Comparison of Sequential and Parallel K-Means Clustering Algorithms

Francesco Gigli
francesco.gigli@edu.unifi.it

October 9, 2024

Abstract

This project explores the optimization of the K-Means clustering algorithm through three distinct implementations: a sequential version, a parallelized version using OpenMP, and a GPU-accelerated version using CUDA. The goal is to evaluate the performance of each approach in processing large datasets, comparing execution times and scalability. The study focuses on how parallelization and GPU resources can improve computational efficiency, addressing the challenges posed by increasing data complexity.

1 Introduction

We introduce the fundamental concepts of the project by addressing the algorithm and the choice of values used.

1.1 Algorithm

K-Means is a widely used unsupervised learning algorithm designed to address clustering problems. The algorithm classifies a dataset into a predefined number of clusters, denoted as k, which is specified *a priori*. Initially, cluster centroids are chosen, and each data point is assigned to the nearest centroid, forming clusters. The algorithm then iteratively updates the centroids by recalculating their positions based on the mean of the points in each cluster. This process repeats until the centroids stabilize, achieving convergence.

In this project, data points are represented in a two-dimensional (2D) space, defined by random (x, y) coordinates. The clusters are represented by their centroids, and the algorithm follows these key steps:

1. **Point Generation**: Generate N random points in a 2D space.
2. **Centroid Initialization**: Initialize k centroids that represent the clusters, with each cluster initially containing only its centroid.
3. **Assignment Step**: For each point, compute the distance to every centroid and assign the point to the nearest cluster.
4. **Update Step**: Recalculate the centroid's position by averaging the coordinates of the points assigned to each cluster.
5. **Iteration**: Repeat steps 3 and 4 until the centroids no longer move significantly, or until a predefined stopping criterion (such as a maximum number of iterations) is met.

While the K-Means algorithm generally converges quickly, it does not always guarantee finding the global optimum. The quality of the final clustering result depends heavily on the number and initial positioning of the centroids.

1.2 Parameter Choices

The K-Means algorithm requires the specification of several parameters, and this section outlines the assumptions and decisions made regarding these parameters:

- **Number of Clusters (k)**: The number of clusters must be specified beforehand. In this project, we evaluate several fixed values of k to analyze their impact on the model's performance.
- **Centroid Initialization**: The initial centroids are selected by randomly choosing k points from the dataset. This randomness can lead to different clustering results in different runs, as the initial positions of the centroids significantly influence the final clusters.
- **Distance Metric**: We employ the Euclidean distance to measure the proximity of points to centroids.
- **Stopping Criterion**: To prevent the algorithm from running indefinitely, we set a maximum number of iterations.

By carefully selecting these parameters, we aim to optimize the performance and efficiency of the K-Means algorithm across different implementations.

2 Implementation

In this section, we describe the structure of the project and the implementation of the three versions of the K-Means algorithm: Sequential, OpenMP Parallel (using both critical and atomic synchronization), and CUDA. We begin by introducing the primary classes used to represent data points and clusters:

2.1 Classes

The core components of the project are the `Point` and `Cluster` classes, which model the essential elements of the K-Means algorithm. These classes are designed to be efficient and compatible with both CPU and GPU environments.

2.1.1 Point Class

The `Point` class represents an individual data point in a two-dimensional space. It has the following main attributes:

- **Coordinates** (`coord_x`, `coord_y`): These store the x and y positions of the point in 2D space.
- **Cluster ID** (`id_cluster`): An integer that identifies the cluster to which the point is assigned.

The class provides methods for accessing and modifying these attributes:

- `get_x()`: Returns the x-coordinate of the point.
- `get_y()`: Returns the y-coordinate of the point.
- `get_cluster_id()`: Returns the ID of the cluster to which the point is currently assigned.
- `set_id()`: Assigns the point to a specific cluster by setting the cluster ID.

This structure ensures that each point can be effectively managed and linked to a specific cluster in the context of clustering algorithms.

2.1.2 Cluster Class

The `Cluster` class is designed to manage the centroid of a cluster and the points assigned to it. It maintains three main attributes:

- **Centroid Coordinates** (`x_coord`, `y_coord`): These coordinates represent the current position of the cluster's centroid.
- **Accumulated Coordinates** (`new_x_coord`, `new_y_coord`): These are the cumulative coordinates of all points assigned to the cluster during an iteration, which are used to update the centroid's position.
- **Size** (`size`): This is the count of points currently assigned to the cluster.

The class provides the following essential methods to efficiently manage the cluster's operations:

- `add_point()`: Accumulates the coordinates of each newly assigned point to the cluster and updates the total point count.
- `add_point_atomic()`: Performs atomic updates to accumulate point coordinates and increment the point count, reducing synchronization overhead in parallel environments.

- `update_centroid()`: Computes the new centroid position by averaging the accumulated coordinates, ensuring the centroid accurately reflects the current composition of the cluster.
- `reset_values()`: Resets the accumulated coordinates and point count, preparing the cluster for a new iteration.

2.2 Sequential Implementation

The sequential version of the K-Means algorithm follows the standard procedure:

- **Initialization**: The algorithm starts by generating N random points in a 2D space and initializing k centroids. These centroids are randomly selected from the generated points.
- **Assignment Step**: For each point, the algorithm calculates the squared Euclidean distance between the point and each centroid. The point is then assigned to the cluster with the nearest centroid. This is achieved by iterating over all the points and clusters using a nested loop.

```
for (int i = 0; i < num_points; ++i) {
    double min_dist =
    ↪ squared_euclidean_distance(points[i],
    ↪ clusters[0]);
    int nearest_cluster = 0;
    for (int j = 1; j < num_clusters; ++j) {
        double dist =
    ↪ squared_euclidean_distance(points[i],
    ↪ clusters[j]);
        if (dist < min_dist) {
            min_dist = dist;
            nearest_cluster = j;
        }
    }
    points[i].set_id(nearest_cluster);
}
```

- **Update Step**: After all points are assigned to clusters, the centroids are updated by calculating the mean of the coordinates of all points assigned to each cluster. The centroid coordinates are reset and then recalculated based on the assigned points.
- **Iteration**: The algorithm repeats the assignment and update steps for a fixed number of iterations (typically 20), or until the centroids no longer change significantly. Each iteration involves recalculating the centroids and reassigning points to clusters.

2.3 Parallel Implementation with OpenMP

The OpenMP version of the K-Means algorithm leverages multi-core CPU processing to parallelize two main steps: point assignment and centroid updates. The overall structure of the algorithm remains the same as the sequential version, with parallelization applied to improve performance.

2.3.1 Point Assignment with OpenMP

The loop that assigns points to the nearest centroids is parallelized using OpenMP. Each thread computes the distances for a subset of points, reducing execution time.

```
#pragma omp parallel for
for (int i = 0; i < num_points; ++i) {
    double min_dist =
    ↪ squared_euclidean_distance(points[i],
    ↪ clusters[0]);
    int nearest_cluster = 0;
    for (int j = 1; j < num_clusters; ++j) {
        double dist =
    ↪ squared_euclidean_distance(points[i],
    ↪ clusters[j]);
        if (dist < min_dist) {
            min_dist = dist;
            nearest_cluster = j;
        }
    }
    points[i].set_id(nearest_cluster);
}
```

2.3.2 Centroid Updates with OpenMP Critical and Atomic Synchronization

When updating the centroids, synchronization is necessary to prevent race conditions. Two methods are employed: **critical sections** and **atomic operations**.

Critical Sections The `#pragma omp critical` directive ensures that only one thread can execute the enclosed code block at a time. This prevents multiple threads from simultaneously updating the same cluster, but can introduce significant overhead due to locking.

```
#pragma omp parallel for
for (int i = 0; i < num_points; ++i) {
    #pragma omp critical
    clusters[points[i].get_cluster_id()]
        .add_point(points[i]);
}
```

Atomic Operations The `#pragma omp atomic` directive allows for atomic updates to individual variables without locking the entire code block. This reduces synchronization overhead and improves performance, especially in high-contention scenarios.

```
#pragma omp parallel for
for (int i = 0; i < num_points; ++i) {
    clusters[points[i].get_cluster_id()]
        .add_point_atomic(points[i]);
}
```

2.3.3 K-Means Algorithm with OpenMP

The parallel K-Means algorithm using OpenMP is implemented with both critical and atomic synchronization methods. Below is an outline of the parallel algorithm:

- **Assignment Step**: Parallelize the assignment of points to the nearest cluster.
- **Reset Clusters**: Reset accumulated coordinates and point counts for each cluster.
- **Accumulate Points**: Use either critical sections or atomic operations to accumulate point coordinates and counts.
- **Update Centroids**: Recalculate centroids based on accumulated data.
- **Iteration**: Repeat for a fixed number of iterations.

2.4 Parallel Implementation with CUDA

The CUDA version of the K-Means algorithm offloads multiple computationally intensive tasks—such as point assignment, accumulation of point coordinates, and centroid updates—to the GPU. This allows the algorithm to fully utilize the parallel processing capabilities of modern GPUs, significantly improving performance.

2.4.1 CUDA Kernel Functions

Squared Euclidean Distance A device function computes the squared Euclidean distance between a point and a cluster centroid.

```
__device__ double
    ↪ squared_euclidean_distance_device(const
    ↪ Point& pt, const Cluster& cl) {
    double dx = pt.get_x() - cl.get_x_coord();
    double dy = pt.get_y() - cl.get_y_coord();
    return dx * dx + dy * dy;
}
```

Assign Points to Nearest Cluster Each thread assigns a single point to the nearest cluster by calculating the squared Euclidean distance and selecting the closest centroid.

```
__global__ void assign_points(Point* points, Cluster*
    ↪ clusters, int num_points, int num_clusters) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < num_points) {
        double min_dist =
    ↪ squared_euclidean_distance_device(points[i],
    ↪ clusters[0]);
        int nearest_cluster_id = 0;
        for (int j = 1; j < num_clusters; ++j) {
            double dist =
    ↪ squared_euclidean_distance_device(points[i],
    ↪ clusters[j]);
            if (dist < min_dist) {
                min_dist = dist;
                nearest_cluster_id = j;
            }
        }
        points[i].set_id(nearest_cluster_id);
    }
}
```

Accumulate Points in Clusters Each thread adds the coordinates of its assigned point to the corresponding cluster. Atomic operations are used to safely update the shared data and prevent race conditions.

```
__global__ void accumulate_points(Point* points,
    ↪ Cluster* clusters, int num_points) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < num_points) {
        int cluster_id = points[i].get_cluster_id();
        clusters[cluster_id].
        atomic_add_to_new_x_coord(points[i].get_x());
        clusters[cluster_id].
        atomic_add_to_new_y_coord(points[i].get_y());
        clusters[cluster_id].atomic_increment_size();
    }
}
```

Update Centroids Each thread updates the centroid of a cluster by averaging the accumulated point coordinates. Once the centroids are updated, their values are reset for the next iteration.

```
__global__ void update_centroids(Cluster* clusters,
    ↪ int num_clusters) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    if (i < num_clusters && clusters[i].get_size() >
    ↪ 0) {
        clusters[i].set_x_coord(clusters[i].
        get_new_x_coord() / clusters[i].get_size());
        clusters[i].set_y_coord(clusters[i].
        get_new_y_coord() / clusters[i].get_size());
        clusters[i].reset_values();  // Reset for the
    ↪ next iteration
    }
}
```

2.4.2 K-Means Algorithm with CUDA

The CUDA implementation of the K-Means algorithm follows these steps:

- **Initialization**: Memory for the points and clusters is allocated on the GPU using `cudaMalloc`, and the data is transferred from the host (CPU) to the device (GPU) using `cudaMemcpy`.
- **Assignment Step**: The `assign_points` kernel assigns each point to the nearest cluster based on the squared Euclidean distance.
- **Accumulation Step**: The `accumulate_points` kernel accumulates the coordinates of points assigned to each cluster. Atomic operations ensure correct accumulation in a parallel environment.
- **Update Step**: The `update_centroids` kernel updates the centroids of the clusters by averaging the accumulated coordinates.
- **Iteration**: These steps (assignment, accumulation, and update) are repeated for a fixed number of iterations (typically 20). During each iteration, data is processed entirely on the GPU to minimize transfer overhead.
- **Finalization**: After completing the iterations, the updated cluster data is copied back to the host using `cudaMemcpy`, and the GPU memory is freed using `cudaFree`.

2.5 Optimization Techniques Implemented in the Code

Several optimization techniques have been implemented in the parallel K-Means code to enhance performance:

- **Thread Management**: In OpenMP, the number of threads is set dynamically using `omp_set_num_threads` to balance performance and resource usage.
- **Atomic Operations**: Atomic operations are used in both OpenMP (e.g., `add_point_atomic`) and CUDA (`accumulate_points` kernel) to safely update shared variables and avoid race conditions.
- **Memory Management**: In CUDA, `cudaMalloc` and `cudaMemcpy` handle efficient memory allocation and data transfers, minimizing latency by keeping data on the GPU during iterations.
- **Load Balancing**: OpenMP's `#pragma omp parallel for` ensures an even workload distribution among threads, preventing bottlenecks and maximizing CPU core usage.

These optimizations contribute to the significant performance gains of the parallel implementations compared to the sequential version.

3 Experiments and Results

3.1 Experimental Setup

The experiments were conducted on a system with the following specifications:

- **Operating System**: Microsoft Windows 11 Pro
- **CPU**: AMD Ryzen 7 5800X (8 cores, 16 logical processors)
- **GPU**: NVIDIA GeForce RTX 3080

Synthetic datasets were generated with sizes of 100,000; 250,000; 500,000; and 1,000,000 points. The number of clusters (k) tested were 5, 10, and 20. The primary objective was to assess and compare the performance of the sequential, OpenMP parallel (utilizing both critical and atomic sections), and CUDA implementations in terms of execution time and scalability.

Table 1: Comparison of Sequential and OpenMP (Critical) Implementations

		Sequential		OpenMP (Critical)	
Points	**Clusters**	**Total Time (s)**	**Iteration Time (s)**	**Total Time (s)**	**Iteration Time (s)**
100,000	5	2.21	0.1105	0.17309	0.00865
100,000	10	2.50	0.1250	0.21171	0.01059
100,000	20	3.00	0.1500	0.30877	0.01544
250,000	5	4.56	0.2280	0.32261	0.01613
250,000	10	5.23	0.2615	0.42733	0.02137
250,000	20	6.12	0.3060	0.60936	0.03047
500,000	5	7.34	0.3670	0.63144	0.03157
500,000	10	8.12	0.4060	0.83830	0.04192
500,000	20	9.56	0.4780	1.20900	0.06045
1,000,000	5	14.50	0.7250	1.24938	0.06247
1,000,000	10	16.34	0.8170	1.67609	0.08380
1,000,000	20	18.90	0.9450	2.60162	0.13008

3.2 Comparison of Sequential and OpenMP (Critical) Implementations

Table 1 compares the sequential implementation with the OpenMP parallel implementation using critical sections. The OpenMP version consistently outperforms the sequential approach across all dataset sizes and cluster counts. For example, with 1,000,000 points and 20 clusters, OpenMP (Critical) reduces the total execution time from 18.90 seconds (Sequential) to 2.60 seconds, demonstrating a significant improvement.

This performance gain is due to OpenMP's ability to distribute the workload across multiple CPU cores, efficiently handling parallel operations. However, the use of critical sections introduces some overhead, as synchronization is required for shared resource access, which slightly limits the maximum achievable speedup.

3.3 Comparison of OpenMP Critical and Atomic Implementations

Following the comparison of sequential and OpenMP (Critical) implementations, we now contrast OpenMP (Critical) with atomic operations. As shown in Table 2, the atomic implementation consistently performs better, especially with larger datasets. For instance, with 1,000,000 points and 20 clusters, the atomic version completes in 2.09 seconds compared to 2.60 seconds for the critical version.

This improvement is due to the reduced synchronization overhead of atomic operations, which allow updates without locking entire code sections, making parallel execution more efficient.

3.3.1 Thread Scalability and Speedup Analysis

Figure 1 illustrates the speedup achieved by the OpenMP (Atomic) implementation as the number of threads increases. The speedup generally increases with the number of threads, reaching a peak performance around 12–14 threads. Beyond this point, the speedup starts to plateau and slightly decrease. This behavior is likely due to the diminishing returns of adding more threads, where the overhead of thread management and synchronization begins to outweigh the benefits of parallel execution.

The speedup curve demonstrates that OpenMP with atomic operations effectively utilizes multiple cores to enhance performance, especially up to a moderate number of threads. However, as the number of threads exceeds the optimal point, resource contention and increased synchronization overhead limit further performance gains.

		Critical		Atomic	
Points	**Clusters**	**Total Time (s)**	**Iteration Time (s)**	**Total Time (s)**	**Iteration Time (s)**
100,000	5	0.17309	0.00865	0.13234	0.00662
100,000	10	0.21171	0.01059	0.19273	0.00964
100,000	20	0.30877	0.01544	0.23883	0.01194
250,000	5	0.32261	0.01613	0.30281	0.01514
250,000	10	0.42733	0.02137	0.33243	0.01662
250,000	20	0.60936	0.03047	0.51124	0.02556
500,000	5	0.63144	0.03157	0.65221	0.03261
500,000	10	0.83830	0.04192	0.69693	0.03485
500,000	20	1.20900	0.06045	0.98261	0.04913
1,000,000	5	1.24938	0.06247	1.25425	0.06271
1,000,000	10	1.67609	0.08380	1.40475	0.07024
1,000,000	20	2.60162	0.13008	2.09404	0.10470

Table 2: Comparison of OpenMP Implementations: Critical vs. Atomic

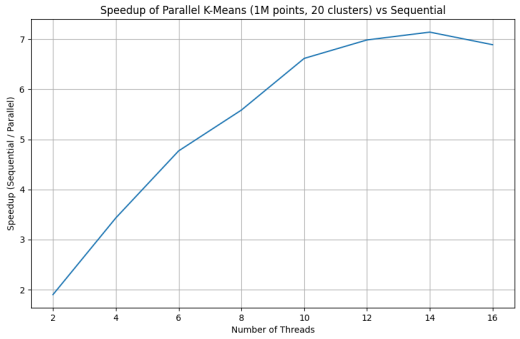

Figure 1: Speedup of OpenMP (Atomic) K-Means with Varying Thread Counts (1M points, 20 clusters)

3.4 Comparison of OpenMP Critical and CUDA Implementations

Table 3 compares the OpenMP implementation using critical sections with the CUDA implementation. The results clearly show that CUDA consistently outperforms OpenMP across all dataset sizes and cluster counts. For example, with 1,000,000 points and 20 clusters, CUDA completes the algorithm in just 0.04453 seconds, compared to 2.60162 seconds for OpenMP (Critical), demonstrating a significant speedup.

This substantial improvement is primarily due to the GPU's ability to leverage massive parallelism, executing thousands of threads concurrently. Unlike CPUs, which are typically limited to a few dozen threads in OpenMP (e.g., 4 to 16 threads), GPUs can handle thousands of threads simultaneously, significantly reducing the time required to assign points to clusters. CUDA efficiently exploits this architecture, particularly in the point assignment phase of the K-Means algorithm, where each point can be processed independently.

Moreover, CUDA benefits from reduced synchronization overhead. In OpenMP, critical sections require locking mechanisms to ensure only one thread updates shared resources at a time, introducing delays. Even with atomic operations in OpenMP, some synchronization overhead persists. CUDA uses atomic operations directly on the GPU hardware, minimizing these delays. The ability to process a vast number of threads simultaneously and the reduced synchronization make CUDA highly efficient, especially for large datasets, where OpenMP shows diminishing returns as the number of threads increases due to synchronization bottlenecks.

Scalability tests indicate that while OpenMP scales well

		OpenMP (Critical)		CUDA	
Points	**Clusters**	**Total Time (s)**	**Iteration Time (s)**	**Total Time (s)**	**Iteration Time (s)**
100,000	5	0.17309	0.00865	0.06808	0.00340
100,000	10	0.21171	0.01059	0.00525	0.00026
100,000	20	0.30877	0.01544	0.00534	0.00027
250,000	5	0.32261	0.01613	0.01534	0.00077
250,000	10	0.42733	0.02137	0.01031	0.00052
250,000	20	0.60936	0.03047	0.01140	0.00057
500,000	5	0.63144	0.03157	0.03000	0.00150
500,000	10	0.83830	0.04192	0.01955	0.00098
500,000	20	1.20900	0.06045	0.02210	0.00111
1,000,000	5	1.24938	0.06247	0.05547	0.00277
1,000,000	10	1.67609	0.08380	0.03852	0.00193
1,000,000	20	2.60162	0.13008	0.04453	0.00223

Table 3: Comparison of OpenMP (Critical) and CUDA Implementations

initially, adding more threads beyond a certain point (8-12 threads) provides limited benefits, as synchronization costs start outweighing the advantages of additional parallelism. In contrast, CUDA continues to perform efficiently with increasing dataset sizes, maintaining low execution times even with more clusters and larger datasets.

4 Visualization and Result Analysis

To provide a more intuitive understanding of the K-Means clustering results, we generated a graphical representation of the clusters along with their centroids. The figure below (Figure 2) illustrates how the data points are grouped into clusters after 20 iterations of the algorithm.

Each data point is colored based on its assigned cluster, and the final centroid of each cluster is marked with a black star. This allows us to visually assess how well the clustering algorithm has separated the data into distinct groups. From the plot, we can observe that the K-Means algorithm effectively grouped the data points into clear clusters, with centroids positioned in the center of each cluster. This graphical representation helps validate the clustering results by showing that the algorithm minimized the distance between points and their respective centroids. The centroids' positions also reflect the average location of all points within their cluster, confirming that the K-Means algorithm iteratively refined these positions over time.

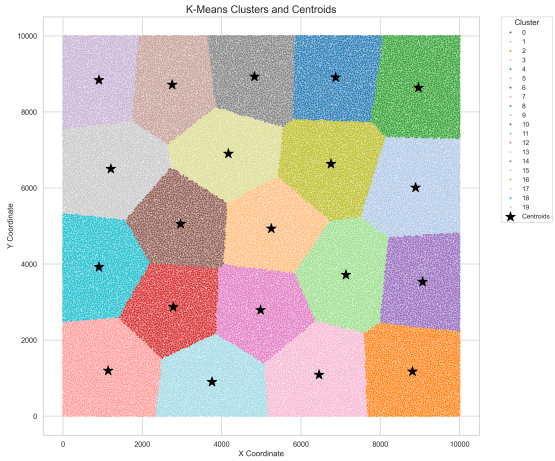

Figure 2: Visualization of K-Means clusters with centroids marked as black stars.

The use of distinct colors for each cluster makes it easier to differentiate between clusters and understand the relationships between data points. The centroids, represented by the larger black stars, signify the calculated center of

each cluster, which plays a key role in the clustering process as the anchor points for data point assignment in every iteration of the algorithm.

5 Conclusion

The results of this study demonstrate that both OpenMP and CUDA offer significant performance improvements over the sequential implementation of the K-Means clustering algorithm. OpenMP effectively utilizes CPU multi-core processing, particularly with atomic operations, to parallelize both point assignment and centroid updates. However, as the number of threads increases, OpenMP experiences diminishing returns due to synchronization overhead, particularly in the critical section version.

CUDA, on the other hand, provides a far greater speedup due to its ability to handle massive parallelism by executing thousands of threads concurrently. The reduced synchronization overhead in CUDA, coupled with the efficient atomic operations implemented directly in GPU hardware, allows for superior scalability and performance, particularly on large datasets.

In summary, while OpenMP can be advantageous in multi-core CPU environments for medium-sized datasets, CUDA remains the optimal choice for large-scale K-Means clustering due to its efficiency in managing parallelism and minimizing synchronization delays. These findings emphasize the importance of selecting the appropriate parallelization technique based on the size of the dataset and the available hardware resources.