpixel” clusters within which each pixel is a certain distance both in 2D-space and color-space from the mean color of the superpixel. This makes it a quick algorithm for coarsely identifying the edges present in a given region of an image. Using SLIC, we are able to generate the “superpixel” regions which map to each pixel in our output image, which carry more contextual information than simple cubic or nearest neighbor downsampling.

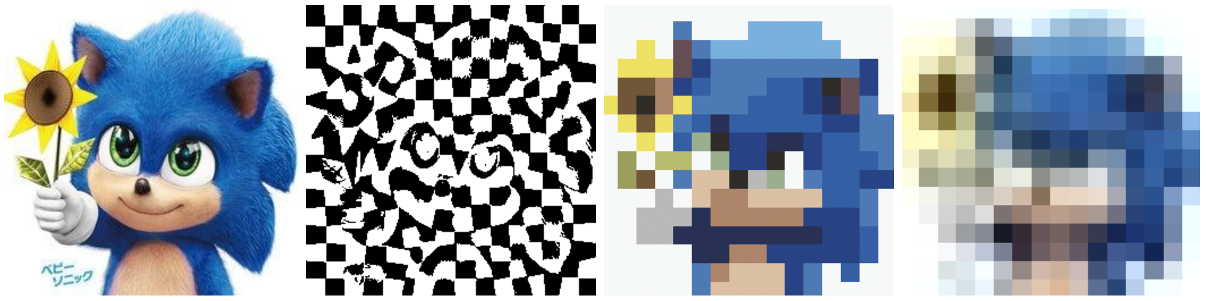


Fig. 1 *An (a) input image and its (b) associated 16x16 superpixels used to generate (c) output, when compared to (d) bilinear sampling to downsample to the same resolution*

$$d(p_i, p_s) = d_c(p_i, p_s) + m\sqrt{\frac{N}{M}}d_p(p_i, p_s)$$

SLIC "distance" between two pixels: d_c represents distance in the CIELAB color space and d_p represents positional distance in coordinates. N = number of input pixels, M = number of output pixels, m = relative weight of color similarity vs. pixel adjacency [0-20]

Mass Constrained Deterministic Annealing (MCDA) is a probabilistic, fuzzy clustering algorithm used in unsupervised machine learning to split data points into clusters along their feature dimensions. This algorithm mimics the physical process of annealing by instantiating a "temperature" and two subcluster boundaries for each cluster which is gradually refined as the temperature cools. When subclusters reach a certain threshold of convergence (error between data points associated with a given cluster and the cluster center), they separate into two clusters, each with their own subclusters. This continues from a single cluster until the desired number of clusters is reached. In this case, we are using MCDA to produce the resulting image's palette in parallel with the segmentation process.

Combining the two algorithms with a few modifications, Gerstner et al. provide the following pseudocode, which we have adapted into a flow diagram during our implementation process. For more details, see the original paper here.

Algorithm 1

```
▷ initialize superpixels, palette and temperature $T$ (Section 4.1)
▷ while ($T > T_f$)
    ▷ refine superpixels with 1 step of modified SLIC (Section 4.2)
    ▷ associate superpixels to colors in the palette (Section 4.3)
    ▷ refine colors in the palette (Section 4.3)
    ▷ if (palette converged)
        ▷ reduce temperature $T = \alpha T$
        ▷ expand palette (Section 4.3)
▷ post-process (Section 4.4)
```

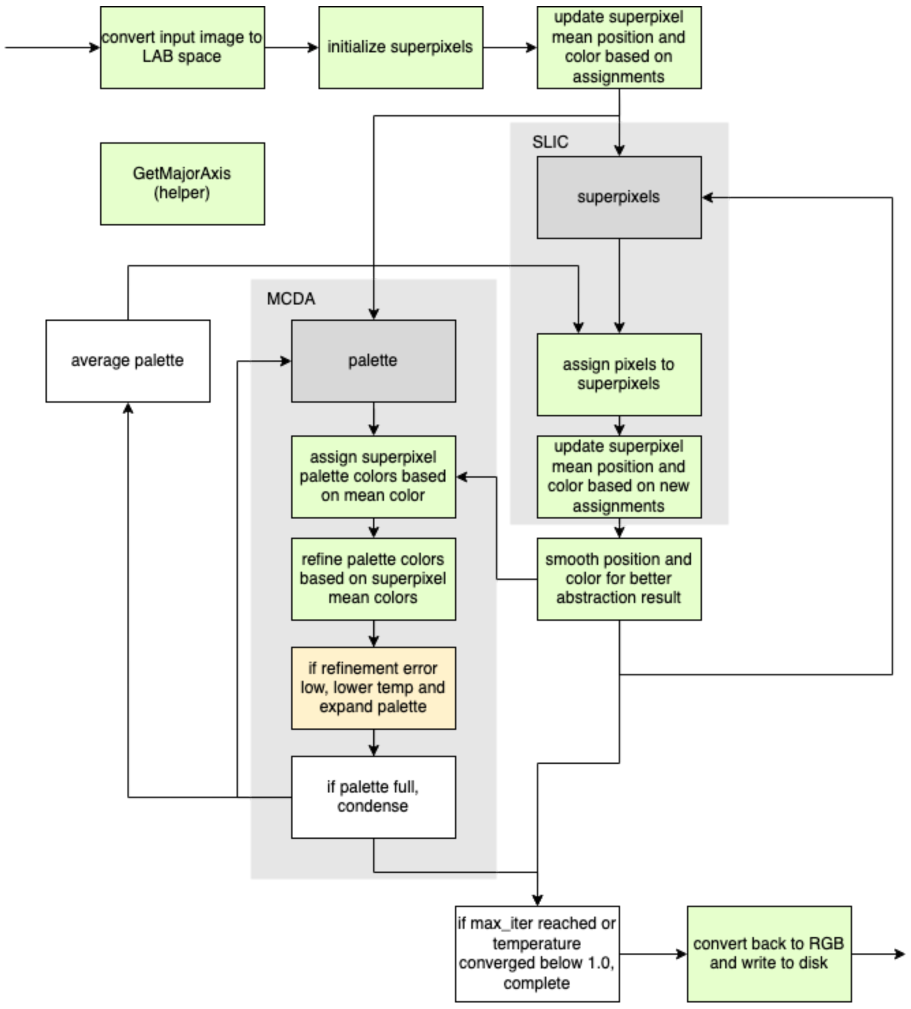

convert input image to LAB space
initialize superpixels
update superpixel mean position and color based on assignments
GetMajorAxis (helper)
SLIC
superpixels
MCDA
average palette
palette
assign pixels to superpixels
assign superpixel palette colors based on mean color
update superpixel mean position and color based on new assignments
refine palette colors based on superpixel mean colors
smooth position and color for better abstraction result
if refinement error low, lower temp and expand palette
if palette full, condense
if max_iter reached or temperature converged below 1.0, complete
convert back to RGB and write to disk

Fig. 2 *(a) Pixelated Image Abstraction algorithm and (b) flow diagram*

Data Structures

There are two main data structures being updated during each iteration, each of whose elements have a certain amount of associated metadata used to compute the next iteration. The two data structures are:

1. **Superpixels**

   Superpixels are stored as an array of superpixel centers, a 2D-array region map mapping each pixel in the input image to a superpixel index, and two arrays containing color information (one being the mean color of the superpixel and the other being the palette color closest to that mean color)

2. **Palette**

   The palette is stored as a pseudo-stack, using an array and an integer which keeps track of the next empty position. The palette stores both subclusters (represented as LAB colors) associated with a given cluster (the average of the two subclusters), and pairing information is stored as a tuple in a separate stack. Since the palette only grows, we only ever needed to “push” elements until our palette is finalized.

In addition to these two main structures, there are several arrays containing probabilities associated with color clusters and superpixels used in the MCDA association process. There are also input and output image arrays which simply store RGB values.

Areas for Optimization

SLIC can be a reasonably data-parallel algorithm: per-iteration, each pixel’s new cluster is assigned based on minimum “distance” to a given superpixel’s center as computed in the last iteration. This can be done independently of other pixels in the image and can be tiled to increase locality of data accesses. However, since minimization introduces some branching conditionals, it may not be well suited toward SIMD if we want to maximize utilization. Then, cluster centers and colors are updated based on the mean values of all of the pixels assigned to the given cluster. Again, this can be achieved either through per-cluster parallelism or per-pixel using reduction. It is important to note that superpixel regions will typically not be rectangular, which does not lend itself well to locality if traversing through a given region.

When combined with MCDA, the palette color assignment and refinement steps introduce data dependencies. Namely, each time a palette color is assigned to a superpixel, probabilities are modified globally. Since palettes will typically be much smaller relative to the size of the output image, MCDA represents less of the computational workload than SLIC.

3. Approach and Results

Serial Implementation

The sequential implementation was written in C++ using the single-file public domain image library ("stb_image.h") to read and write images. Originally, we wanted to approach the project using libraries which enable fast prototyping such as OpenCV or domain-specific languages like Halide. However, because of difficulties installing the required libraries on the GHC machines and the overhead of learning new technologies, we decided to write our own image functions as needed, including RGB-XYZ-CIELAB conversion and a closed-form solution for 3x3 eigendecomposition.

After completing our serial implementation, we were able to analyze some baseline timing metrics and how the speed up varied based on changes to the input and output image size and desired palette size for our output image, found in Figure 3.

Image Used	Input Dimensions	Output Dimensions	Color Count	Computation Time (ms)
SonicFlower.jpeg	256x256	32x32	32	649
		32x32	100	1053
		64x64	32	1579
		64x64	100	3814
BabyMeerkat.jpeg	694x1000	32x64	32	3210
		32x64	100	4135
		64x128	32	4738
		64x128	100	6727

Fig 3.

OpenMP

To begin the parallelization process, we decided to prototype rough solutions with OpenMP to identify potential vectors for later parallelization on the GPU. We started by identifying steps in the algorithm which we were able to intuitively reason could be parallelized. The clear data parallel steps were converting between RGB and LAB when reading input and rendering output. This caused a decent amount of speedup in the initialization phase.

Then, we highlighted each phase in our flow diagram we expected that we would be able to parallelize. We highlighted the phases we were confident could be parallelized green and phases we thought could be partially parallelized in yellow (Figure 1(b)).

As noted in Section 2, the SLIC algorithm is fairly data parallel, allowing for simple parallelization for the most part. One thing we changed to our algorithm was how we assigned pixels to superpixels. Gerstner et al.'s approach loops through each superpixel then searches and updates pixels in the surrounding region to minimize for "distance". While this is faster serially, it introduces data dependencies and race conditions when parallelized, since different superpixels could be reading and writing to the same pixel to minimize distance, which is not an atomic operation. To minimize synchronization cost, we decided to change the order of the loops such that the algorithm loops over each pixel then searches its surrounding region for superpixels, then minimizes its own superpixel assignment. This ensures that each pixel only reads shared data, writing only to its own address in the region map array.

Since we were only prototyping with OpenMP, we decided not to spend too much time pushing its limitations. The two main challenges we found were that the version of OpenMP available on the GHC machines does not offer the ability to reduce over array values or struct members, and that synchronization through atomic/critical sections was overly costly when used to update shared data structures containing probabilities for MCDA.

Even with these challenges, we were still able to implement some amount of parallelism for our algorithm using the strategies described above (Figure 4). Given what we learned we proceed to employ these strategies in our CUDA implementation as well.

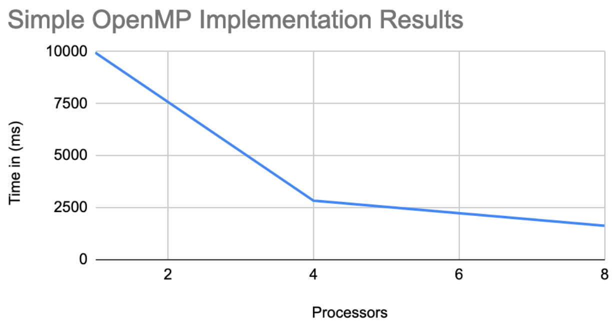


Fig 4.

After some preliminary testing we were able to achieve 3.51x speedup at 4 OpenMP threads and 6.11x speedup at 8 threads on the GHC machines. The main limitations to our parallelization were data dependencies during the MCDA palette clustering phases and the high cost of synchronization in OpenMP.

CUDA

Using CUDA to parallelize our serial implementation, we quickly realized the advantages and disadvantages of CUDA vs. OpenMP. CUDA's greatest disadvantage is that it requires a ground-up refactoring of the code and data structures into kernels, inlines, and device memory, whereas OpenMP requires minimal restructuring of the existing implementation (for a naive parallelization). However, CUDA offered a lot more flexibility and granularity over shared resources.

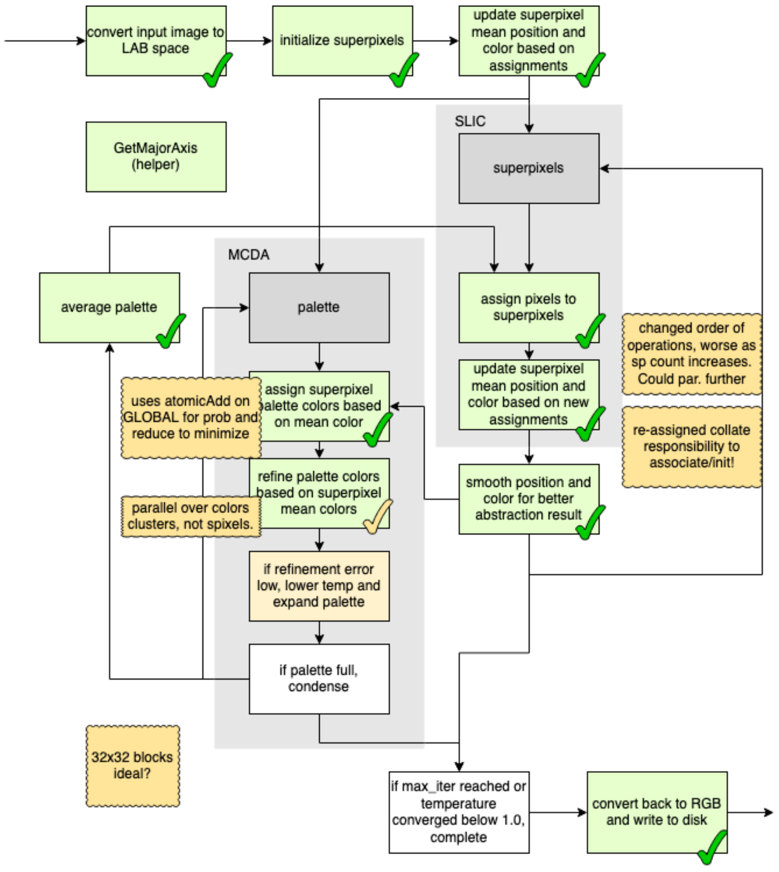
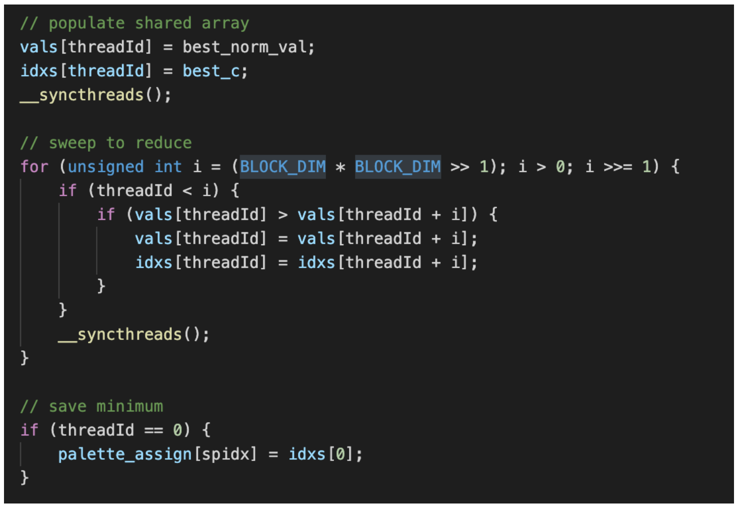

convert input image to LAB space
initialize superpixels
update superpixel mean position and color based on assignments
GetMajorAxis (helper)
SLIC
superpixels
MCDA
average palette
palette
assign pixels to superpixels
changed order of operations, worse as sp count increases. Could par. further
uses atomicAdd on GLOBAL for prob and reduce to minimize
assign superpixel palette colors based on mean color
update superpixel mean position and color based on new assignments
re-assigned collate responsibility to associate/init!
parallel over colors clusters, not spixels.
refine palette colors based on superpixel mean colors
smooth position and color for better abstraction result
if refinement error low, lower temp and expand palette
if palette full, condense
32x32 blocks ideal?
if max_iter reached or temperature converged below 1.0, complete
convert back to RGB and write to disk

Fig 5. *Green ticks represent phases successfully parallelized and yellow means suboptimally parallelized, respectively.*

Using our flow diagram, we were able to structure our CUDA implementation by creating kernels for each specific section we desired to parallelize, which can be visualized with Figure 5. As with OpenMP, we started by parallelizing the most data parallel tasks: color space conversion, superpixel initialization, and output rendering. Then, we worked through each phase in the algorithmic loop. For the SLIC phases, we were able to parallelize superpixel assignment in a similar fashion to our approach in OpenMP, by changing the order of the inner loop and the outer loop to reduce synchronization overhead. Then, for computing superpixel mean position and colors, we decided to move the responsibility of summation of superpixel values (color, position, count) to the prior step, since it was already parallelizing over pixels, whereas the mean was being calculated over superpixels. By passing the resulting counts as global variables between two kernels with different parallelization vectors, we were able to ensure that each step of the mean-computing phase was parallelized optimally.

Another technique we implemented from the course material was parallel reduction. When selecting the best palette color for each superpixel, the algorithm loops through the palette to select the color with the minimum difference to the superpixel's mean color. We initially attempted to parallelize this phase over the superpixels, such that each superpixel could loop through the palette in parallel. However, we found that as the palette size increased, the MCDA-related phases of the algorithm bottlenecked the computation time. To tackle this, we tried first to use atomicMin, which became complicated as each new minimum value would also require us to store multiple different probabilities, which still caused race conditions. Ultimately, we decided to use parallel reduce to provide a log(N) speed up, where N represents the number of colors being looped through.


```
// populate shared array
vals[threadId] = best_norm_val;
idxs[threadId] = best_c;
__syncthreads();

// sweep to reduce
for (unsigned int i = (BLOCK_DIM * BLOCK_DIM >> 1); i > 0; i >>= 1) {
    if (threadId < i) {
        if (vals[threadId] > vals[threadId + i]) {
            vals[threadId] = vals[threadId + i];
            idxs[threadId] = idxs[threadId + i];
        }
    }
    __syncthreads();
}

// save minimum
if (threadId == 0) {
    palette_assign[spidx] = idxs[0];
}
```

Fig 6. *Using parallel reduce to sweep through an array of local minimums and obtain the value and index of the block's minimum at vals[0] and idxs[0], respectively*

In multiple instances (initialize superpixels, assign superpixels, assign palette colors), we made use of atomicAdd to sum values over data parallel phases which would later be used to compute averages or check cumulative errors. We were cautious to use atomic operations as we expected it to incur heavy costs for synchronization. However, because the atomic operations were generally being performed on different addresses at any given time, we did not see any heavy penalties in computation time.

One parallelization we weren't able to finish was for refining the palette, which consists of two nested loops, an outer loop over the color clusters and an inner loop over the superpixels. While we were able to parallelize over clusters, the phase could be further parallelized by using a reduction to sum over superpixels rather than looping over them serially. Luckily, this was the lowest priority to be parallelized since it was measured to take a relatively low amount of computation time.

CUDA Results

After the completion of our implementation, we were able to see significant and consistent positive results. In order to measure how well our optimization techniques were working, we tested with two different image files "SonicFlower.jpeg" (Figure 1(a)) and "BabyMeerkat.jpeg" (Appendix B). Our overall algorithm execution times were compared to the same versions tested for our CPU sequential implementation, where the output image dimensions and desired color palette were modified to test our algorithms efficiency.

Image Used	Input Dimensions	Output Dimensions	Color Count	Speed up (CPU vs GPU speed up)
SonicFlower.jpeg	256x256	32x32	32	2.15
		32x32	100	2.44
		64x64	32	1.62
		64x64	100	2.48
BabyMeerkat.jpeg	694x1000	32x64	32	4.2
		32x64	100	4.57
		64x128	32	1.59
		64x128	100	1.89

Fig 7. *Table of speedup using different benchmark images, superpixel counts, and color counts.*

As can be observed in Figure 7, we were able to see speedups for a variety of different image inputs given to our algorithm. One observation was a clear correlation in speed up when changing from a smaller color size to a larger color size. We can observe that all (with exception of SonicFlower.jpeg output 64x64) has an approximate increase in speed up of .3 when increasing from 32 to 100 colors in our output palette. Although prominent, compared to overall speedup, this is minimal since palette related data structures were generally much smaller than pixel counts.

A notable amount of speedup could be found for BabyMeerkat.jpeg of output size 32x64, which was around x4 faster, compared to the other images which averaged around x2. This showed us how a larger image was able to increase speedup, due to the greater number of pixels available to be parallelized over. Unfortunately, as visible for both images, whenever we increase the number of superpixels present, the speedup involved decreases, attributed to the overhead that comes from pixels communicating to a greater amount of pixels.

One factor to note is that the desired palette size is reached by expanding the MCDA cluster through sequential iteration, so while increasing the palette size will increase the time taken for a given computation, the computation time per iteration remains roughly the same. Likewise, most operations parallelized over input pixels are highly data-parallel, such that the size of an image doesn’t necessarily dictate speedup. However, the superpixel count is a parallelization vector in both SLIC and clustering. Increasing the number of superpixels means there will be more superpixels to search through during the palette refinement phase, thus greater superpixel counts may bottleneck speedup.

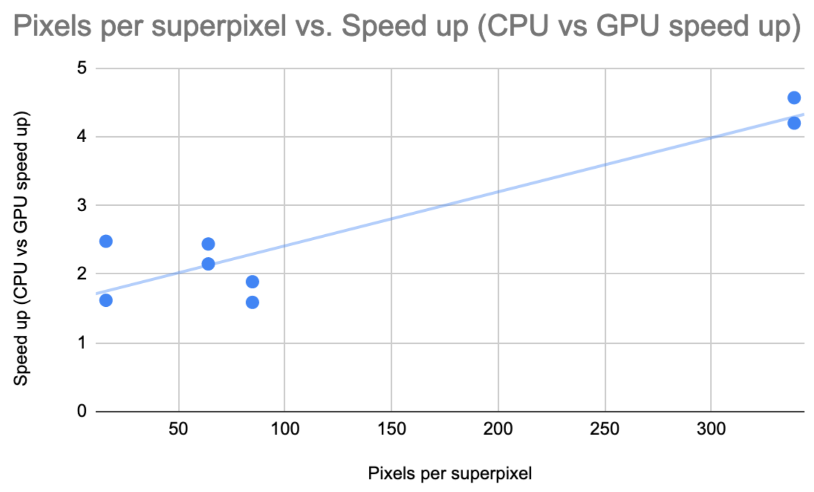


Fig 8. *Speed up increases as pixels per superpixel increases. Note that more superpixels means fewer pixels per superpixel.*

The main limiting factor for our speed up is the overhead involved with reading and writing to the device global parameters and the synchronization required to transition between our kernels. Since every portion of the algorithm relied on each other, but each part needed to be parallelized in isolation, we were forced to implement various global buffers to store data that was passed between steps of the algorithm. To minimize this, most of the buffers were only allocated on the Device, allowing us to avoid as much unnecessary communication between the CPU and GPU as possible. In addition to this, multiple kernels used throughout our algorithm created the greatest amount of overhead, since our threads needed to be constantly synced between each iteration. If there had been a way to parallelize each portion of the algorithm together, using multiple kernels rather than smaller kernels, we would have achieved a greater amount of speedup by avoiding the need for a lot of overhead from synchronization.

4. References

1. **Gerstner et al.**, “Pixelated Image Abstraction.” *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering* (NPAR), June 2012.
   https://gfx.cs.princeton.edu/pubs/Gerstner_2012_PIA/Gerstner_2012_PIA_full.pdf
   https://cragl.cs.gmu.edu/pixelate/

2. **Rose et al.,** “Mass-Constrained Deterministic Annealing”
   https://authors.library.caltech.edu/78353/1/00170767.pdf

3. **Achanta et al.,** “SLIC Superpixels”
   https://www.iro.umontreal.ca/~mignotte/IFT6150/Articles/SLIC_Superpixels.pdf

5. Appendix

A. LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT

Anthony (45%):

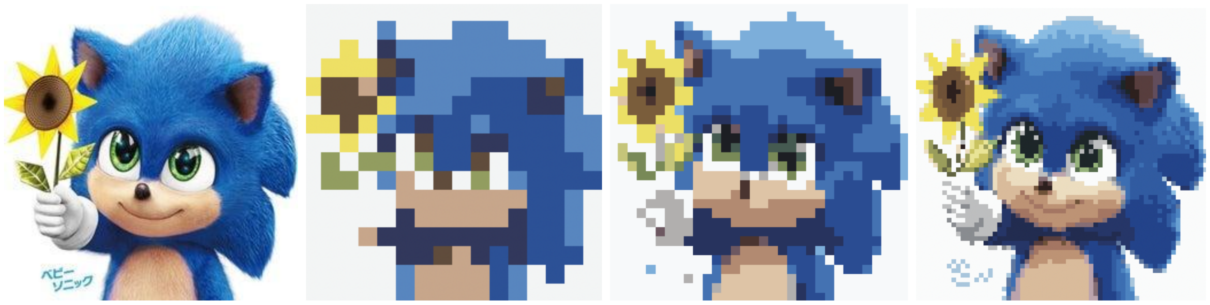
- Project Setup
- STBI Image read/write library
- Style and class refactoring
- MCDA clustering (30%)
- CUDA refactoring
- CUDA parallelization (20%)
- Timing and performance benchmarking

Kevin (55%):

- Color conversion
- SLIC Superpixel segmentation
- MCDA clustering (70%)
- OpenMP prototyping
- CUDA parallelization (80%)
- Documentation

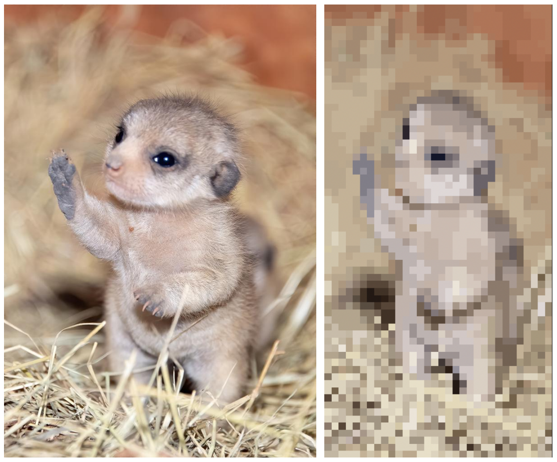
B. Image Results

SonicFlower.jpeg was the go-to benchmark when testing. Our final implementation was able to achieve sub-second computation of each of the following output images.



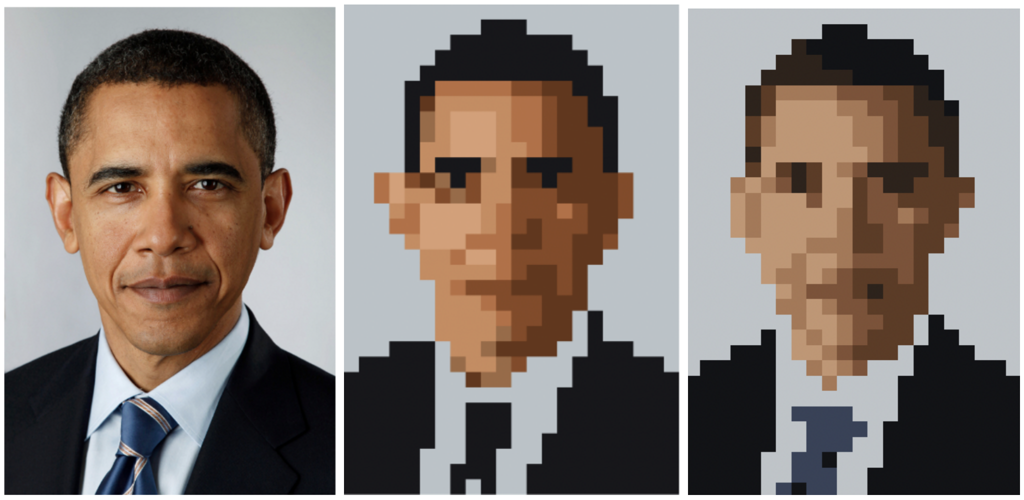
(a) Input image, (b) 16 x 16 (8 colors), (c) 32 x 32 (16 colors), (d) 64 x 64 (32 colors)

Another benchmark image used was BabyMeerkat. We decided to look for an image with uneven dimensions and a less dynamic color palette to ensure that we were covering edge cases when parallelizing.



(a) Input image, (b) 32 x 64 (100 colors)

Finally, we also made sure to test our algorithm out on the example image used by Gerstner et al. in their paper to demonstrate Pixelated Image Abstraction.



(a) Input image, (b) Gerstner et al., (c) Our result. Note: Gerstner et al. decided to increase the saturation of the resulting image as a final step, where we did not.