Parallel Image Processing Toolbox

Shreya Bali (sbali), Kusha Maharshi (kmaharsh)

December 14, 2020

1 Summary

The goal of this project is to implement and analyze the scalability, scope, and benefit of performing image segmentation in parallel. In particular, we implemented the sequential versions of K-Means, Otsu Binarization, and Edge detection algorithms in C++ and their corresponding parallel versions using CUDA and OpenMP. Our project website is https://kmeshx.github.io/ and the most recent iteration of our code has been submitted to Autolab.

2 Background

Image analysis is a growing area of research in both the machine learning and the systems sub-fields. It has numerous applications such as object detection, graphic processing, three dimensional reconstruction etc. In machine learning, we often need to pre-process images so that the algorithm can appropriately learn the 'right' features for the task. One of the common ways to do this is using image segmentation - a method to divide the image into different parts. This is an overall challenging task as different images may have vastly different features, and thus such an algorithm should be broad scoped enough to work for such different images, yet specific enough to identify the required features of the different images.

Considering the growing need of image segmentation, as well as the growth of machine learning on large datasets, it becomes imperative to have algorithms that are able to compute the results quickly. This is also beneficial in cases where the evaluation needs to be in real time. A simple way to do this would be to construct simple algorithms that don't require much computation. However, this would severely effect the results. A natural alternative is thus to make existing algorithms faster using better computing methods - such as processing them in parallel. Our project aims to do the latter.

Image segmentation techniques can be broadly classified into three categories - thresholding, edge detection and clustering. In this project we have implemented one algorithm from each of these main categories, and parallelized them using CUDA and OpenMP. Different parallelization strategies are used for each of these algorithms in order to obtain the best speedups. While these algorithms have the same common goal of image segmentation, their respective details are very different, and each one thus requires carefully testing strategies to get the optimal solution.

Due to the specialized nature of each of the algorithms, this paper discusses the algorithm, methodology and results for each of them separately. Before these discussions, we also have a common methodology section for the three algorithms that discusses the common experiment strategy used across all three algorithms.

These algorithms have different characteristics, and we use this opportunity to study different parallelization techniques tailored to work for different conditions, using concepts learnt in 15418. For example, the main purpose of Otsu binarization is to check how parallelization works in low arithmetic intensity conditions; for edge detection, it is to measure the effect of parallelizing convolutions, and for K-Means, it is to test different parallelization strategies for high work-load sequentially iterated algorithm. This work can then be used to create benchmarks in similar problems that match with one or more of these characteristics, and be extended to developing optimal parallel algorithms for image processing problems that might require parallelization over multiple sub-parts that match different characteristics (e.g. the

Figure 1: Images Used for Experiments. Top left: Small(481×321), Top Right: Medium(700×700), Bottom: Large(1920×1080)

low arithmetic intensity parts can be optimized using strategies mentioned in Otus, while a part that requires Clustering could use the strategies mentioned in K-Means). The use of different algorithms with different characteristics helped us focus on each of these parts separately, and optimize accordingly. Common image processing tasks use a pipeline of such methods to pre-process the image into a state that is usable - for example using K-Means to reduce the number of colors in the image, using Edge detection to find the edges in this new image, and then finally an Otsu Binarization to decide which edges to keep and which to remove. Hence, while we present the different algorithms in different sections, these should be regarded as sub-parts of an image processing pipeline rather than a grouping of mini projects.

3 Procedure for Experiments

The experiments for OpenMP were conducted on the Bridges cluster while those for CUDA were conducted on the GHC machines. We implemented the sequential versions of the algorithms in C++. It first must be noted that there are two natural ways of parallelizing the code-(easy) parallelizing over images, (harder) parallelizing different parallelizable steps of the algorithms. In this paper, we focus on the latter. However, we also did implement a parallelization over the images for the OTSU algorithm for reference. Similar parallelizations can be extended to the others. The others assume that the input just consists of one image, and analyzes different techniques to parallelize these algorithms. Each of the algorithms are tested on different size images(labelled Small, Medium and Large according to their sizes) to observe the scalability of parallelization over differing sizes. The three images along with their sizes are shown in Figure 1. Due to the experiments being conducted on different machines, we can't create an absolute analysis between the two implementations, but can compare their trends to the respective sequential versions.

4 Otsu Binarization

One of the easiest ways to create a image segmentation is to set a threshold, and set the pixels with values above this threshold as 1, and the others as 0. However, this has obvious scalability issues. Hard-coding a threshold requires fine-tuning to different datasets and it is unlikely that this carries over well to other images. An alternative to this hard-coding is to let the algorithm select this threshold in an adaptive-manner by analyzing the distribution of the values in the image. Otsu Binarization is one such method, in which the algorithm assumes that the distribution of the image is bimodal(one mode for the background and the second for the foreground). The algorithm aims to find a threshold value that maximizes the inter-class variance value (here one class is composed of pixels less than the threshold, and the other is composed of pixels greater than the threshold. In particular, the sequential version of the algorithm works as follows:

Algorithm 1: Otsu Binarization

```
total_sum = 0
for i ← 0 to NUM_PIXELS do
    hist[img[i]] += 1
    total_sum += img[i]
end
sum_less = 0
num_less = 0
for i ← 0 to MAX_INTENSITY do
    sum_less += i*hist[i]
    num_less += hist[i]
    sum_more = total_sum - num_less
    num_more = NUM_PIXELS - num_less
    μdiff = sum_more/num_more - sum_less/num_less
    σ^2 = num_more × num_less ×μ_diff
    if σ^2 > max_var then
        max_var = σ^2
        threshold = i
    else
    end
end
//set values according to threshold
```

The results of applying the OTSU Binarization on the different images in the dataset are displayed in Figure 2.

Figure 2: Otsu Results(Images not shown to scale). The actual input to the OTSU algorithm is the gray scale version of the images

While the sequential algorithm is relatively easy to implement, the low arithmetic intensity of the algorithm makes it challenging to achieve speedups using parallelization. As mentioned above, we use both parallelization over different images, as well as within the algorithm.

Since we are parallelizing over both the images and the algorithm, we iterated over the same image 400 times to artificially create more images with equal workload.

4.1 Parallelization using OpenMP

4.1.1 Approach

The first approach was to parallelize the implementation over images. The parallelization is obtained by using a parallel for loop over these iterations. In addition, we use SIMD for setting the values after obtaining the thresholds. Interestingly, not much benefit was obtained using SIMD instructions which suggests that the overhead of creating the vectorization almost balances our the benefit obtained.

The second method involves the more interesting parallelization within the algorithm. The main algorithm can be broken into three main subsections-setting the values in histogram, and computing the

threshold, and setting the values of the final image based on the obtained threshold. The first phase has a huge locking requirement since a race condition could cause the histogram values to be incorrectly updated. As a result, we tried three different approaches to counter this. The first is to use openmp atomic operations to update the values, the second is to use fine-grained locks, and the third is to conduct this operation sequentially. In the end, we observed no advantage over doing the process sequentially, and in fact observed worse performance after doing this, and hence we ended up not parallelizing this part of the code. This is probably due to the small size of the histogram(256). It is likely that parallelization using fine grained locks would be beneficial in cases with similar problems but with larger array size.

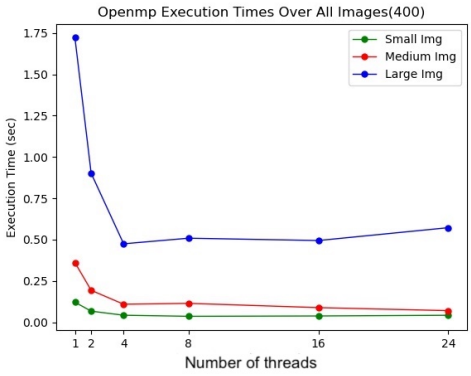
The second phase is to find the threshold. The sequential algorithm cannot be directly used in the parallel version due to inter-dependencies between the different loop iterations in the second for loop. Hence, we add two additional arrays-cumulative sum and cumulative histogram for the sum of the expected values(i.e value $\times$ number of pixels having the value) , and the number points below each threshold respectively. This naturally adds to the cost since the creation of the array, and the corresponding memory accesses are not present in the sequential iteration. To create the arrays we use parallel scan. Since in OpenMP the number of threads available is limited, we prefer the more work efficient version of scan to the more parallelizable one. After we have constructed the required arrays, we want to calculate the threshold that maximizes the inter-class variance. This also poses a bottleneck since max_var, and threshold are shared variables, and hence race conditions need to be prevented using critical sections. Another way to do this is using the built-in reduce 'max' of the openmp construct. This adds the requirement of using a second loop to check if the variance of the thresholds actually match the maximum variance-this is conducted by creating a third array to store the variances. However, in our particular project, the maximum sizes of all these additional arrays(cumulative histogram, cumulative sum, and variance) is 256, and the size of the L1 cache in the machine is 32k. Thus, we do not expect this to be a huge bottleneck as we expect all these arrays to fit into the cache.

The third phase(setting the values above the threshold to 255, and less to 0) is the easy part to parallelize-since there are no inter-dependencies between different for loops. Hence, we directly use a parallel for loop in this region.

4.1.2 Results

In this section, we will analyze the effect of using different optimizations on the performance. The parallelization over different images can be regarded as the best possible parallelization since the same image is used multiple times leading to a perfect work split.

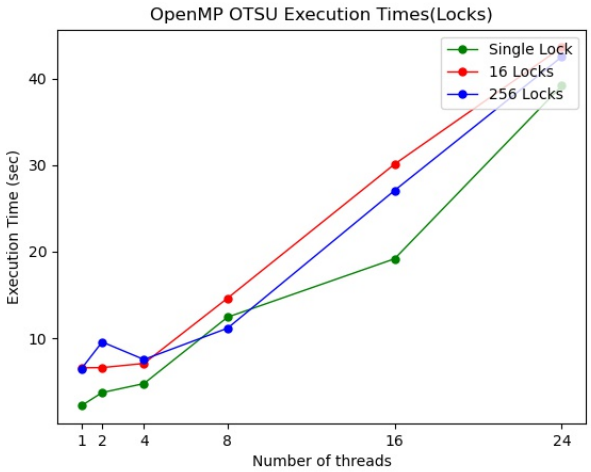
Figure 3: Results- OpenMP OTSU(Over Images)



Parallelization over different images

Please refer to Figure 3 for the corresponding graph. We can observe a good scale up using an increasing the number of threads. The time almost linearly decreases on increasing the number of threads from 1 to 2 to 4. However, after this, we can see that the execution times become almost constant after 4 threads. This is surprising since the workload is equally split. However, it must be noted that the amount of work done by each thread is not a lot(since the OTSU algorithm doesn't require much overall computation). Hence, once the number of threads is increased beyond 4, the cost of spurning more threads counteracts the increased parallelization obtained. The speedup can be observed by seeing the slope of the different graphs. It is clear that an increasing image size increases the speedup over all the treads till 4 threads. This is because higher image sizes require higher computation, and thus have a higher arithmetic intensity.

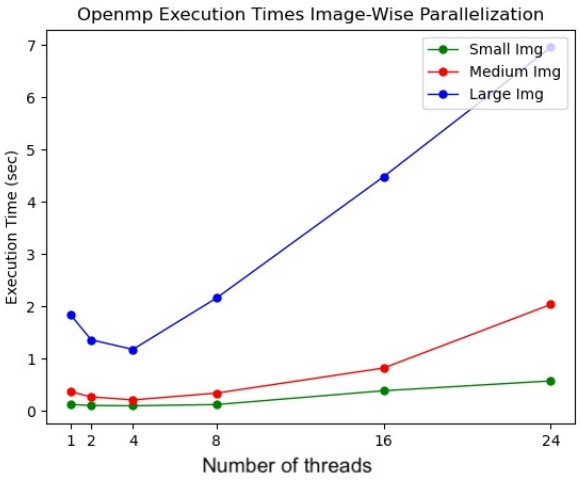
Figure 4: Results- OpenMP OTSU



Locking-based approaches

In addition to the final parallel approach followed(described below), we tested our implementation using different locking strategies while updating the histogram. In essence, we use a different number of locks for the updates of the histogram values. For the 1 lock case, we instead use the omp atomic clause, while for others we manually create the different locks. The size of the histogram is 256. The graph displayed in Figure 4 shows the results when using 1, 16, 256 locks on the medium sized image. As observed, we get both poor results, and scalability on doing this. This is due to the fact that the size of the histogram is small(only 256 ints). As a result, the cost of using locks/atomic sections causes a large amount of contention-since the size of the figure is large, as the number of threads increaes this far outweighs the advantage of increased parallelization. Among these approaches, it is observed that the atomic operations perform the best consistently(except a small deviation when number of threads is 8). We can attribute the atomic operation to perform the best since there is no process of 'acquiring/releasing the lock'. For less than 4 threads, the 16 and 256 locks perform similarly(256 is slightly worse) while for a higher number of threads, the number of 16 locks case performs worse than the 256 locks case. This can be attributed to the fact that when the number of threads is greater than 4, there is a high contention situation, which means it is more beneficial to have more locks-which results in. the 256 lock case to outperform the 16 lock case. However, for the 4 or below lock case, there is an additional overhead of creating 256 locks as compared to 16 locks with not as much contention leading to the 256 locks case to perform slightly worse.

Figure 5: Results- OpenMP OTSU



Final Parallelization over the algorithm

Please refer to Figure Below 4 threads, and across all images we observe a similar trends as compared to parallelization over images, which is promising since the parallelization over images is perfectly balanced. However, once the number of threads increases beyond 4, we start observing increase in execution times in contrast to the constant time observed before, which suggests that the cost of spurning more threads overpowers the speedup obtained. This is expected since we have some parts of the algorithm that are not parallized(such as the different levels of the scan operation, the setting of the histogram values), which means that the benefit of parallelization is lower as compared to parallelizing over the images-however the cost of the additional spurns remain constant. Similar to the parallelization over different images, it is clear that an increasing image size increases the speedup over all the treads till 4 threads. This is because higher image sizes require higher computation, and thus have a higher arithmetic intensity.

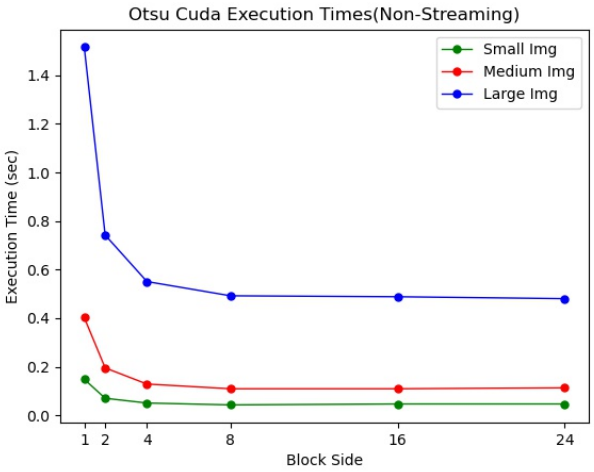
4.2 Parallelization using CUDA

4.2.1 Approach

Overall, the approach for parallelizing using the GPU is similar to the process used in OpenMP-we construct different arrays to process the cumulative histogram, and cumulative sum arrays, and use a reduction operation to do this. The bottlenecks are also similar in this case-the values of the histogram need to be performed in a manner that prevents race conditions. Despite the same overall idea, due to the specialized nature of the GPU, we used a different approach to counter these problems. In particular, one of the main bottlenecks of the operations on the GPU is the fact that the images need to be copied from the host device to the GPU and vice versa. This produces a huge bottleneck which is not present in the case of using OpenMP. This must be countered since the OTSU algorithm has low arithmetic intensity makes the process of getting efficient parallelization hard. Consequently, we used CudaStreams to allow for asynchronous memory copying and we operate on all the images in an asynchronous manner. In particular we start 400 streams for processing the different images asynchronously. This helps hide the memory latency to some extent.

After the image has been copied over to the GPU, we use kernels to produce the final thresholds, and update the values. To construct the histograms, we use atomic operations. This is much faster as compared to the OpenMP processes due to the high parallelization provided by the CUDA Kernal. For this kernel, the parallelization takes place over the different pixels.

We then construct the cumulative arrays using Shared Memory Inclusive Sum Scan. Here the parallelization occurs over the different pixel intensity values. Since there are only 256 such values, we can directly process these as a single block. After this, we use a synchronization so that all the threads have in fact processed the required arrays. Instead of using two different scans, we use the same scan operation to create the two arrays-this was seen to give much more speedup as compared to doing the scan on two different times despite requiring the same number of operations due to the less number of synchronization operations required. Finally, we calculate the variances(see sequential code) in the same kernel. The maximum variance is calculated using an Atomic Max operation. We again use atomic due to the increased parallelization contexts as compared to openmp, and correspondingly set up the threshold values. We then finally use a new kernel parallelized on the pixels again to set the final values using the calculated threshold value. The tests are conducted over a varying number of block size for checking scalability across different number of threads for the kernels parallelized over the pixels. However, the number of threads per block for the parallelization over the different pixel values is kept constant at 256.



)

Figure 6: Results- CUDA OTSU(Non-Streaming)

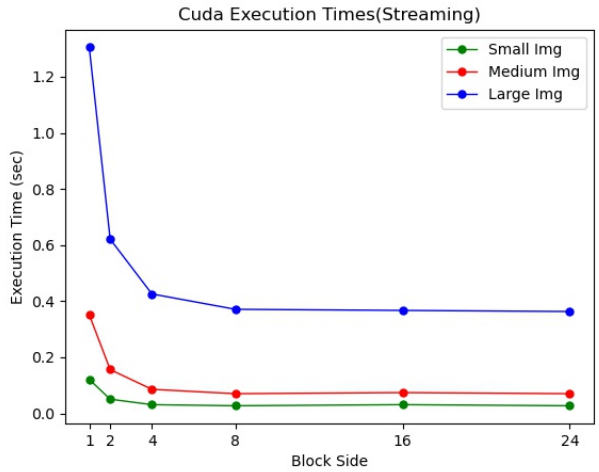


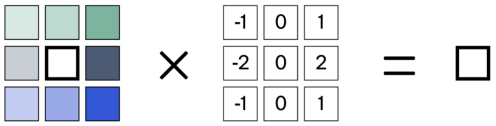
Figure 7: Results- CUDA OTSU(Streaming)

4.2.2 Results

Please refer to figures 6 and 7 for the corresponding graphs. The number of threads in a block is the square of the block side. First it must be noted that the slopes of the two graphs(i.e streaming and non streaming are the same). This is expected since streaming only hides the latency and shouldn't effect the change of performance of increasing the number of threads in the blocks. Further, the latency does in fact go down, and the lines can be seen shifted below their original placement. However, the shift is not constant across the different image sizes, in fact it is almost negligible when the size of the image is small. This is also expected since small images have less latency, so the use of streaming wouldn't make a huge difference. In contrast, the use of streaming for larger images does make a large difference(the observed time is about 14% less in the case of the large images)

We would now analyze the trends on increasing the number of threads in each block, for this you can refer to either figure since the trends are the same across streaming/non-streaming. It can be seen that increasing the block size to 64(side=8) decreases the execution time across all the images. However, beyond that there is no additional benefit observed. This is probably because of the use of atomic operations that counteracts the effects of increasing parallelization obtained by increasing the threads. This is also the reason why the slope slowly decreases on increasing the size of the block. It must also be noted that similar to OpenMP, larger images have higher speedup(see the slope of the graphs) as compared to smaller images due to increased arithmetic intensity.

Figure 8: Sobel Kernel applied over complete image



5 Edge Detection

Another way to binarize an image is to color the edges as one color and the non-edges as another. To use this, we can implement a edge detection algorithm that automatically identifies the edges. A common way of doing this is by finding the gradient of each pixel and comparing it to the surrounding pixels. A way of doing this is by using the Sobel 2D convolution kernel. The is defined in figure 8.

Hence a natural way to parallelize this is by parallelizing the process of applying convolutions in parallel. This is a common parallel processing problem with several techniques to have faster results. However, it is still worthwhile to see how these algorithms scale across different parallelization methods(i.e OpenMP, CUDA).

A benefit of using the Sobel operator is that is linearly separable as a combination of two one-d convolutions. Hence, we can use a similar procedure as stated in the lecture notes for OpenMP for blurring in this case. In particular, we implement and compare parallelization using both the separate convolutional kernels, and a single 2D kernel and report the findings on OpenMP and CUDA. The results of the application of the Sobel kernel of size 3 are displayed in figure 9.

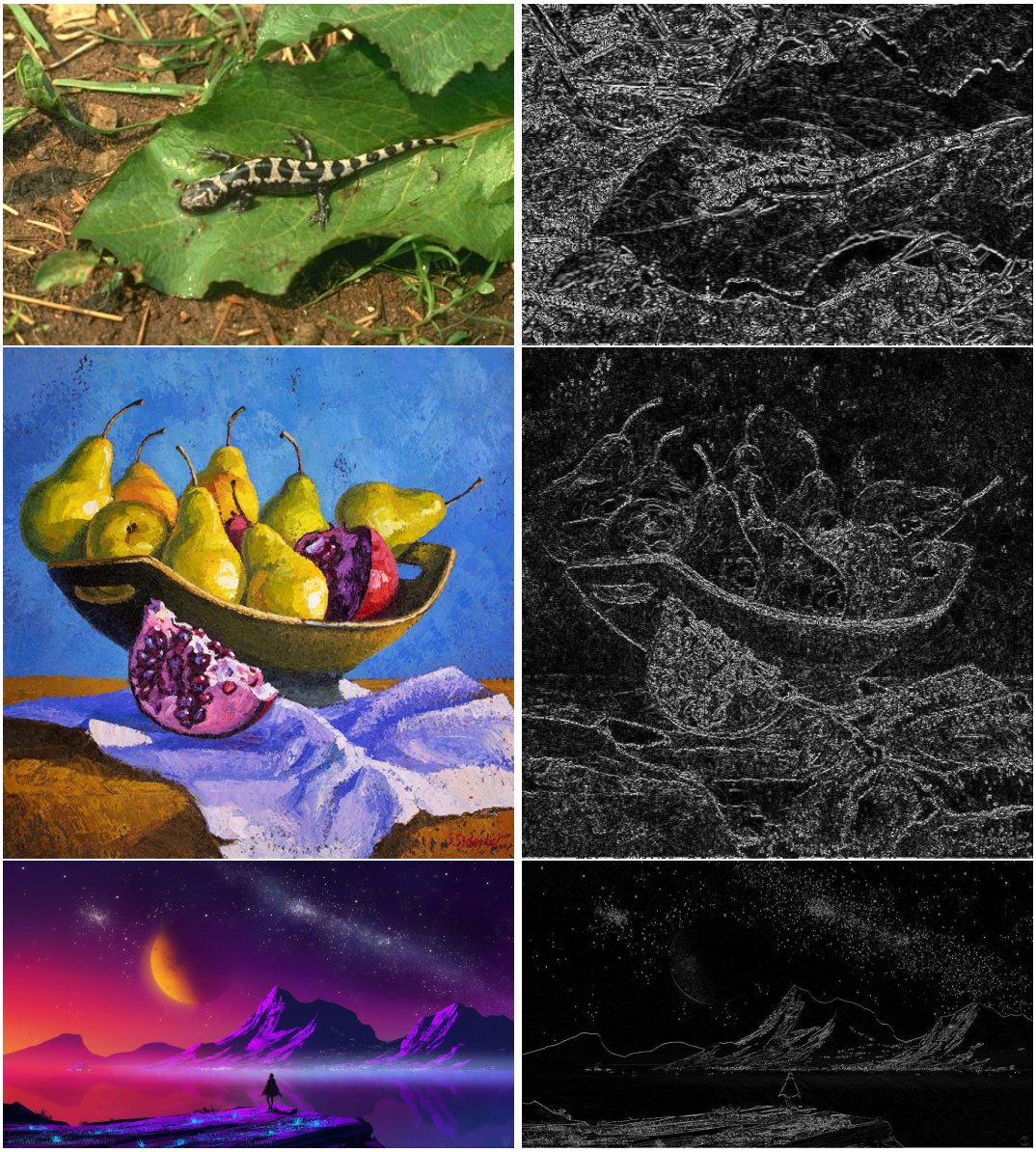


Figure 9: Sobel Edge Detection Results(Images not shown to scale). The actual input to the Edge detection algorithm is the gray scale version of the images

5.1 Parallelization using OpenMP

5.2 Approach

As mentioned above, we try the parallelization using two strategies- directly using the 2D convolutional kernel, or alternatively by exploiting the fact that this particular convolution kernel is separable-and first applying horizontal kernels, followed by the vertical kernels. The former requires a total work of 9N for N pixels, while the latter only requires 6N-as each pass requires a calculation over a kernel of size 3. However, the two pass solution requires the construction of a large temporary buffer. As mentioned in the lecture notes(with respect to blurring), a third intermediate strategy is to use chunking, wherein the image is subdivided into different chunks and the temporary buffer only stores the information corresponding to that particular chunk.

We implemented all three of these strategies. In particular, we parallelized the implementation over the different pixels, and also used Single Instruction Multiple data(and reduction) using pragma simd, over the kernels to obtain the final results.
For the chunking, we do not use the same size of the chunk across all but change the number of rows

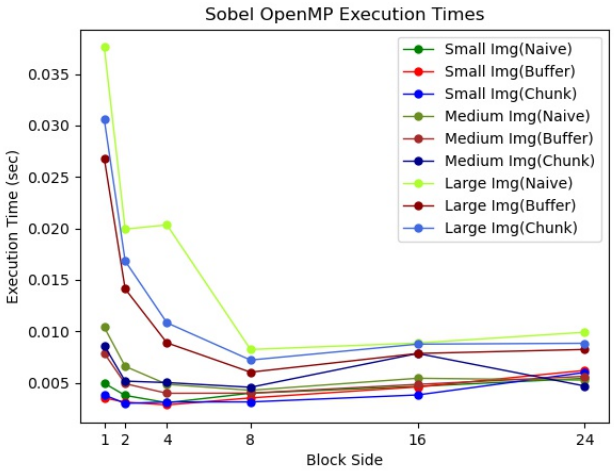


Figure 10: Results- OpenMP Sobel

present in each chunk according to the size of the image, to allow for the chunk to fit into the caches. The size of the L1 cache is around 32k, accordingly, we use a chunk size of 24 for the small image, 10 for the medium image, and 8 for the large image. The performance is compared using speed ups obtained over an increasing number of threads to analyze the scalability of the different algorithms. Across all the algorithms, we realize that since the workload is similar for all the pixels, there is no added advantage of using dynamic allocation-so we only use static allocation during parallelization.

5.2.1 Results

Please refer to Figure 10 for the graphs. Since the difference in the execution times are not very different, they are displayed on the same plot so that the changes are clearly observable. The execution times itself are small so the differences are still important. Shades of green are used for the naive implementation(i.e single pass), shades of red are used for the two pass implementation, and shades of blue are used for the chunked implementation. It can be seen that the naive implementation consistently performs the worst, across all the images, and over different number of threads. The results obtained are better for chunking for the medium and large images. However, they are worse than shared for the large image. The work for each image in Chunked is 6.4 $\times$ height $\times$ width, while the benefit is caching the values. Notice, that larger images have much higher height and width, and thus it makes sense for the increased cost to overpower the benefit of caching for these images.

5.3 Parallelization using CUDA

Similar to the implementation in OpenMP, we try experiments convolving using the 2D kernel directly, and by working on the linearly separable kernel. However, due to the specialized nature of the GPU, we adapt the method to exploit the use of shared memory in order to get the best speedups.

Our naive implementation works using a kernel in which each thread loads the pixel and the neighboring pixels to construct the result.

A second improvement on this implementation is using a shared space memory model, in which different threads collectively load the required pixels for that partiucular into a shared address space model. Further, we use two threads to load the pixels at the borders into the the shared address space. We then synchronize all the threads in the current block to ensure that the required values have been loaded into the shared array. We then apply the kernel on the array to obtain the final result.

The third improvement made is to extend the concept of chunking into the cuda model. Here the size of the chunks are exactly the size of the block, i.e we first use the horizontal 1D convolution on the value and then the vertical 1D convolution in the same block. This reduces the work done by each thread from 9B to 6B(where B is the size of the block).

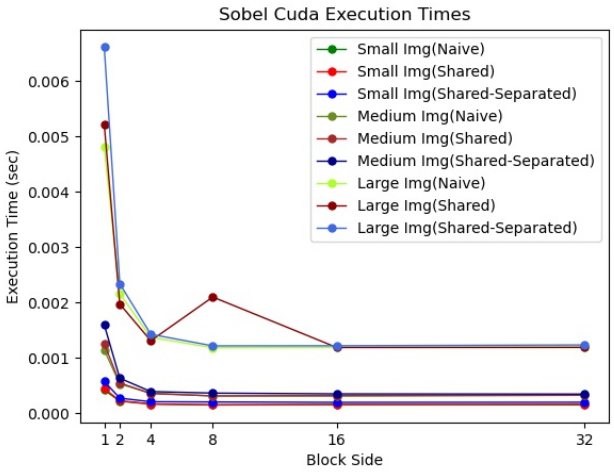


Figure 11: Results- CUDA Sobel

5.3.1 Results

Please refer to figure 11 for the graphs. Despite several optimizations, we weren't able to get any particular speedups using the different optimizations. The lines almost merge into one when the number of threads increase. This might be due to the small size of the convolution kernel. Increasing the number of threads likely decreases the amount of work done by each thread to a negligibly small amount, so much so the difference becomes negligible, and hence the benefit of decreasing the amount of sequential work done by each thread is offset by the cost of constructing new shared access arrays. This is an interesting observation that suggests that these optimizations don't change the execution times for smaller kernel sizes. We tried this on multiple sizes of kernels such as 3, 5, 7. However, no kernel observed any substantial change in speed at 1024 block size. Since, generally only these sizes of kernels are used in edge detection, we do not feel that these implementations are beneficial for this case.

6 k-means

In this project, we use the k-means algorithm for segmenting images by color. In particular, the algorithm clusters together pixels in an image which have a similar R,G,B color profile. **k** is specified as the number of clusters we want to see in an image. The sequential version of k-means proceeds as follows:

Algorithm 2: k-means Sequential

initialize k cluster centres randomly;
while *iteration* $<$ *NUM_ITERS* **do**
  assign closest cluster to all pixels using `color_distance`;
  //store in `assignments` array
  for $i \leftarrow 0$ **to** *NUM_PIXELS* **do**
    add `pixel`[i]'s corresponding `assignments`[i] to the cluster's sum of $x, y, r, g, b, count$;
  end
  for $c \leftarrow 0$ **to** *NUM_CLUSTERS* **do**
    update cluster c's mean using its accumulated $x, y, r, g, b, count$ values from pixels
  end
end
Result: Set the final image to be pixel's color values same as their final cluster's average

The `color_distance` function is given as:

$$(p1.r - p2.r)^2 + (p1.g - p2.g)^2 + (p1.b - p2.b)^2$$

where $p1, p2$ are two pixels and r, g, b are the red, green, blue components of a pixel.
Some of the data structures which will be relevant to our upcoming analysis are:
Arrays: `points, assignments, clusters`
Struct: `Point` (has fields $x, y, r, g, b, count$ for x, y coordinates, r, g, b color components and *count* (individual or accumulated) of pixels)
The results of the color segmentation on 3 of our sample images are given below.
All several hundreds or thousands of pixels in an image are looped over to calculate their cluster assignments and eventually to calculate cluster centres. We also go through multiple iterations of repeating this process, so we can ultimately find good cluster averages. Note that in an alternative k-means implementation, the iterations can stop at convergence as opposed to a constant value; we go up to a constant value. Finally, we must also perform the cluster average computation over k clusters.
There are also some steps in the naive k-means algorithm that are fundamentally sequential. Before calculating the new cluster averages in an update step, we must wait for all the pixels to be assigned to their closest cluster. Before moving on to a next iteration, we must wait for all the new cluster averages to be calculated.

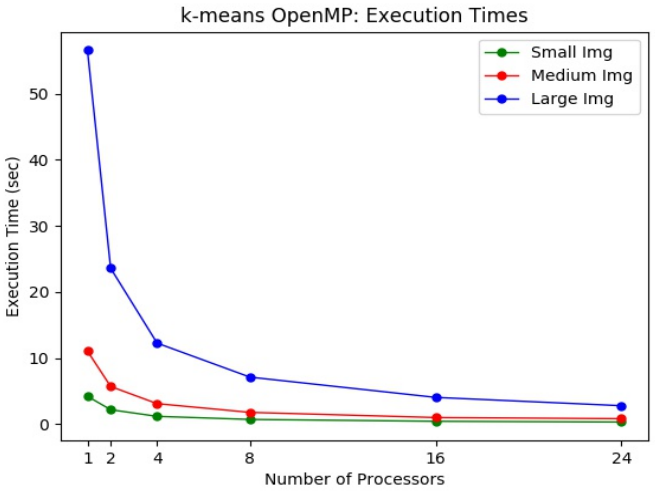
Figure 12: Results for KMeans Image Segmentation on k=3

6.1 Parallelization Using Open MP

We implemented k-means in OpenMP and tested it on Bridges with $1, 2, 4, 8, 16, 24$ processors. There are few different operations that happen several times in k-means, as discussed in the sequential section.
a. We calculate the assignments of points' closest cluster centre in parallel.
b. We use multiple reduction variables to then sum over the different pixels that belong to a given cluster. We do this for every cluster.
c. We apply averaging by dividing by count in parallel over all clusters.

Some of the crucial shared variables include points, assignments, means (of clusters). Since the iterations are fundamentally sequential and so is the dependency between the assignment and averaging step, we do not parallelize across those. We attempted to reduce multiple loops for the summing of $x, y, r, g, b, count$ values and averaging by including a critical region within one parallel loop over all the pixels. Instead of reducing the time due to less sequential steps, it significantly increased the execution time. Given the size of images we are dealing with, several hundred pixels trying to atomically update a bunch of variables causes a lot of synchronization stalls. This overpowers the benefit of more efficient looping otherwise.
Moreover, we noticed near linear speedups over increasing processors in OpenMP with the parallel im-

Figure 13: $k = 3, \text{num\_iters} = 2048$



plementation mentioned (Figure 13). We report the execution times for $k = 3$ with 2048 iterations. OpenMP times were mostly consistent across different ks and iterations, as one would expect.

6.2 Parallelization Using CUDA

6.2.1 Basic Parallel

We also implemented and tested k-means on GPUs using the CUDA programming language. The parallel and sequential regions remain portions of the algorithm remain the same, but pose a very different challenge when working with kernels in CUDA. Many threads of lower compute power are working together in a GPU. k-means on image processing, with several pixels to work with at once, presents itself nicely for parallelization. At the same time, we need to be storing the sums of all different clusters across all the pixels (of their x, y, r, g, b values) in an image. Certainly this requires updating the same memory location for several threads. A simple but slow solution is to just perform this section in sequential. In an unrealistic scenario, this would give the most desirable speedups if we had access to one powerful processor just for the sequential regions (throwback to heterogenous memory lecture). Since all cores on the GPU are weak, this simple solution increased the execution time on the small image with $k = 4, iters = 2048$ by a factor of 10. As a next step, we attempted using atomic add, which would essentially make the summing kernel a sequential operation. So, we moved on to using different parallel methods for this global pixels' summation.

6.2.2 Global Scan

In this approach, we used our exclusive scan implementation from Assignment 2, but modified to work with Point structs and different summation operators. We use a scratch array, of the same dimensions as our points array, to fill in the cumulative sums of the pixels in the update step. We only use the last element in this array; we summed it with the last element in the points array, since scan is exclusive (but removed due to it being unnecessary in the grand scheme of averaging). This parallelized a completely sequential update region to be extremely parallel. Unfortunately, this also increased the memory access times by a lot. In particular, note that we introduced an otherwise redundant set of accesses to the k-means algorithm. We observed $2x$ execution times with this implementation as compared to sequential. On doing a deeper dive into what operation exactly was causing the bottleneck, we realized it were the following read-write-modify instructions during the summation operations in prefix sum calculation by scan:

```
x+=p.x;
y+=p.y;
```

```
r+=p.r;
b+=p.b;
g+=p.g;
count+=p.count;
```

Certainly, we expected some increase due to memory accesses but this was still a lot with no significant improvement over different input images. After experimenting with some C++ class vs struct idiosyncrasies, we concluded this really was a result of not optimal struct declaration (throwback to 213 struct questions). Some struct fields were doubles when they need not be and there was inefficient padding. On re*srtuct*uring, we started to see improved execution times, in fact with speedups relative to sequential for larger images (Figure 15). The overhead of memory accesses is overpowered by the extreme parallelization power over thousands of pixels for bigger input images.
Note that making similar changes in the sequential and OpenMP versions did not lead to any significant timing differences. This is because the GPU cache is smaller as compared to that of the bridges or latedays machines we work with for sequential/OpenMP. Thus, such fundamental structural change led to significant speedups.

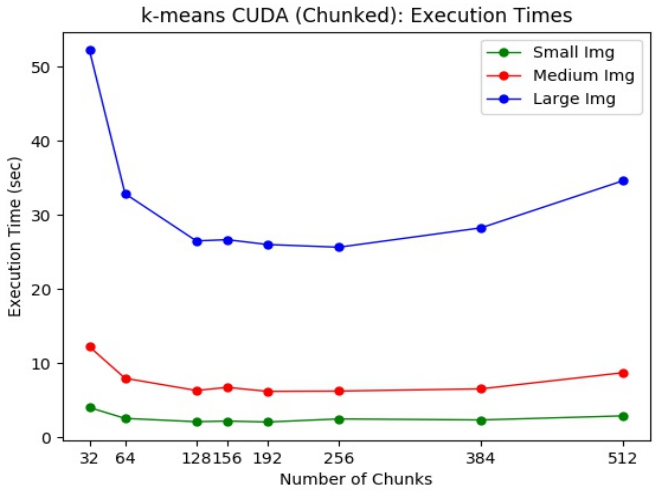
6.2.3 Shared Memory Scan

As a natural extension of last approach, we moved on to a shared memory prefix sum implementation. We expected that the additional memory accesses as a result of scan would become overpowered by image size even sooner in this case as opposed to using global scan. In doing so, we used the warp-based shared memory scan implementation from assignment 2. After spending significant time debugging the issue with this implementation, our closest conclusion to why it was not working was large shared array declarations (note that these struct arrays are bigger than just the int arrays we used in assignment 2). Since CUDA is very shy about reporting errors, we decided to try another approach after trying several methods here and spending significant time debugging this.

6.2.4 Chunked Updates

In this approach, we divide our update step over all pixels into equivalent chunks. The image is divided into *num_chunks* equal regions in row major order. We use a global array which stores the chunk-wise averages for a given cluster. These chunk wise sums are then used to calculate the overall sums and updated cluster centres. The calculation of overall values happens in sequence using one CUDA thread, which did not make sense to parallelize over a small number of iterations using scan/atomic operations. Certainly, in this approach we save on the memory accesses that scale with number of pixels in an image in the scan implementation. But, we have some degree of sequential-ness in this due to only splitting into a fixed number of chunks. Now, note that, in theory it should be it should be possible to essentially simulate the sort of chunking that goes on underneath the hood of OpenMP. Then, we could have a competitive execution time. Again, this does not quite happen due to weaker individual GPU cores. We experimented with different numbers of chunks and noticed some interesting curves as shown in Figure 14 for different chunk numbers.
There is a sweet spot for getting the best execution times. This was consistently in the range $128 - 192$ for different image sizes. If the number of chunks is less, the net parallel compute power of weaker threads is not enough to overpower the compute capability of one strong CPU core. If the number of chunks is more than the sweet spot, the overhead of summing in sequence the values of different chunks (that too for a large number of iterations) takes over the benefit of increased compute parallelization.

Figure 14: $k = 3, \text{num\_iters} = 2048$

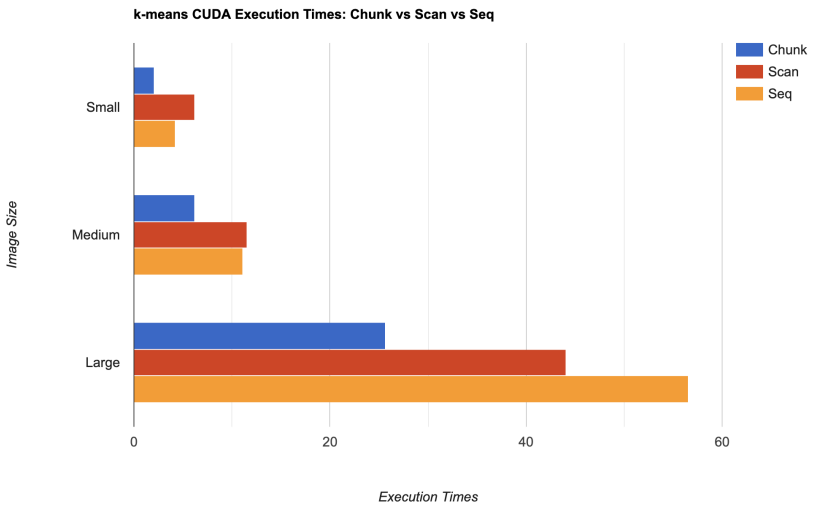


Scan vs Chunk

We compare our scan implementation with our chunk-based implementation (referred to as chunk here on wards).

1. As seen in Figure 14, chunk (at 192 chunks) consistently outperforms the sequential implementation with respect to execution times, while scan's benefit becomes net positive only for larger image sizes, as discussed before. The factor by which chunk speeds up k-means increases with image size. For 321 pixels, it gives $2x$ speedup. For 1920×1080 pixels, it increases to $2.25x$ speedup. The benefit of parallelizing computation over larger number of pixels starts to overpower the overhead of overall, sequentially summing chunked sums for every cluster in every iteration.
2. The relative number of cache references in scan and chunk are indicators of the additional prefix sum array memory accesses in the former and the absence of that in the latter. Certainly, chunk outperforms here.
3. Overall, we observe very low cache miss rates (max is 10% for large image in chunk). Note that (Figure 16), we notice much lower cache miss rates for our scan implementation as compared to chunk. This is because the scratch array getting used for storing prefix sums is accessed with good temporal and spatial locality. On the other hand, chunk uses does not make any such accesses. The only large arrays it accesses are points and assignments. These are consistently accessed each iteration; we observe overall low miss rates as a result. But, scan has even lower miss rates, because with the much increased number of cache references comes very good concentrated locality in the scan operations.

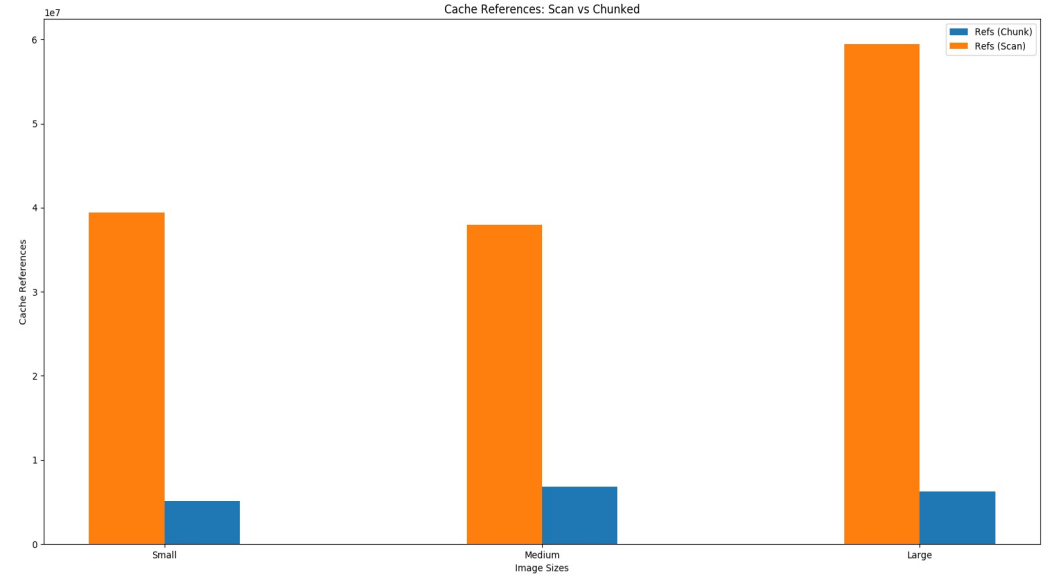
Figure 15: $k = 3, \text{num\_iters} = 2048$

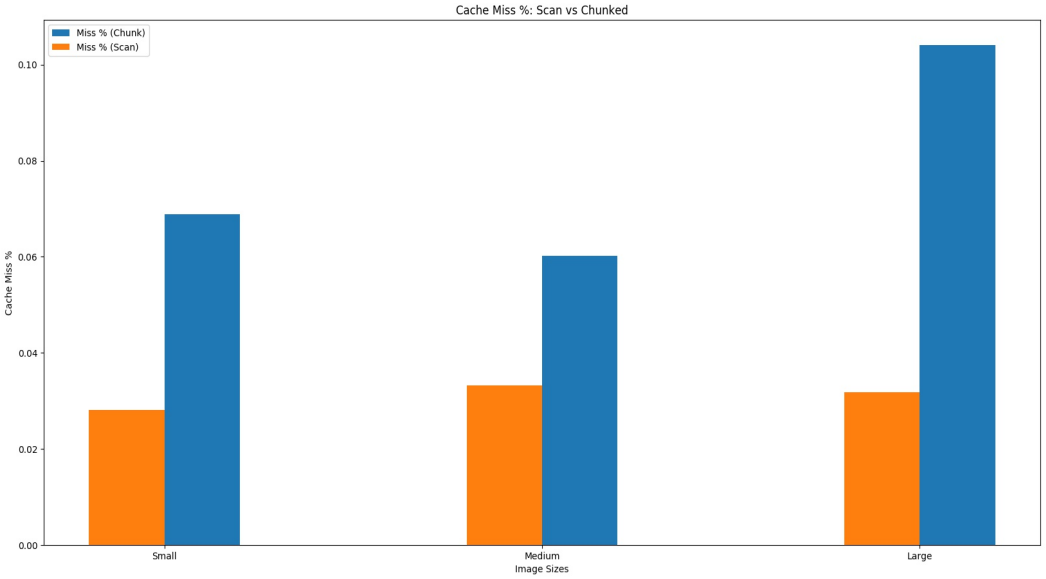


6.2.5 Shared Support

In this approach, we sort of combined the chunked and shared memory approaches. Since scan is too memory inefficient for summation, we added shared support arrays per block to copy over the points from global memory. Certainly, we had to introduce a block level barrier using `__syncthreads()` in order to make the storing of global points into shared block wise arrays synchronized. Then, we used a single thread in the block to update a given block's cluster sums. And after global synchronization beyond this step, the the global mean was calculated in sequence by a single thread.
Note that this approach worked better than the shared scan approach as we do not run out of shared memory too soon, but we do eventually. Certainly, we optimized our memory usage by only story 8-bit integer values in the shared array where we could (e.g. r, g, b).
As compared to the chunk implementation, at block size = 1024, we observe
approx. **2x** speedup for **small** image (2.045479/1.034825) &
approx. **2.6x** speedup for **large** image (6.178567/2.378680).
But, this method does not work for our sample large image most likely due to memory limitations.

Figure 16: $k = 3, \text{num\_iters} = 2048$





6.3 Overall k-means

Overall, we are able to achieve good speedups for the k-means algorithm in both OpenMP and CUDA as compared to the sequential algorithm.
We observe that OpenMP is able to give near linear scaling (upto 16) in speedup with increase in number of processors $(1, 2, 4, 8, 16, 24)$. At the same time, CUDA's best performance on our small and medium test images is overall approx. $4x$ speedup (shared support approach) and for large image it is approx. $2.25x$ (chunked approach). Fundamentally, the k-means algorithm involves significant reduction steps (i.e. averaging) over a very large number of pixels with multi-channel information. So, we have to give up on execution time for memory access stalls if we want a parallel reduction approach like scan or sequential merging overhead if we want a memory free approach like shared support and/or chunking.
Assuming availability of 24 processors on Bridges machine and the GHC GPUs, these are the best relative execution times of k-means with sequential, OpenMP and CUDA implementations for $k = 3, num\_iters = 2048$ with optimal hyper-parameters in respective implementations.

Table: Execution Times at $k = 3, num\_iters = 2048$

Image	Sequential	OpenMP	CUDA
Small	4.221389	0.329557	1.034825
Medium	11.067143	0.829493	2.378680
Large	56.586520	2.796634	25.625183

Table: Speedups at $k = 3, num\_iters = 2048$

Image	OpenMP	CUDA
Small	12.8x	4x
Medium	13.3x	4.6x
Large	20.2x	2.2x

7 Conclusions

Overall, we saw some overarching common themes. It is hard to strike the right balance between preventing memory access stalls and still performing reduction operations in parallel. Since image processing fundamentally involves several hundreds and thousands of pixels, all of them making accesses and needing synchronized reduction operations makes this set of problems challenging. Nonetheless, we were able to notice significant performance improvement over our final versions of OpenMP and CUDA algorithms for otsu binarization, sobel edge detection and k-means clustering. Please refer to the figures and tables included throughout to look at same.

8 Future Work

Overall, for each of the algorithms we worked with, we were very excited by the potential for further optimizations that we could not get to due to time limitations.
For instance, a special case of Otsu binarization is more regional and adaptive as compared to the one we implemented. We calculate the binarization thresholds over different subregions of the image. This is naturally parallelizable (think blocks). Certainly, it does not solve the problem of finding a global threshold though. To do so, we propose a mid-way, heuristic method to get a compromise between regional and global thresholding, but with better execution times as compared to the sequential global implementation. After computing the blockwise thresholds based on regional maximal difference in summed variance, we assign a *fake* global threshold as the mean/median of these thresholds. The interesting pixels are those whose grayscale values fall between corresponding regional threshold and the fake global threshold. We can then assign pixels on the higher half of this interval as white while the lower as black.
And then in the case of k-means, we can follow an approximation approach as well. We can asynchronously update (with some level of granularity) the cluster means as we keep getting new points for a cluster as opposed to waiting for the complete assignment step to finish. We can then exit the k-means

algorithm on convergence, and it would be an interesting question to analyse the trade-off between convergence time and average per iteration execution time.

9 References

1. Otsu N., A Threshold Selection Method from Gray-Level Histograms. https://engineering.purdue.edu/kak/computervision/ECE661.08/OTSU_paper.pdf
2. http://www.cs.cmu.edu/afs/cs/academic/class/15418-f20/public/lectures/25_dnn.pdf
3. https://reasonabledeviations.com/2019/10/02/k-means-in-cpp/,https://github.com/goldsborough/k-means/blob/master/cpp/k-means.cpp

10 Work Distribution

Shreya - Otsu binarization and Edge detection seq, OpenMP, CUDA; some k-means CUDA debugging; final testing and report
Kusha - Infrastructure; k-means seq, OpenMP, CUDA; some Otsu and Edge debugging; Project checkpoint report and testing; final testing and report
Work Distribution - 50%-50%