2D K-means clustering with GPU

Plator Rama
plator.rama@edu.unifi.it

Abstract

In this work, I will study the CUDA K-means clustering. I will compare the sequential version (running on CPU) of the program with the parallel version (running on GPU). I will describe the code I've used and the obtained speed up.

1. Introduction

This project aims to explore the performance benefits of GPU-accelerated computing using the k-means clustering algorithm implemented with CUDA. The primary objective is to compare the computational efficiency of the parallel GPU version against a baseline sequential implementation. The k-means algorithm is a popular clustering technique that partitions data points into a predefined number of clusters based on their proximity to randomly initialized centroids.

To evaluate the performance of the GPU implementation, several tests were conducted by varying key parameters:

- The **number of points**: Monitoring execution times as the number of data points increases.
- The **number of centroids**: Analyzing how the algorithm scales with different numbers of centroids.
- The **number of threads per block**: Investigating the effect of thread configuration on GPU performance.
- The **number of iterations**: Measuring the impact of the number of iterations on execution time and convergence.

These tests provide insights into the scalability and efficiency of the k-means algorithm when offloaded to the GPU, highlighting the potential speedups that can be achieved through parallel processing.

1.1. Hardware Setup Used

The experiments for the k-means algorithm were conducted on a system with the following hardware configuration:

- **GPU**: NVIDIA GeForce GTX 1660 Super, equipped with 6 GB of GDDR6 memory and built on the Turing architecture (TU116). This GPU has 1,408 CUDA cores and operates at a base clock speed of 1,530 MHz, with a boost clock of up to 1,785 MHz. It supports CUDA Compute Capability 7.5, which was utilized for parallelizing the k-means computations.
- **CPU**: AMD Ryzen 5 3600X, a 6-core, 12-thread processor based on the Zen 2 microarchitecture. It features a base clock speed of 3.8 GHz, with a maximum boost frequency of 4.4 GHz. The processor has a total L3 cache of 32 MB and supports PCIe 4.0, providing fast data transfer between the CPU and GPU. This CPU was used for benchmarking the sequential and parallel implementations of the algorithm on the CPU.

2. Code Explanation

In this section, we will see all the code involved in this comparison. The goal is run k-means clustering algorithm for n iteration.

2.1. Data Structure

In this project, two main data structures were used to represent the information needed for the k-means clustering algorithm: `Points` and `Centroids`. Both structures follow the "Array of Structures" (AoS) approach, which organizes data into separate arrays, improving access and parallel processing on the GPU by leveraging memory coalescence.

2.2. Struct `Points`

The `Points` structure handles the information related to the points to be clustered. The key fields include:

- `float *x`: an array of values representing the x-coordinates of the points.
- `float *y`: an array storing the y-coordinates of the points.
- `int *closestCentroid`: an array that tracks the closest centroid for each point, updated during each iteration of the algorithm.

This design allows efficient access to the points' coordinates and assigns each point to its nearest centroid, facilitating parallel computation on the GPU.

2.3. Struct `Centroids`

The `Centroids` structure represents the centroids of the clusters and includes the following fields:

- `float *x`: an array for the x-coordinates of the centroids.
- `float *y`: an array for the y-coordinates.
- `int *pointCount`: an array that keeps track of the number of points assigned to each centroid.
- `float *sumX`: an array accumulating the sum of the x-coordinates of points associated with each centroid, used to update the centroid's position.
- `float *sumY`: an array accumulating the sum of the y-coordinates.

This structure enables the centroids to be easily updated during each iteration of the algorithm based on the points assigned to them.

2.4. Sequential Algorithm

The sequential version of the k-means clustering algorithm was implemented to serve as a baseline for comparison with the parallel version. The sequential algorithm follows a classic iterative approach to cluster points around centroids based on their proximity. It continuously updates the centroids until convergence or until a set number of iterations is reached.

The key steps of the sequential k-means algorithm are:

1. **Initialization**: At the start of each iteration, the accumulators that store the sum of the points' coordinates and the count of points assigned to each centroid are reset.
2. **Assigning Points to the Nearest Centroid**: For each point, the algorithm computes the Euclidean distance to each centroid and assigns the point to the closest centroid. The accumulators for the assigned centroid are then updated with the point's coordinates.
3. **Updating the Centroids**: Once all points have been assigned to their nearest centroids, the algorithm recalculates the position of each centroid by averaging the coordinates of the points assigned to it.
4. **Iteration**: This process repeats until the centroids converge or the specified number of iterations is reached.

This algorithm runs sequentially on the CPU and processes each point and centroid one at a time. As the number of points and centroids grows, the computation time increases linearly. The goal of this sequential version is to establish a baseline for comparing the performance improvements introduced by the parallel GPU version of the algorithm.

2.5. Parallel Algorithm with CUDA

The parallel implementation of the k-means clustering algorithm on the GPU uses CUDA to leverage the massive parallelism of modern graphics processing units. This version divides the work across multiple threads to speed up both the assignment of points to centroids and the update of centroid positions. Two main CUDA kernels are employed: one for assigning points to the nearest centroids and another for updating the centroids' positions based on the assigned points.

1. **Kernel 1: Assigning Points to Centroids**
   The `assignCentroids` kernel computes the Euclidean distance between each point and all centroids, determining the closest centroid for each point. The accumulators that store the sum of the point coordinates and the number of points assigned to each centroid are updated using atomic operations to avoid race conditions.
2. **Kernel 2: Updating Centroids**
   Once all points have been assigned to centroids, the `updateCentroids` kernel calculates the new positions of the centroids by averaging the coordinates of the points assigned to them. The accumulators are then reset in preparation for the next iteration.

The algorithm runs iteratively until a given iteration tolerance is met.

This CUDA implementation dramatically improves performance compared to the sequential version by taking advantage of the parallel processing power of the GPU. The number of threads and blocks is configured based on the problem size, ensuring that each point and centroid is processed concurrently.

3. Tests

4. Summary

This section summarizes the key findings from the performance tests conducted:

- **Speedup with Number of Points**: his section shows that speedup increases with more points, particularly at higher iteration counts, indicating good scalability of the GPU implementation;

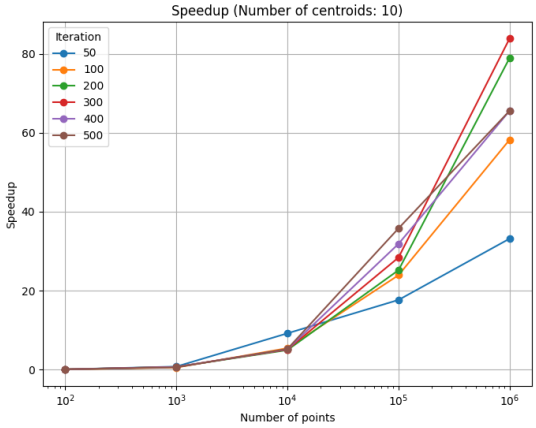

Figure 1. Speedup vs. Number of Points for Different Iteration Counts (Number of Centroids: 10)

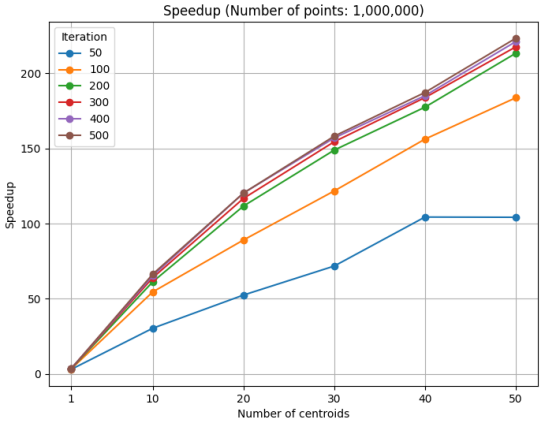

Figure 2. Speedup as a function of the number of centroids for 1,000,000 points.

- **Speedup with Number of Centroids**: Similar to the previous test, speedup improves as the number of centroids increases, especially for more iterations, demonstrating the advantages of parallelization;

- **Efficiency**: This section reveals that efficiency is highest with fewer threads per block, decreasing as thread count rises, highlighting the need for an optimal balance to maximize performance.

4.1. Speedup Analysis: Number of Points

In this test, we measured the speedup of the k-means algorithm on the GPU compared to the CPU as the number of data points increased, while keeping the number of centroids constant at 10. The graph below displays six lines representing different iteration counts: 50, 100, 200, 300, 400, and 500.

As shown in the graph, the speedup increases significantly with the number of points, especially when the number of points exceeds 10^4. The speedup becomes more pronounced as the iteration count grows, with the highest iteration counts (300, 400, and 500) reaching a speedup of over 80x for 10^6 points. This result highlights the efficiency of GPU parallelization, particularly in scenarios where large datasets and multiple iterations are involved. The lower iteration counts (50 and 100) show a slower rise in speedup, as they require less computation, making the benefits of parallelism less pronounced for smaller datasets.

4.2. Speedup Analysis: Number of Centroids

In this test, the speedup was analyzed by varying the number of centroids while keeping the number of points constant at 1,000,000. The graph in Figure 2 shows the performance trends for six different iteration counts: 50, 100, 200, 300, 400, and 500 iterations. As the number of centroids increases, the speedup improves steadily across all iteration configurations.

From the graph, it's evident that as the number of centroids increases, the algorithm benefits more from parallelization, leading to higher speedups. However, beyond approximately 40 centroids, the speedup curve begins to plateau for the lower iteration configurations (e.g., 50 iterations), indicating that there might be diminishing returns in terms of performance gain for certain configurations.

This test further highlights that the number of centroids plays a significant role in the overall performance, and increasing the number of centroids leads to better utilization of the GPU, especially for larger iteration counts.

4.3. Efficiency

Figure 3 shows the efficiency trend as a function of the number of threads per block used during the execution of the k-means algorithm on the GPU. In this test, the number of points was kept constant at 1,000,000 and the number of centroids at 10.

Efficiency is defined as the ratio between the speedup achieved and the number of threads per block. From the figure, we observe a decreasing trend in efficiency as the number of threads per block increases. Specifically, efficiency is maximized for a small number of threads per block (around 32-64) but tends to decrease drastically as the number of threads per block grows.

This behavior is due to the fact that having too many threads per block prevents optimal utilization of the GPU's parallelization capabilities, leading to resource saturation and, consequently, a reduction in overall efficiency.

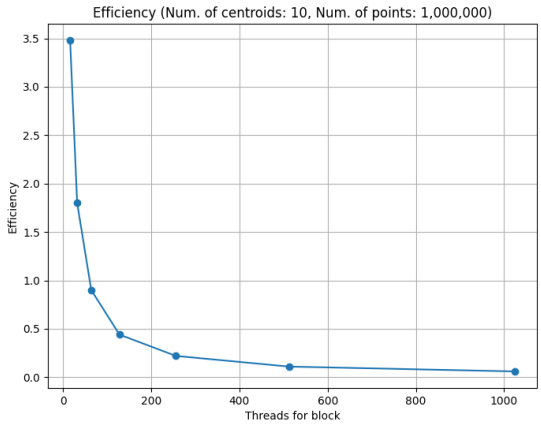

Figure 3. Efficiency as a function of threads per block (Num. of centroids: 10, Num. of points: 1,000,000).

5. Conclusion

The results of this project demonstrate the significant performance benefits of implementing the k-means clustering algorithm on a GPU using CUDA. By leveraging the parallel processing capabilities of the GPU, we observed substantial reductions in execution time compared to the sequential implementation on the CPU. The tests, which varied the number of points, centroids, threads per block, and iterations, provided valuable insights into how different configurations impact performance.

The most notable improvements were observed as the number of data points increased, highlighting the effectiveness of parallel computation for handling large datasets. Similarly, the optimal thread configuration per block played a crucial role in maximizing the GPU's efficiency, underlining the importance of properly tuning CUDA parameters for each specific problem.

In conclusion, this project confirms the potential of GPU computing to accelerate computationally intensive tasks like k-means clustering, particularly in scenarios with large datasets and high computational demand.