K-Means Parallelization
Final Project Report

Brian Dunlay
EE590A - Winter 2017

March 8, 2017

Overview

K-Means Clustering is vector quantization algorithm. Given a set of k centroids (or initial values), it groups similar data points together by associating each data point with its most similar centroid. Centroids are moved by taking the average value of the cluster, and this process is repeated until convergence. I have applied this algorithm with image processing with the goal of segmenting an images by color.

Given an integer k and a set of n data points $X \subset \mathbb{R}^d$ we aim to choose k centers C so as to minimize the following function [2]:

$$\phi = \sum_{x \in X} min_{c \in C} \|x - c\|^2$$

0.1 Iterative Algorithm

1. Choose k centroids $C = c_1, c_2, \cdots, c_k$
2. For each $i \in 1, \cdots, k$, set the cluster C_i to be the set of points in X that are closer to c_i than they are to c_j for all $j \neq i$
3. For each $i \in 1, \cdots, k$, set c_i to be the center of mass of all points in C_i: $c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$
4. Repeat Steps 2 and 3 until C no longer changes

0.2 Implementation

In order to run the program, a set of command line arguments must be provided.

1. Input file path (bmp)
2. Output file path
3. 2 or more x,y pixel coordinates (value taken from each as initial centroids)

The BMP format is preferred for this project due to its simplicity. The header contains the dimensions of the image, and the pixels are stored as three bytes of Blue, Green, and Red in that order. [**?**]

The BMP file is read into memory and OpenCL Buffers are allocated. In order to simplify the kernel processing, I extracted the image data from the bmp file in memory into a linear buffer.

BMP format specifies that each line of the image is stored with 4-byte alignment, so care was taken to avoid padding data when copying image bytes between buffers.

The RGB data is then converted into YUV colorspace as a pre-processing step.

In my design specification, I calculated the AI of the kernel to be $\frac{K3D+K}{KD+D+1}$ where K is the number of centroids, D is dimension and D is the colorspace dimension. RGB is three dimensional while a commonly used colorspace YUV is two dimensional if we ignore the Y (luma) component. Instead of computing the distance between pixels and centroids in three dimensional space, the distance can be computed using two the two-dimensional space of YUV, reducing the computational complexity slightly. The *Big O* complexity is examined further in this paper.

There are other reasons for choosing YUV colorspace, but I will not elaborate for this project.

Interestingly, regardless of the dimension chosen, the AI remains nearly the same. Computing the AI with $K = 5$ and $D = 3$ (and using Floats) yields .3448. The AI is very similar when reducing the dimensionality to $D = 2$, yielding .3608

Also notable, as K grows, the AI only increases slightly as the the numerator and denominator cancel one another out (plateauing around about .437)

After the buffer is converted to YUV colorspace, the centroid values are cached and the kernel executes with a buffer of K centroids and a buffer of image data. Every kernel operates on a single pixel, where the distance between the pixel and each centroid is calculated. A third buffer of MxN is used as a one-to-one centroid-assignment. Each index of the centroid-assignment buffer corresponds with a pixel in the buffer, and the minimum distance centroid index is assigned to its respective position.

Back on the host, the centroid-assignment buffer is iterated over and centroid averages are calculated. These averages are compared against the cache, and if they do not match, the kernel is run again. This repeats until convergence.

1 Results

1.1 Sequential

I wrote a sequential reference in C++ in order to achieve a baseline prior to implemeting my parallelized code using OpenCL and a GPU.

It should be noted that the sequential code was not written using OpenCL. Instead, standard C++ was used along with LLVM clang on a fairly standard CPU (Intel i3 circa 2013). I suspect that had it been written using OpenCL vector data types and using OpenCL APIs (e.g. *distance(floatn, floatn)*, there would have been a significant improvement due to, at minimum, better SIMD exploitation. Regardless, it serves as a good baseline for how an iterative algorithm might perform on standard hardware using typical tools without parallelized optimization techniques.

I did not time the colorspace conversions given the fact that it is not relevant to the immediate problem (K-Means clustering), however the conversion could be parallelized.

For a visual indication of the result of the k-means clustering, I colorized the clusters based upon their mean color value along with a constant luma value. This was also omitted from the timing of the runtime.

This image was segmented based on 5 coordinates that were hand-picked as input to the program. The average runtime over 30 runs for processing this image was *3.026 seconds.*

1.2 Parallel

Again using 5 hand-picked coordinates as inputs, I ran the program 30 times and recorded an average of *105.23 ms*

1.3 Sample Images

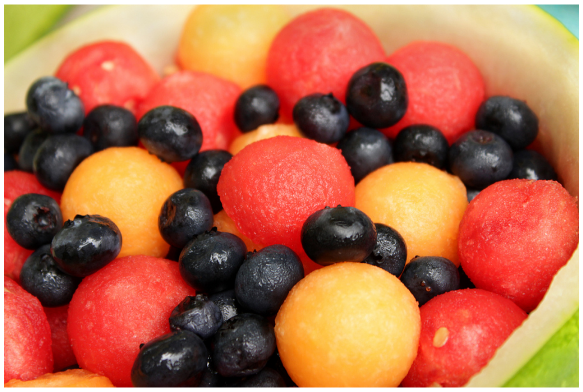

Figure 1: Image[4] before segmentation

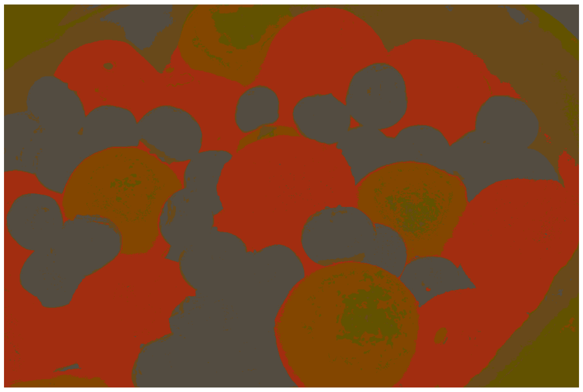

Figure 2: Image after segmentation (CPU)

2 Analysis

2.1 Kernel Analysis

I ran the kernel with random data using the Visual Studio IDE's OpenCL Session Analyser to get an idea of kernel performance. Using random "image data" of 1024x800 and a K of 5, I saw fairly good results.

- Kernel EU utilization: *86.8%*.

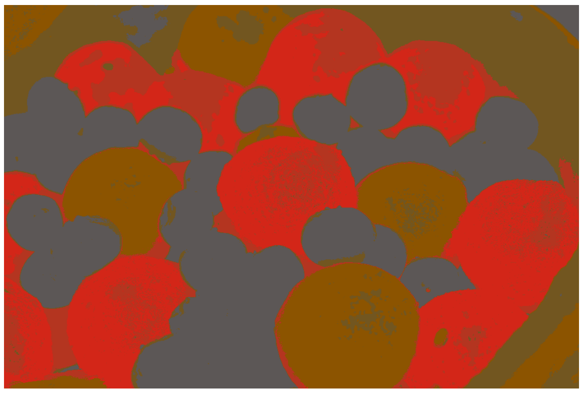

Figure 3: Image after segmentation (GPU)

- Kernel on GPU: *0.76 ms* execution time.
- Kernel on CPU: *2.5 ms* execution time.

My most latent operations were in accessing memory. I optimized this by accessing the pixel value only once rather than per distance calculation.

I took a big performance hit by implementing the algorithm as a split CPU/GPU task. Looking at individual actual run analysis results, the individual kernel execution times had an average duration of *670.775 us* and a total duration of *10061.625 us*. The iterative calculations for mean centroid values and the memory transfer between CPU and GPU for each iteration accounted for the large majority of the runtime.

2.2 Algorithmic Complexity

There are two primary stages of this algorithm, and I will analyize each of them individually. It should be noted that sinces the algorithm is iterative until convergence is reached, there is a unknown factor in which we will recalculate the following two steps. In most cases it should be much less than N.

2.2.1 Clustering

The first stage is clustering each pixel in the image with an associated group. For each pixel x_i in the image, we compute the euclidean distance between x_i and each centroid c_i with dimensionality D.

O(NKD)

Where N is the number of pixels in the image and K is the number of centroids. As previously mentioned, there is a slight reduction in the algorithmic complexity by reducing the dimensionality of the colorspace from 3 (RGB) to 2 (U and V of YUV).

D could technically be omitted given that it is a constant that is likely smaller than K and likely significantly smaller than N.

2.2.2 Mean centroid calculation

The second stage is to calculate the average value for all pixels in a particular cluster $c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$. The algorithmic complexity is: $\mathrm{O}\big(ND\big)$ where we are calculating a summation of D values over N data points.

2.3 Arithmetic Intensity

The arithmetic intensity for the clustering step is K iterations of a (1) the difference between two centroids (2) squared and (3) summed over D dimensions, followed by a (4) square root. FLOPS $= K(3D + 1)$.

The data access of the clustering step is $KD+D$ reads (where one byte is read per dimension) and the 1 write (the cluster id) for a total of $KD+D+1$.

The AI is therefore $\frac{K3D+K}{KD+D+1}$

3 Conclusion

The problem itself, despite being a fairly simple algorithm, was clearly a good candidate for parallelization. The data independence of all pixels made excellent use of the EUs when enqueued to the GPU, and out-of-order access was limited (unlike what you might see in matrix multiplication, for example).

While my implementation did see quite an improvement from the sequential implementation, there is still plenty of room for improvement. A good place to start would be in storing the pixel values for each centroid assignment in a respective centroid buffer – perhaps using atomic index increments to synchronize data writes. Then, device-side enqueueing could be used to perform a reduction on each of those buffers to calculate the mean centroid values, until convergence.

References

[1] Multiple contributors, Accessed 3/8/2017, *BMP file format*, https://en.wikipedia.org/wiki/BMP_file_format

[2] David Arthur and Sergei Vassilvitskii, *k-means++: The Advantages of Careful Seeding*, http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf

[3] J. Sirotkovi, H.Dujmi, V. Papi *K-Means Image Segmentation on Massively Parallel GPU Architecture*

[4] Didriks, *Fruit Salad*, https://flic.kr/p/a6W1Te

[5] Richard Szeliski, *Computer Vision: Algorithms and Applications*, http://szeliski.org/Book/drafts/SzeliskiBook_20100903_draft.pdf