Parallel Computing course

a.a. 2021–2022

Comparative Analysis of K-Means Algorithm

using Sequential C++ and Parallel OpenMP, CUDA implementation modes

Leonardo Paroli

for the 2021-2022 Parallel Computing course by Prof.Bertini

Abstract. In this work we give an analysis of the implementation of the K-Means algorithm using different approaches, including sequential implementation in C++, and parallel implementations using OpenMP and CUDA. The main objective is to evaluate the performance of these implementations in terms of speed and efficiency, comparing execution times and highlighting key differences among the various implementation modes, providing proofs for selecting the optimal approach based on specific usage contexts.

Key words. K Means - Parallel Computing - Speedup - Parallelization - C++ - CUDA - OpenMP

1. Context

The K-Means algorithm is one of the primary data analysis and clustering tools used in various fields, including machine learning, image analysis, and bioinformatics. The main goal of the algorithm is to partition a dataset into homogeneous groups, known as clusters: data within the same cluster are similar to each other and different from data in other clusters.

The implementation of K-Means involves repeated iterations to assign data points to corresponding clusters, recalculating the cluster centers based on the assignments at the end of each iteration.

The sequential implementation of K-Means serves as the foundation for building parallel versions, providing a a benchmark for evaluating the speedup brought about by parallel versions in terms of performance and scalability.

Parallel implementations of the K-Means algorithm in OpenMP and CUDA modes focus instead on optimizing execution by utilizing the multicore architectures of CPUs and the parallel processing units of GPUs, respectively.

---

2. Test structure and hardware

The goal is to measure the execution time of the K-means clustering algorithm in both sequential and parallel modes. Due to the nature of the algorithm, measuring execution time on a dataset in a non-arbitrary manner requires structuring the test according to certain assumptions.

First and foremost, all execution modes of the algorithm operate on the same dataset of three-dimensional points (which are not modified during the algorithm's execution). The initial centroids, selected during dataset generation, are applied to the first iteration of the algorithm in each programming mode to avoid the "lucky shot" scenario, where one execution of the algorithm with a given "lucky" set of initial centroids would immediately (or much much faster) converge to the solution.

The first step for the comparative analysis is to develop the algorithm in various modes and assess its effectiveness, i.e., whether it indeed solves the clustering problem. Given a number N of points and a number K of clusters, the datasets are generated by selecting K points with coordinates x, y, z within the range 0, 1000 respectively. Additionally, N/K points are generated according to a Gaussian distribution centered around the selected point. This approach allows visual verification of the algorithm's results since K-Means favors globular cluster formations. The point generation process also includes a noise function, introducing enough sparsity in the data to prevent K-means from converging too rapidly.
After generating the data and initial centroids, they are saved in a text file named "initialization.txt" and can be visualized using a dedicated Python script employing the Matplotlib plotting library (as shown in figures 1,2).

The measurement of execution time starts when the algorithm receives the dataset and begins its iteration, continuing until the result is returned in the form of clusters containing assigned dataset points.
The measurement is performed using std:: chrono::high_resolution_clock, with the result reported in milliseconds.
Once the algorithm is executed on the generated dataset, in both sequential and parallel modes, the results are computed and recorded in a text file named "results.txt". These results can be graphically visualized using a dedicated Python script, as mentioned earlier.

The analysis was conducted on a laptop with the following hardware specifications:

- **Processor**:AMD Ryzen 7 5700U,
- 8 cores, 1.8GHz
- **L2 Cache**: 4MB
- **L3 Cache**: 8MB
- **RAM**: 16 GB , 3200 MHz
- **GPU**: NVIDIA GeForce RTX 3050 Laptop GPU
- **VRAM**: 4 GB
- 8.6 computational power on the NVIDIA scale.

All tests have been developed and ran on Windows 10, using Clion-2023 prior to the integration of the Visual Studio 2022 toolchain for building, compiling, debugging and running the program.
Tests ran in parallel mode have been conducted through the use of Nvidia CUDA Toolkit 12.1.

3. Sequential implementation

The next step was writing the actual code for the K-means algorithm in sequential mode, using C++ Object Oriented Programming. An algorithm overview:

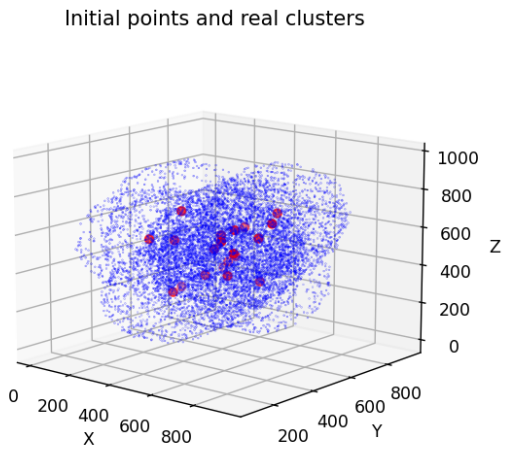

Fig. 1. Plot of the initial dataset points (in blue) and the center, or real centroids, of their distributions.

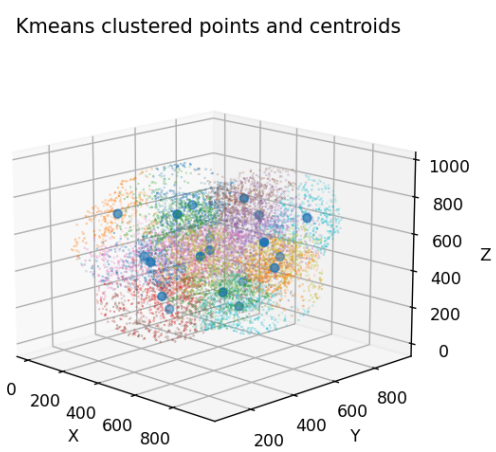

Fig. 2. Plot of the clustered points (each different color is assigned to a different cluster) after using K-means.

- The code starts by calculating the initial sum of squared errors (SSE) for the provided dataset and the initial centroids. The sum of squared errors is calculated as follows:

$$SSE = \sum_{i=1}^{N} \sum_{j=1}^{K} w_{ij} \|x_i - c_j\|^2 \quad (1)$$

  where N is the number of data points, K is the number of clusters, x_i represents the i-th data point, c_j represents the centroid of the j-th cluster, and w_{ij} is an indicator variable indicating whether data point x_i belongs to cluster c_j.
- It then initializes arrays for cluster assignments and cluster sizes.
- The K-Means iteration loop begins.
- Within each iteration, points are assigned to the nearest cluster centroid, evaluating their distance through the calculation of the Euclidean distance as follows:

$$\sqrt{(x_P - x_Q)^2 + (y_P - y_Q)^2 + (z_P - z_Q)^2} \quad (2)$$

  where x_P, y_P, and z_P are the coordinates of point P, and x_Q, y_Q, and z_Q are the coordinates of point Q.

- New centroids are computed based on the assigned points.
- The SSE is updated for the new centroids.
- When the improvement between the last SSE and the current one is small enough (a threshold of 0.01 is used for this), the iteration stops.
  To avoid unstable local minimums, an iteration number threshold has been applied aswell.
- Finally, the code assigns each point to its respective cluster based on the computed assignments.

The points in the dataset were introduced as an array of structures Point, with each Point struct holding the x,y,z coordinate of the data point.

4. OpenMP implementation

The first approach was oriented towards parallelizing the sequential algorithm while maintaining its object-oriented structure, by applying pragma directives to enable OpenMP parallelization.
The use of classes, instead of primitive types and arrays, influenced the performance of the parallelized blocks, with performances heavily impacted by the presence of multiple object instantiations.
Abandoning the object oriented programming for the sake of performances, the parallel OpenMP mode was later revised, using primitive types, particularly the use of an array of indexes of length equal to the number of points, to assign each point to the index of one of the clusters, greatly speeding up the parallelization.

The initial SSE calculation is performed in a parallel mode, along with the point assignment step, which involves calculating distances between points and centroids: this section benefits greatly from parallelization, especially when dealing with a large number of points and clusters.
This part involves accumulating the coordinates of points assigned to each cluster, by using thread-local accumulators that are later combined in a critical section.

Parallelization to the point assignment process has been made through the use of pragma omp directives to initiate parallel execution over the dataset points, where each point is evaluated to determine the nearest cluster centroid, by calculating the Euclidean distance between the point and each cluster centroid. Atomics are used to prevent concurrent access conflicts that could occur due to multiple threads simultaneously modifying the same location over the cluster sizes array when adding a point.

Following the point assignment operation, the SSE update involves calculating distances and updating the SSE values for each point, by assigning points to clusters (in parallel), with a critical section used to avoid race conditions when updating the cluster information.

An extensive empirical experimentation has been made to find the best combination of data structures for speeding up the process, as described in the Results chapter later on this report.

5. CUDA Implementation

As done in the OpenMP implementation of K-means, the first approach was to simply apply parallelization through the use of CUDA methods and kernel calls to improve the speed of the clusterization process. Using an object oriented programming influenced the total speedup greatly, as the instancing of non-primitive types and arrays with the use of many allocations slowed down the process quite dramatically.
Using primitive types, performances were better, but there was more that could've been done: for example, through the conversion of Array of Structure data into Structure of Arrays, the kernel calls could exploit memory bursts to load data efficiently during the parallel execution of the clusterization.
On a successive iteration, CUDA shared memory was also used to speed up the process even more: a small, high-speed memory space shared among threads within a block, with significantly enhanced memory access speed compared to global memory, to efficiently load the data needed for the operation.

The assignPointsToClusters_SOA CUDA kernel is a pivotal component of the k-means clustering algorithm as it computes the assignment for all data points to their nearest clusters based on the Euclidean distance to cluster centroids, parallelizing this process across multiple threads: each thread corresponds to a unique data point, identified by its thread index.

To ensure correctness and consistency, __syncthreads() is used to synchronize threads within a block, guaranteeing that all threads have finished loading their data into shared memory before the distance calculations begin.

In summary, the kernel for assigning points using a SoA effectively utilizes shared memory and parallelism to streamline the assignment of data points to clusters, reducing redundant memory operations, and capitalizing on the GPU's parallel processing power.

The code also exploits parallelization to initially compute and then update the SSE within each iteration of the K-means algorithm, using the same techniques to efficiently load and process the data.

6. Results

Using the sequential implementation of K-Means as a baseline, tests have been conducted to determine the performance of the OpenMP parallelized code and CUDA parallelized code.
The following tables help visualize the performance improvement when using all the precedently described parallization techniques: the first ones (1 and 2) have as values the mean execution time of the algorithm in the sequential and parallel modes, in milliseconds.

It should be noted that due to the nature of the K-means algorithm, variance between results is over the 35% of the mean value, even when repeating the tests. This is due to the fact that at the start of each test, the dataset and the initial centroids are randomly selected, leading to edge cases of "fast" and "slow" executions.

Table 1. Performance Results in milliseconds with fixed number of clusters (20) and varying number of points for sequential and parallel OpenMP implementations of Kmeans

Algorithm ms / # Points	100	1000	10000	100000	250000	500000	750000
Kmeans Sequential	3	84	2579	29527	75783	303971	538470
Kmeans Parallel OMP	8	66	397	3969	9867	38851	67968

Table 2. Performance Results in milliseconds with fixed number of clusters (20) and varying number of points for sequential and parallel CUDA implementations of Kmeans

Algorithm ms / # Points	100	1000	10000	100000	250000	500000	750000
Kmeans Sequential	3	104	1830	46781	109800	285705	546069
Kmeans Parallel CUDA	7	19	163	2772	6056	8294	14287

The resulting speedup in table 3 confirm that the parallel OpenMP version of the K-Means algorithm can properly exploit the full extent of the multiple cores of the CPU, using threads to parallelize the process, growing more and more efficient as the number of points increase. Lower speedups with fewer points are expected: the time to prepare the dataset and threads, and then accumulate and reduce the results of different threads can lead to a visible overhead when the execution time is brief.

Table 3. Speedup Results between the parallel implementations for the K-means algorithm

Speedup / # Points	100	1000	10000	100000	250000	500000	750000
Parallel OMP	0.375	1.2727	6.4962	7.4394	7.6804	7.8240	7.9224
Parallel CUDA	0.4286	5.4737	11.2270	16.8763	18.1308	34.4472	38.2214

The same can be said for the CUDA version, where the speedup is even greater at higher number of points.
In the following pages, we repeated the tests by keeping the number of points fixed to 10000, and then varying the number of clusters (tables number 4,5,6).

Table 4. Performance Results in milliseconds with fixed number of points (10000) and varying number of clusters for sequential and parallel OpenMP implementations of Kmeans

Algorithm# Clusters	10	15	20	25	30	35	40
Kmeans Sequential	946	1959	2095	2879	3157	4343	3504
Kmeans Parallel OMP	245	438	379	489	496	663	467

Table 5. Performance Results in milliseconds with fixed number of points (10000) and varying number of clusters for sequential and parallel CUDA implementations of Kmeans

Algorithm# Clusters	10	15	20	25	30	35	40
Kmeans Sequential	1147	1272	2345	2612	4008	4708	4992
Kmeans Parallel CUDA	119	108	188	154	202	232	234

Table 6. Speedup Results between the parallel implementations for the K-means algorithm

Speedup / # Clusters	10	15	20	25	30	35	40
Parallel OMP	3.8612	4.4726	5.5277	5.8875	6.3649	6.5505	7.5032
Parallel CUDA	9.6387	11.7778	12.4734	16.9610	19.8416	20.2931	21.3333

More tests have been conducted with the previous iterations of the code, especially when experimenting with shared and static memory in CUDA implementations, or when switching from an Object Oriented Programming to primitive types, but results were worse performance wise.
An example in the table 7 below, the static CUDA version is slower than the one using shared memory.

Table 7. Speedup Results

Speedup / # Points	100	1000	10000	100000	250000	500000	750000
Parallel CUDA Static	0.4253	4.6463	10.2342	12.4392	16.1232	26.2673	30.9224
Parallel CUDA Shared	0.4286	5.4737	11.2270	16.8763	18.1308	34.4472	38.2214

7. Conclusions

The comparative analysis made on the various parallel versions of the K-Means algorithm show that using parallelization methods, be it with OpenMP or CUDA, lead to a great performance increase in the execution time, while maintaining the same efficacy of the sequential version. OpenMP parallelization is bound by the number of cores and threads available to the hardware, while CUDA streaming processors can exploit the higher number of threads and computational cores to efficiently execute the algorithm iterations over a large dataset, with gains increasing exponentially to the number of points.
As a side note, GPUs have usually lower memory to work with than the actual main memory a CPU can utilize to do its computations, a detail not to be overlooked when dealing with large datasets, that may require much more careful handling and partitioning if processed by using CUDA kernels.