Accelerating Image Segmentation with PCA-Optimized K-mean clustering and CUDA Parallelization

Noshin Chowdhury & Hongfei Liu & Weiheng Xiao & Kaifan Zheng

Abstract — Our project, "Accelerating Image Segmentation with PCA-Optimized K-mean Clustering and CUDA Parallelization," presents an innovative approach to improve image segmentation in computer vision. By integrating Principal Component Analysis (PCA) with K-Means clustering and CUDA parallelization, we significantly enhanced accuracies and processing speeds in image segmentation. PCA aids in efficient data handling and feature extraction, while K-Means clustering ensures precise data segmentation. CUDA parallelization leverages GPU computing for rapid execution. This method not only refines the accuracy of image segmentation but also addresses the challenges of real-time processing and large dataset management, marking a substantial advancement in computer vision applications.

keywords: Image Segmentation, Principal Component Analysis, K-Means Clustering, CUDA Parallelization, GPU Computing, Computer Vision, Machine Learning, Computational Efficiency

I. Introduction

In the rapidly evolving field of computer vision, accurate and efficient image segmentation remains a crucial challenge. Our project, "Accelerating Image Segmentation with PCA-Optimized K-mean Clustering and CUDA Parallelization," introduces a groundbreaking approach to address this challenge. By integrating Principal Component Analysis (PCA) for dimensionality reduction and feature extraction, alongside K-Means clustering for effective data segmentation, we substantially enhanced the precision of image segmentation. A significant innovation in our approach is the application of CUDA parallelization, utilizing GPU computing to drastically accelerate the processing speed. This project not only aims to refine the accuracy of image segmentation but also targets the efficiency challenges, particularly in scenarios involving real-time analysis and large datasets. Through this integration of advanced techniques, our work endeavors to set a new benchmark in the field of computer vision, contributing substantially to its applications ranging from medical imaging to autonomous vehicle navigation.

II. Experiment Procedure and Analysis

A. Part 1: Parallelized PCA Sequential algorithm analysis

Principal Component Analysis (PCA) is a method employed for examining structures in high-dimensional data. Its primary function is to reduce the dimensionality of data, enabling the identification of features that are more comprehensible and facilitating the acceleration of information processing. Furthermore, PCA has applications in both visualization, in reducing data dimensions to two and noise reduction. The process shown as the formulas following.

Covariance and divergence matrices:

Sample mean: sample covariance:

$$X = \frac{1}{n}\sum_{i=1}^{N} x(i) \qquad S^2 = \frac{1}{n-1}\sum_{i=1}^{n} (x(i) - X)$$

Principle of eigenvalue decomposition matrices:

$$Av = \lambda v$$

The eigenvalue λ corresponds to its eigenvector v, and a collection of eigenvectors from a matrix forms a set of orthogonal vectors.

$$A = Q\Sigma Q^{-1}$$

In this context, Q represents the matrix comprising the eigenvectors of matrix A, while Σ is a diagonal matrix whose diagonal elements are the eigenvalues.

In our experiments, we adhered to the PCA formula as previously described, focusing initially on implementing a sequential program for PCA. This implementation, in the absence of parallelization,The time complexity of the sequential PCA program can be articulated as O(n^3).

To illustrate the actual performance of our implementation, we measured the run time of the sequential PCA program under various conditions. The results of these measurements are depicted in the table below.

data size(number of pixel)	CPU runtime(ms)
32*32	0.3520
64*64	0.9380
128*128	2.9520
256*256	7.7910
512*512	33.318

As the data size increases, the CPU runtime increases exponentially. This graph provides a visual representation of the execution time, allowing us to analyze the efficiency of the sequential PCA implementation and set a baseline for comparing the performance improvements achieved through CUDA parallelization in subsequent experiments.

B. Part 2: Parallelized PCA algorithm analysis

In our project's next phase, we parallelized key PCA sub-functions using CUDA kernels to enhance efficiency. This

included parallelizing the mean calculation, data normalization, covariance matrix computation, matrix transposition, and matrix multiplication. By leveraging GPU capabilities, we aimed to significantly reduce computation times, particularly for large datasets and real-time analysis, and compared this to our initial sequential implementation to gauge the improvements. Our team tested the runtime by setting the number of threads to 1024, and number of block of 32 The runtime result as the graph below:

data size(number of pixel)	GPU runtime(ms)
32*32	0.7572
64*64	1.1801
128*128	2.7338
256*256	7.5808
512*512	30.565

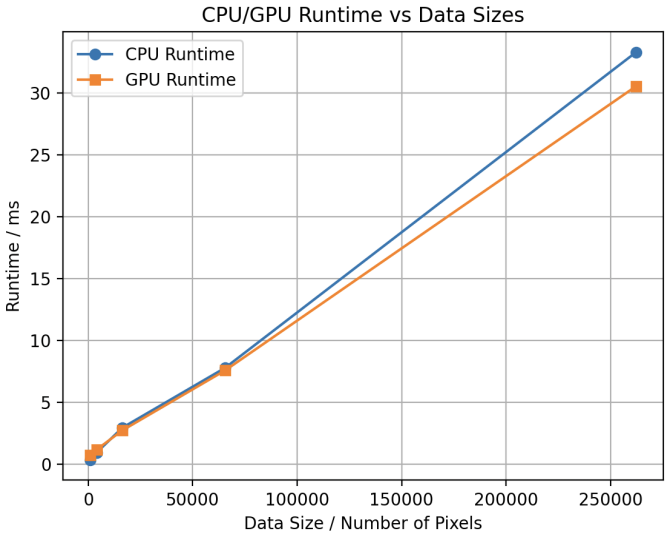

Figure 1: Result of different runtime

Both CPU and GPU runtimes increase as the data sizes grow. This is expected as larger datasets require more computational resources. However, the rate of increase differs between the CPU and GPU. Initially, for smaller data sizes (e.g., 16x16), the CPU and GPU runtimes are relatively close, with the GPU being slightly slower. This could be due to the overhead involved in parallel processing setups in GPUs, which may not be as beneficial for handling smaller data sizes. The increase in runtime for both CPU and GPU is non-linear. This suggests that the complexity of processing increases more rapidly than the data size itself. The intersection point between the CPU runtime curve and GPU runtime curve is at Number of Pixels = 10559. For the CPU, this increase is more pronounced than for the GPU, particularly at higher data sizes. The data indicates that while GPUs offer a significant performance advantage for larger datasets, there is still a notable increase in runtime as data size grows. This points to the importance of algorithm optimization and efficient use of hardware resources to manage larger datasets effectively.

In this part, the GPU runtime is close to the CPU runtime. This is due to the heavy and non-parallelized eigen-decomposition process in PCA operation.

Next, our team measures the runtime when the number of threads used is equal to {1, 4, 8, 16, 64, 128, 256, 512, 1024} with the data size of 256*256. The measuring result shown below:

number of threads	GPU runtime(ms)	Speedup
1	214.98	1.000
4	83.662	2.570
8	58.176	3.695
16	32.263	6.663
64	15.274	14.075
128	10.223	21.03
256	8.963	23.99
512	7.597	28.30
1024	7.504	28.65

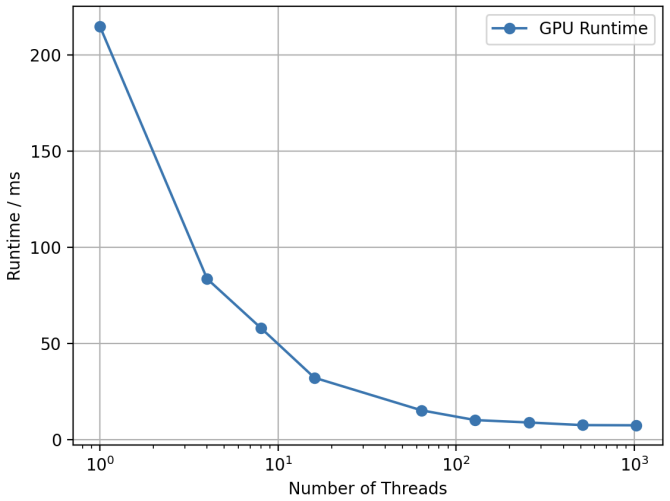

Figure 2: Result of Different number of threads

The most obvious observation is the sharp decrease in GPU runtime as the number of threads increases. Starting from a high runtime of 214.98 ms with a single thread, there is a rapid decline as the thread count rises, particularly up to 16 threads. Beyond a certain point, specifically after 64 threads,

the rate of decrease in runtime becomes less pronounced. While runtime continues to improve as the number of threads increases from 64 to 1024, the incremental gains are smaller. This suggests that there is a threshold beyond which adding more threads yields diminishing returns in terms of reducing runtime due to parallelization overhead. The data indicates an optimal range of thread utilization where performance gains are most significant. Initially, increasing the number of threads from 1 to 16 results in substantial improvements. However, beyond 64 threads, the improvements are less dramatic. This pattern reflects the balancing act between parallelization benefits and the overhead associated with managing a large number of threads.

Then we computed the speedup and plotted the speedup against the number of threads:

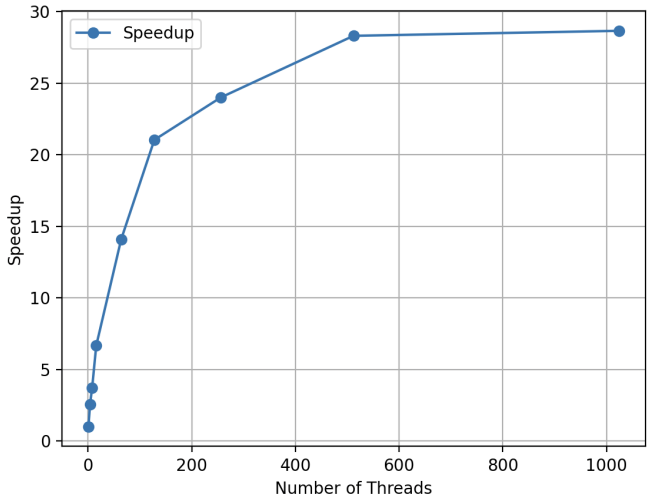

Figure 3: Speedup on different number of threads

With a closer look at the above diagram:

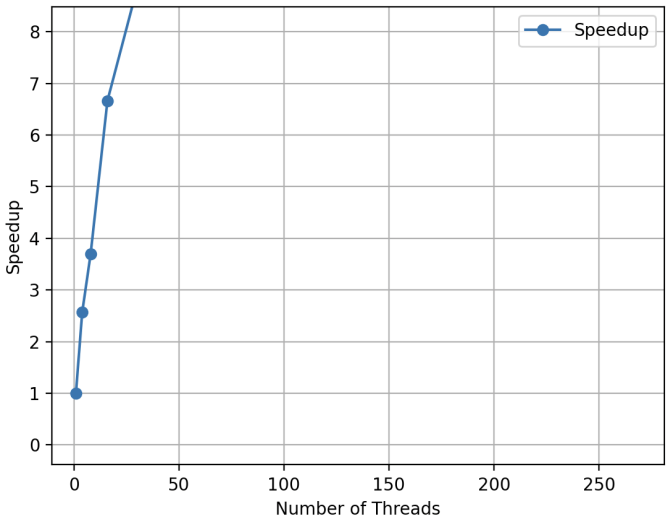

Figure 4: Zoom in figure on figure 3

In the diagram, we can see that with growing numbers of threads, the runtime decreases and the sequential parts get more dominant and slow down the speedup. This follows the theory of Amdahl's Law. According to Amdahl's Law, we can estimate a sequential execution time of around 7ms.

C. part 3: K-mean sequential algorithm analysis

K-means is a clustering algorithm that partitions a dataset into K distinct, non-overlapping subgroups (clusters) based on similarity. It iteratively assigns data points to these clusters based on the mean values of the points in each cluster, minimizing variance within clusters and maximizing it between them.

Select initialized k samples as initial cluster centers:

$$a = a1, a2, a3, ..., ak$$

computing cluster center:

$$a(j) = \frac{1}{|c(i)|} \sum_{x \epsilon c(i)} x$$

Following the outlined formula, our team developed a sequential implementation of the K-means algorithm. The algorithm contains two important parts; update the center coordinate, and find the nearest center for each point.

The time complexity for the sequential program is O(n*k*i*d). We conducted tests to evaluate the CPU runtime across various input data sizes, and the results of these tests are presented in the table below.

data size(number of pixel)	CPU runtime(s)
32*32	2.2051
64*64	8.0480
128*128	26.904
256*256	110.49
512*512	441.27

D. part 4:Parallelized K-mean algorithm analysis

In advancing our project, we parallelized key components of the K-means algorithm using CUDA kernels, focusing on speeding up the calculation of new cluster centers and the simultaneous label assignment for each pixel.

Noticeably, we used atomic operation atomicAdd() when updating the sums and counts. This atomic operation helps avoid race conditions, and allows parallelized functions to access the critical section separately.

Utilizing GPU capabilities, our objective was to drastically cut down on computation times, a crucial factor for handling large datasets and enabling real-time analysis. This was benchmarked against our initial sequential implementation to evaluate the extent of performance enhancement. For our tests, we configured the system to use 1024 threads, and the outcomes in terms of runtime are depicted in the graph provided below.

data size(number of pixel)	GPU runtime(s)

32*32	1.888
64*64	2.389
128*128	2.580
256*256	9.067
512*512	46.42

To compare the CPU runtime and GPU runtime, we plotted the two curves on the same diagram:

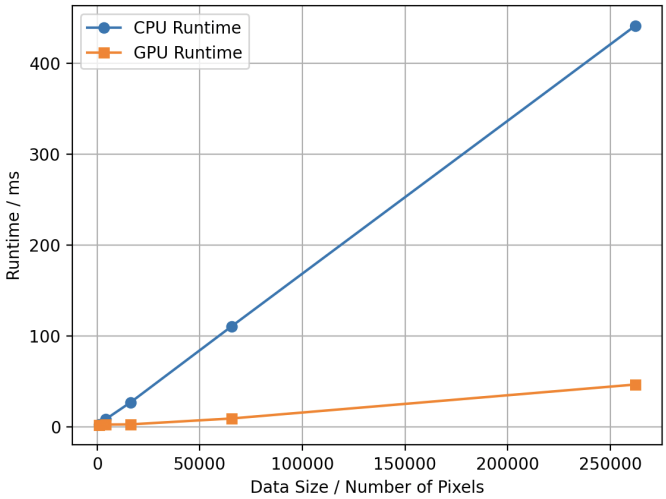

Figure 5: Parallelized result

With a closer look at the above diagram:

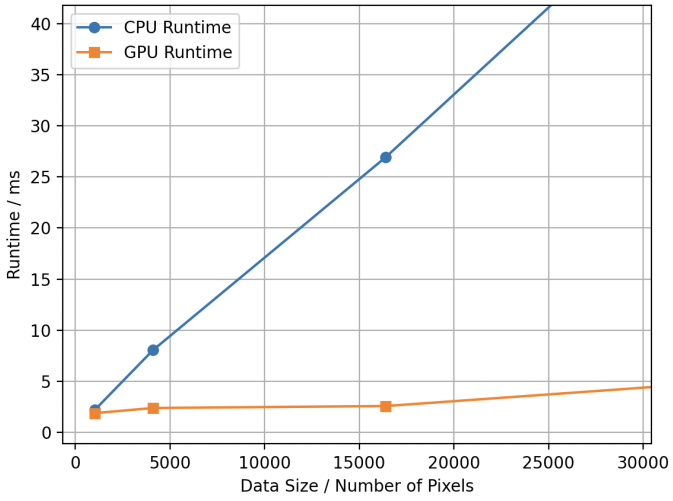

Figure 6: Zoom in figure on figure 5

For the CPU runtime curve, we can see that it strictly follows a linear relationship between the runtime and data size such that approximately Time = Number of Pixels / 595 for large data sizes (Number of Pixels starting from 128*128). While for the GPU runtime curve, we can also observe an approximately-linear relationship of Time = Number of Pixels / 6400. For the K-mean part, without the heavy and non-paralleled eigen-decomposition process, we can see the effectiveness of parallelization in enhancing the computation speed by around 11 times.

Noticeably, for small data sizes (Number of Pixels at 32*32 and 64*64), we can see that by multiplying the image size by four only enhances the execution speed by 1.265 times. This is due to the existence of sequential overheads for setting up the paralleled computation processes.

Subsequently, our team conducted runtime measurements using varying thread counts, specifically {1, 4, 8, 16, 64, 128, 256, 512, 1024}, while processing a data size of 256*256. The results of these measurements are presented in the table below.

number of threads	GPU runtime(s)	Speedup
1	1913.061	1.000
4	445.422	4.295
8	313.914	6.094
16	187.912	10.18
64	67.0908	28.51
128	20.4180	93.69
256	9.54750	200.4
512	9.07221	210.9
1024	10.5352	181.6

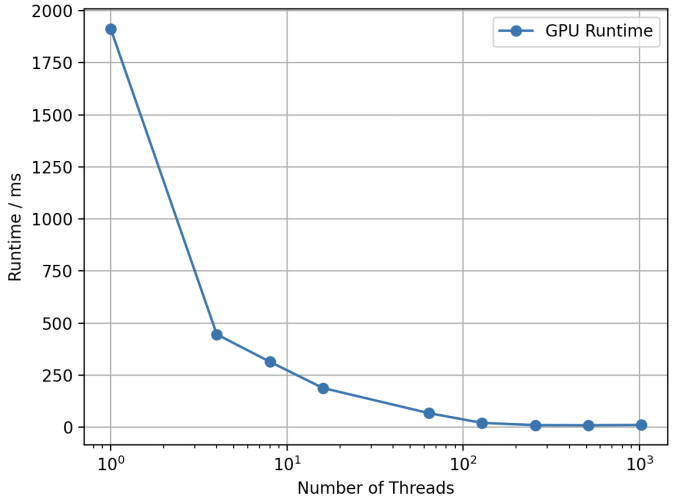

Figure 7: Result of Different number of threads

The most obvious observation is the sharp decrease in GPU runtime as the number of threads increases. Starting

from a high runtime of 1913.1 ms with a single thread, there is a rapid decline as the thread count rises, particularly up to 256 threads. Beyond a certain point, the runtime fluctuates around 10ms. This suggests that there is a threshold beyond which adding more threads yields diminishing returns in terms of reducing runtime due to parallelization overhead. The data indicates an optimal range of thread utilization where performance gains are most significant. Initially, increasing the number of threads from 1 to 256 results in substantial improvements. However, beyond 256 threads, the improvements are less obvious. This pattern reflects the balancing act between parallelization benefits and the overhead associated with managing a large number of threads.

Then we computed the speedup and plotted the speedup against the number of threads:

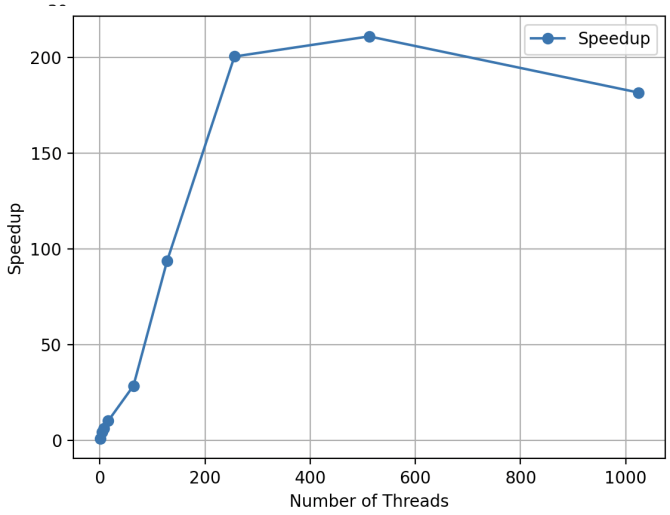

Figure 8: Speedup on different number of threads

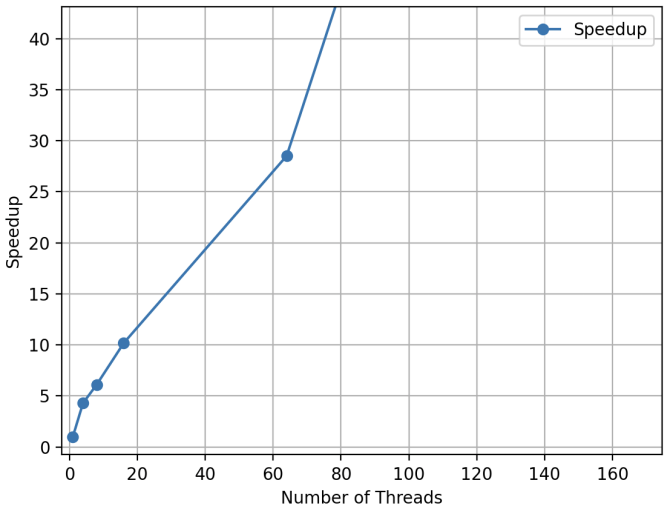

Figure 6: Zoom in figure on figure 8

In this diagram, we can observe that from thread number equals 1 to 256, the speedup curve approximately follows the pattern of doubling the thread number doubles the speedup.This corresponds to the theory of parallelization.

Similar to the "Speedup vs Number of Threads" in the PCA part, this diagram also follows Amdahl's Law. For thread numbers larger than 256, the speedup values fluctuate around 200ms. Hence, according to Amdahl's Law, we can observe the sequential parts execution time of around 9ms.

Compared to the PCA past, the K-mean part generates a much more significant speedup. This is mainly due to the direct API calls for matrix computations, which follow sequential execution and stay unparalleled.

E. part 5: Integration program result analysis

Putting everything together, we examine our result by comparing with python's numpy library implementation of PCA and K-Mean, the results are matched. Next step, we run a few tests on the real life image and see the result of image segmentation and evaluate the accuracy by observation.

First of all, we input a simple test case with a phone on the table, we set k value to 2, in order to test the algorithm's ability to distinguish the phone in the picture. The result as shown below.

Figure 8: Original picture; phone on table

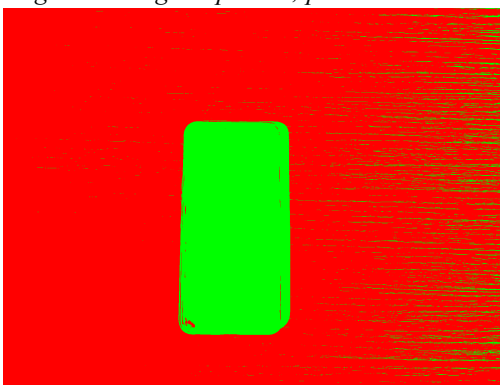

Figure 9: after segmentation; phone on table

As we can see in the result image, the phone has been successfully labeled differently as the table, which is the expected. Next step we added another item and increased the complexity of the testing case and set k value to 3, the result as shown below.

Figure 10: Original picture; phone with opus

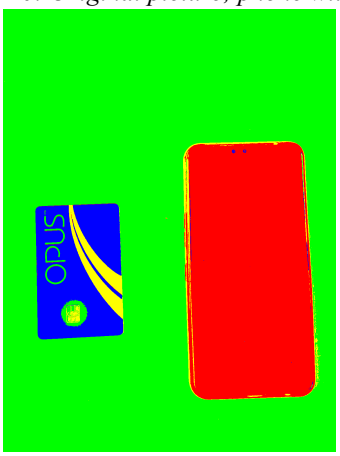

Figure 11: after segmentation; phone with opus

The result of segmentation is still promising and aligns with our expectation. Furthermore, we reduced the color contrast of the image and kept the k value 3 to test the algorithm. The result as shown below.

Figure 12: Original picture; phone with Mac

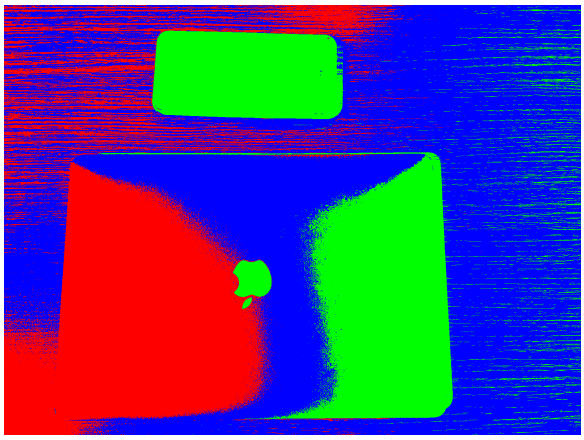

Figure 13: after segmentation; phone with opus

The algorithm is not able to distinguish phones and the Mac. After some more similar testing, we conclude that the algorithm doesn't work well with low color contrast objects due to the random and limited optimization of k-means.

F. part 6: Limitation and future improvement

K-means clustering, while widely used, often encounters challenges in consistently delivering the desired results, primarily due to its sensitivity to initial conditions and data anomalies. The initial placement of centroids is crucial, as random selection can lead to varying clustering outcomes and potentially cause the algorithm to converge to a local minimum rather than the optimal global solution. Furthermore, K-means is particularly sensitive to outliers, which can skew centroid positions and result in unexpected clustering. Another critical factor is the choice of the 'k' value, the number of clusters; an inappropriate selection here can lead to suboptimal clustering, underscoring the importance of accurately determining 'k' for effective clustering.

In terms of future improvements, our team is considering transitioning from K-means to the Gaussian Mixture Model (GMM) for more precise image clustering. GMM offers a more sophisticated approach as it does not assume clusters to be spherical and can adapt to the actual distribution of data. This flexibility allows for more accurate clustering, particularly beneficial in complex image segmentation tasks where the data distribution is not uniform.

However, the parallelization of the Gaussian Mixture Model poses significant challenges and demands extensive development time. GMM's complexity, primarily due to its reliance on Expectation-Maximization (EM) for parameter estimation, makes it more computationally intensive and harder to parallelize efficiently compared to simpler algorithms like K-means. The EM algorithm in GMM involves iterative calculations that are highly dependent on previous steps, complicating the parallelization process.

Despite these challenges, the potential gains in clustering accuracy and segmentation quality make GMM a promising avenue for future research and development. Our team plans to invest time in developing strategies to parallelize

GMM effectively, aiming to harness the power of GPU computing to make this sophisticated model more viable for large-scale and real-time applications in image analysis. This would involve overcoming significant technical hurdles but could ultimately lead to more versatile and accurate clustering methods in the field of computer vision.

For the next step in our project, our team plans to integrate data prefetching into both the PCA and K-means implementations. This advancement aims to enhance the performance of these algorithms by optimizing memory access patterns and reducing latency in data retrieval.

Incorporating data prefetching into K-Mean will target the acceleration of key steps such as finding labels and recalculating the centers. Given k-mean's reliance on large matrix operations, prefetching can significantly speed up these processes by ensuring that the necessary data is readily available in the GPU's cache, minimizing delays caused by data fetching from slower memory sources.

III. CONCLUSION

In conclusion, our project successfully demonstrates the enhancement of image segmentation efficiency through the integration of PCA-optimized K-mean clustering and CUDA parallelization. The sequential implementation of PCA and K-means laid the groundwork for understanding their computational demands. Subsequent parallelization of these algorithms using CUDA significantly reduced computation times, particularly for large data sets and in scenarios requiring real-time analysis. Our experiments with varying thread counts and data prefetching further showcased the potential of GPU acceleration in optimizing complex data processing tasks.

However, we also identified inherent limitations in the K-means algorithm, particularly its sensitivity to initial centroid placement and its susceptibility to outliers. This insight has prompted us to consider advanced clustering techniques like the Gaussian Mixture Model for future work, despite the anticipated challenges in parallelizing such complex models.