Escola SUPERIOR DE TECNOLOGIA E GESTÃO

Final project report

Darya Martsinouskaya - 26610

Master’s program in Informatics Engineering and Internet of Things (IoT)

Course Unit: High Performance Computing

Teacher: José Jasnau Caeiro

Beja. 2025

Contents

Figures

Introduction

The K-means clustering algorithm is one of the most widely used unsupervised machine learning techniques for partitioning datasets into distinct clusters. Its primary goal is to group data points into k clusters based on their similarity, where k is a user-defined parameter. The algorithm works by iteratively assigning data points to the nearest centroid and updating the centroids based on the meaning of the points in each cluster. However, the algorithm's time complexity increases significantly as the size of the dataset and the number of clusters grows, making it computationally expensive for large-scale datasets [1].

K-means is commonly used for tasks like image classification, such as with the MNIST dataset, which contains many images and requires significant computation. To improve its efficiency, the K-means algorithm can be parallelized, allowing it to run faster by using multiple processors or GPUs.

This report focuses on improving the K-means algorithm's performance through three different parallelization strategies:

1. A basic sequential version: the standard, non-parallelized implementation of the K-means algorithm, which serves as a baseline for performance comparison.
2. A parallel multicore version using OpenMP: this version uses the OpenMP parallel programming model to distribute the computation across multiple cores of a CPU, allowing the algorithm to process larger datasets more efficiently.
3. A GPU-accelerated version: this version takes advantage of the massive parallel processing power of GPUs, further speeding up the algorithm by offloading computational tasks to the GPU.

The work involves implementing the K-means algorithm in C++ and comparing the performance of the three versions. The sequential version serves as a baseline, while the parallel implementations are tested for speedup and scalability. The goal is to measure how much faster the parallel versions are compared to the sequential one.

This report is organized as follows: Section 2 explains the K-means algorithm, how it works as well as the description of the problems solved by implementing parallel implementations. Section 3 describes the parallel versions using OpenMP and GPU as well as covers the experimental setup and performance measurements.

1. Problem Description

The K-means algorithm is one of the most popular clustering methods used in data mining and machine learning for partitioning a dataset into k clusters. The primary objective of the algorithm is to assign each data point to a cluster such that the intra-cluster distance (Euclidean distance from data points to their respective centroids) is minimized. The algorithm proceeds iteratively by selecting k initial centroids from the dataset and reassigning points to the nearest centroid while updating the centroid positions. This process continues until the centroids no longer change, indicating convergence. Thus, the algorithm steps are:

1. Choose the number of clusters, k.
2. Randomly generate k clusters and determine the cluster centers or directly generate k random points as cluster centers.
3. Assign each point to the nearest cluster center.
4. Re-compute the new cluster centers.
5. Repeat the two previous steps until some convergence criterion is met (usually that the assignment hasn't changed) [2].

K-means is often employed in image classification tasks, where the goal is to group similar images based on pixel values. This is especially useful in datasets such as MNIST, which consist of grayscale images of handwritten digits. In the context of image classification, the K-means algorithm is used to cluster images into groups that represent different categories, where each category corresponds to a specific digit or type of fashion product.

For instance, the MNIST dataset contains 60,000 training images and 10,000 test images of handwritten digits. Each image is represented by 784 features (28x28 pixels). The challenge of clustering in MNIST arises from the fact that images of the same digit can vary significantly in terms of slant, thickness, and roundness. As a result, it may be necessary to use more than 10 clusters to capture the full variety of digit variations. The labels in the dataset allow for cluster assignment to specific digit values, thus enabling the classification of test images based on the clusters formed during training [1].

The K-means algorithm is widely used for clustering, but it can be slow when dealing with large datasets. This is because the time it takes to process data increases with the number of data points, the number of clusters, and the number of dimensions in the dataset. This becomes a significant issue when working with complex tasks like image classification, where datasets contain high-dimensional data such as images with thousands of pixels.

Parallelizing the K-means algorithm can significantly reduce the time it takes to perform clustering. The core steps of the algorithm can be parallelized to run faster by utilizing multiple processors or GPUs.

Two main parallelization techniques can be applied to improve K-means performance:

1. Multicore parallelization using OpenMP: OpenMP is a framework that enables parallel computing on systems with multiple CPU cores. By using OpenMP, tasks such as assigning points to clusters and calculating the new positions of centroids can be split across multiple processor cores. This allows for much faster execution, particularly with large datasets.
2. GPU-accelerated parallelization: Graphics Processing Units (GPUs) are capable of running many operations in parallel simultaneously, making them perfect for tasks like K-means clustering. By offloading the computation to the GPU, the algorithm can take advantage of thousands of smaller threads, which results in a massive speedup compared to CPU-based execution.

In this work, three versions of the K-means algorithm are developed to compare how parallelization affects performance and efficiency on large datasets such as MNIST:

1. Sequential version: the sequential version of the K-means algorithm follows the basic, unoptimized approach where all steps—assigning points to clusters and updating centroids— are executed one after another. This version serves as a baseline for performance comparison.
2. Parallel multicore version: this version uses OpenMP to split the work across multiple CPU cores. By running these operations in parallel, the algorithm can process large datasets much faster, reducing execution time significantly.
3. Parallel GPU-accelerated version: in this version, OpenMP target offloading is used to run the algorithm on the GPU. This takes advantage of the GPU's capability to handle many threads simultaneously, leading to much faster performance compared to multicore processing. This version is expected to show the greatest speedup, particularly with large and high-dimensional datasets like EMNIST.

By implementing these three versions, the study demonstrates the power of parallel computing in improving the K-means algorithm's efficiency. The performance of each version is compared in terms of execution time, scalability, and classification accuracy. The results highlight how parallelization can make K-means more efficient and scalable for large datasets, ultimately leading to faster clustering and better results.

2. Experimental Part

In this chapter, we present the improvements and optimizations made to the original K-means clustering algorithm, particularly with the aim of enhancing performance and leveraging parallel processing. The optimizations include the use of OpenMP and memory management strategies, resulting in more efficient execution of the algorithm for large datasets such as the MNIST dataset.

In the first variation a basic sequential version was implemented. This is the standard, non-parallelized implementation of the K-means algorithm, which serves as a baseline for performance comparison.

The baseline sequential implementation of the K-means algorithm is structured into several key components. The primary goal is to cluster high-dimensional image data, such as EMNIST, by iteratively assigning data points to the nearest centroid and updating centroids accordingly. Key functions and structure include:

1. Classification of Images (classify_images). This function assigns each data point to the nearest centroid:

- Iterates over all images in the dataset.
- Computes the Euclidean distance between each image and all centroids.
- Assigns the image to the closest centroid.

This is an O(nk) operation, making it computationally expensive for large datasets.

2. Centroid Initialization (initialize_centroids_kmeans_pp). The K-means++ initialization method is used to enhance clustering performance:

- The first centroid is chosen randomly.
- Remaining centroids are initialized based on distance metrics.

This function helps reduce poor initial centroid choices, leading to better convergence.

3. K-means main iteration. The core iterative K-means clustering process consists of:

- Label Assignment: each point is assigned to the nearest centroid.
- Centroid Update: the centroid of each cluster is recalculated based on the mean of assigned points.
- Convergence Check: if centroids remain unchanged, the loop terminates.

The centroid update step is computationally expensive because it involves iterating over all data points and computing averages.

4. Accuracy Calculation (calculate_accuracy). After training, the model's accuracy is evaluated against ground-truth labels. Measures clustering quality, useful for benchmarking optimizations.
5. Execution time is measured using std::chrono provides high-resolution timing data, ensuring accurate performance evaluation.

The K-means algorithm serial implementation with the MNIST dataset. Figure 1 shows the results of execution time measurement for the serial program with different parameters k.

Serial implementation	Execution time [s]
k = 10 training accuracy: 0.01975 test accuracy: 0.0188	838.213
k = 50 training accuracy: 0.232333 test accuracy: 0.0234	1495.33
k = 100 training accuracy: 0.00911667 test accuracy: 0.0082	921.122
k = 200 training accuracy: 0.01915 test accuracy: 0.0192	691.691

Figure 1. Results of testing the serial implementation of the algorithm

Thus, the sequential implementation of the K-means algorithm has some notable strengths. It is simple, easy to understand, and serves as a clear baseline for performance comparison. The use of K-means helps improve initial centroid selection, reducing the likelihood of poor clustering. However, the implementation also has significant limitations. The classification process is computationally expensive due to the O(nk) complexity of distance calculations. The lack of parallelization results in slow execution, especially for large datasets. Additionally, inefficient memory access patterns can lead to frequent cache misses, further slowing down execution.

Despite these limitations, this baseline implementation provides a strong foundation for further optimizations. The next step involves exploring parallelization strategies using OpenMP to improve performance on multicore processors, followed by GPU acceleration to further enhance computational efficiency.

Parallelization using OpenMP was implemented in the second variation. One of the key improvements was the introduction of OpenMP for parallelizing the main loops in the K-means algorithm. Specifically:

- Classifying images (assigning labels to data points): in the second variation of the algorithm, the classification step was parallelized using the #pragma omp parallel for directive. This enables the assignment of labels to all data points in parallel, significantly reducing the time spent on this task. The workload for each data point, which involves computing the Euclidean distance to each centroid, can be computed independently for each data point, making it a perfect candidate for parallelization.

- Accuracy calculation: another part of the algorithm that benefits from parallelization is the accuracy calculation. By utilizing the #pragma omp parallel for reduction(+:correct) directive, the algorithm computes the accuracy across multiple threads, reducing the computation time significantly.

Figure 2 shows the results of testing the parallel implementation of the K-means algorithm using OpenMP library. To calculate speedup, the execution time of the sequential implementation with the execution time of the parallel (either multicore or GPU-accelerated) implementation were compared. The general formula for speedup is: S = execution time of sequential variation / execution time of parallel (GPU) variation.

Parallel implementation	Execution time [s]	Multicore speedup [s]
k = 10 training accuracy: 0.0110167 test accuracy: 0.0095	100.817	8.31420296
k = 50 training accuracy: 0.01705 test accuracy: 0.0166	48.6627	30.7284635
k = 100 training accuracy: 0.0124333 test accuracy: 0.0135	20.0596	45.9192606
k = 200 training accuracy: 0.01375 test accuracy: 0.0137	40.2045	17.2043179

Figure 2. Results of testing the parallel implementation of the algorithm using OpenMP library

Comparing the serial and parallel implementations of the algorithm, several key observations were made regarding both execution time and accuracy across different values of k.

Execution Time:

a. The parallel implementation demonstrates a significant reduction in execution time compared to the serial implementation. The speedup values ranged from 8.31 for k=10 to 45.92 for k=100, indicating substantial performance improvements with parallelization.
b. As the value of k increases, the execution time for both the serial and parallel versions increases, as expected, due to the larger number of neighbors being considered during computation.
c. Notably, the parallel implementation continues to perform efficiently, although the speedup for k=200 decreased slightly to 17.20, which could be attributed to diminishing returns from parallelization as the problem size grows.

Accuracy:

a. Accuracy trends show typical behavior when increasing k: training accuracy decreases as k grows, reflecting the model's transition from overfitting (for small k) to underfitting (for large k). Similarly, test accuracy is lower for larger k values, indicating that the model struggles to generalize effectively with larger neighborhoods.
b. The slight decrease in accuracy from the serial to parallel implementation is expected and likely due to parallelization overhead or data handling differences across threads, though the differences in accuracy are minimal.

Speedup:

a. The speedup achieved with parallelization is generally impressive, especially for smaller values of k (e.g., k = 10 and k = 50). The highest speedup is observed for k = 100 with a value of 45.92, demonstrating the potential for significant time savings when using parallelization with larger datasets or computational tasks.
b. However, the speedup for k = 200 drops, indicating that beyond a certain threshold, parallelization may not provide as much of a benefit, and other factors such as communication overhead or system limitations could affect performance.

Final Observations:

a. The results confirm that parallelization yields substantial time savings, especially for computationally intensive tasks. However, the diminishing returns observed for higher k values suggest that there are inherent limits to how much speedup can be achieved, particularly when the workload increases further.
b. For future work, further tuning of the parallel implementation could optimize performance for larger values of k, and more tests could be conducted to explore the behavior of the algorithm with even larger datasets or different parallelization strategies.

In summary, the second parallel implementation significantly outperforms the serial implementation in terms of execution time, with speedup values scaling well for smaller to moderate values of k. However, as k increases, there are diminishing returns, which warrants further analysis and optimization to maximize the benefits of parallel computing.

In the third variation, an important optimization was implemented by transforming the data and centroid matrices into one-dimensional arrays using the flatten() function. This allows better memory management and improved access patterns, which is especially beneficial for parallel execution. Key changes are:

- Data flattening: instead of working with a 2D matrix of data points and centroids, both the train_data and centroids are flattened into 1D vectors. This allows for better memory locality and more efficient memory access, crucial when working with large datasets.
- Use of arrays for OpenMP Offloading: the flattening step is followed by copying the data into raw arrays (float*), which are then used in the OpenMP offloading section. This is crucial for the performance boost as OpenMP offloading can better handle raw arrays, leading to improved parallel execution on modern hardware.

The third variation also employs OpenMP target offloading for the centroid update step. By using #pragma omp target teams distribute parallel for with offloading to the GPU or other accelerators, this optimization makes use of parallel computing resources more effectively. This results in faster centroid computation, especially when the dataset size is large.

In this step, centroids are updated in parallel across multiple threads, and memory is managed by using raw arrays and mapping them to the parallel context with the map(to) and map(from) directives. This optimization ensures that the centroid updates are computed in parallel and are efficiently transferred to and from the device memory.

Memory management was another focus area for optimization in the third variation. After each parallel operation, dynamically allocated memory (raw arrays) is properly freed using delete[]. This prevents memory leaks and ensures that the program can handle larger datasets without running into memory issues.

The performance improvements mentioned above directly contribute to a reduction in the execution time of the K-means algorithm. Specifically, parallelization has allowed for a substantial decrease in the runtime, particularly when processing the large MNIST dataset, with time reductions in both the classification and centroid updating phases. This makes the algorithm more scalable, enabling it to handle datasets with millions of data points and centroids.

The combination of parallelization, memory optimizations, and offloading computations to GPUs or accelerators (via OpenMP) leads to significant performance improvements in the K-means algorithm. In practice, this allows for faster execution, making it feasible to apply the K-means algorithm to larger datasets and in real-time scenarios.

Figure 3 shows the results of execution time measurement for the parallel GPU accelerated program with different parameters k.

Parallel implementation	Execution time [s]	GPU speedup [s]
k = 10 training accuracy: 0.00996667	1.26322	663.552667
k = 50 training accuracy: 0.0271333	1.23969	1206.21284
k = 100		

training accuracy: 0.0121167	1.2539	734.60563
k = 200 training accuracy: 0.0251667	1.24277	556.572013

Figure 3. Results of testing the parallel GPU accelerated implementation of the algorithm

Comparison of the GPU-accelerated implementation and the initial sequential implementation of the K-means algorithm reveals significant performance improvements in terms of execution time, though both implementations show similar challenges in accuracy. Key points are:

Execution time: the GPU-accelerated version achieves remarkable speedup, reducing the execution time by over 500 times compared to the sequential version. For example, while the sequential implementation takes up to 1495.33 seconds for k = 50, the GPU-accelerated version completes the task in approximately 1.24 seconds. This demonstrates the efficiency of GPU parallelization in handling computationally intensive tasks like K-means clustering.

Accuracy: despite the drastic reduction in execution time, the accuracy results of the GPU-accelerated version remain low, similar to the sequential implementation. The training and test accuracies for both implementations are underwhelming, with values ranging between 0.009 and 0.023. This indicates that while GPU acceleration significantly speeds up the process, it does not necessarily improve the algorithm's ability to effectively cluster the data.

Thus, while the GPU-accelerated implementation offers substantial performance gains in terms of speed, the low accuracy observed in both sequential and GPU implementations highlights the need for further optimization of the algorithm. This could involve experimenting with different values of **k**, enhancing data preprocessing, or refining the clustering process to improve the model's ability to fit the data accurately.

To sum up, we highlighted the performance optimizations made to the K-means clustering algorithm. The most significant improvements include the use of OpenMP for parallelization and offloading, along with efficient memory management through flattening and dynamic memory allocation. These changes result in a faster, more scalable implementation of the K-means algorithm that can be used effectively on large datasets.

Conclusion

In this report, we have explored and implemented three variations of the K-means clustering algorithm to improve its performance: a sequential baseline version, a parallel multicore version using OpenMP, and a GPU-accelerated version utilizing OpenMP offloading. The performance of these implementations was tested on large datasets, such as the MNIST dataset, and various metrics, including execution time, speedup, and accuracy, were evaluated.

The sequential implementation provided a solid foundation for comparison, demonstrating the basic functioning of the K-means algorithm. However, it was significantly slower compared to the parallel versions, especially as the dataset size and the number of clusters increased. As expected, the sequential version struggled with large datasets due to its O(nk) time complexity.

The parallel multicore version using OpenMP achieved notable speedups across a range of cluster sizes. The execution time was significantly reduced, and the speedup values ranged from 8.31 for small datasets to 45.92 for medium-sized ones. While the accuracy remained relatively stable, the execution time reductions confirmed the effectiveness of parallelizing the classification and accuracy calculation tasks across multiple CPU cores. However, diminishing returns were observed for very large cluster sizes ($k = 200$), indicating that beyond a certain point, the overhead of parallelization and system limitations hindered further improvements.

The GPU-accelerated version, which involved offloading centroid updates to the GPU using OpenMP target offloading, demonstrated the most significant improvements in execution time. This approach capitalized on the massive parallel processing power of GPUs, significantly reducing the runtime for large datasets and improving the overall scalability of the K-means algorithm. Memory management optimizations, such as flattening the data and centroid matrices, contributed to improved memory access patterns, further enhancing the GPU acceleration benefits.

Overall, the parallelization strategies greatly enhanced the performance of the K-means algorithm, particularly for medium to large datasets. While the GPU-accelerated version outperformed both the sequential and multicore versions, the diminishing returns observed for higher cluster counts ($k = 200$) suggest that further fine-tuning of the parallelization approach is needed to maximize performance at larger scales.

Future improvements could include exploring advanced parallelization techniques like CUDA for better GPU control, optimizing memory management with shared memory in GPU implementations, and testing other parallel processing frameworks for improved scalability. Additionally, experimenting with larger datasets could reveal where parallelization becomes less effective and offer insights for optimizing K-means clustering for large-scale tasks.

Bibliographic References

1. Anja Tanović and Vuk Vranjković. “Implementation of parallel K-means algorithm for image classification using OpenMP and MPI libraries”. In: 2024 Zooming Innovation in Consumer Technologies Conference (ZINC). 2024, pp. 54–59. doi: 10.1109/ZINC61849.2024.10579351.
2. Reze Farivar, Daniel Rebolledo, Ellick Chan, Roy Campbell. “A Parallel Implementation of K-Means Clustering on GPUs”. In: DBLP. 2008. URL: https://www.researchgate.net/publication/221134288_A_Parallel_Implementation_of_K-Means_Clustering_on_GPUs.