Sapienza University of Rome

Multicore

K-Means: Parallel Clustering Algorithm

Barbalata Radu Ionut

Matriculation Number: 2002688

February 10, 2025

Contents

1 K-Means Clustering Algorithm

The k-means algorithm is a widely used clustering technique that partitions a given dataset into k clusters. It is an iterative algorithm that alternates between two main steps:

1. **Assignment step:** Each data point is assigned to the nearest cluster center based on a chosen distance metric, typically the Euclidean distance.
2. **Update step:** The cluster centers are recomputed as the mean of all data points assigned to each cluster.

These two steps repeat until a stopping criterion is met. The most common termination conditions include:

- The cluster assignments no longer change between iterations.
- The change in cluster centroids between consecutive iterations falls below a predefined threshold.
- A maximum number of iterations is reached.

The algorithm aims to minimize the within-cluster variance, defined as:

$$J = \sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||^2$$

where C_i represents the set of points belonging to cluster i, and μ_i is the centroid of that cluster.

While k-means is simple and efficient, it has some limitations, such as sensitivity to the initial placement of centroids and difficulty in handling non-convex clusters. Various optimizations and parallel implementations, such as MPI, OpenMP, and CUDA, can improve its efficiency for large datasets.

2 CUDA Optimization Techniques

In this section, we explore different approaches and optimization techniques for implementing CUDA kernels. The goal is to start with a simple implementation and then progressively refine it for performance improvements. We will begin with the first kernel.

2.1 Initial Kernel: assignDataToCentroidsKernel

The initial kernel implementation is straightforward, with each thread calculating the distance between data points and centroids, updating the class map, and performing an atomic operation when a change is detected.

Algorithm 1 Kernel: assignDataToCentroidsKernel

```
Require: Data points to classify $D$, centroids $C$, class map $M$ of $D$, changes counter $g$
 1: Initialize a variable $minDist$ to store the minimum distance
 2: Initialize a variable $nearestCentroid$ to store the closest centroid
 3: for each centroid $c$ in $C$ do
 4:     Calculate the distance between $D[i]$ and centroid $c$
 5:     if the calculated distance is less than $minDist$ then
 6:         Update $minDist$ to the new distance
 7:         Update $nearestCentroid$ to the current centroid $c$
 8:     end if
 9: end for
10: Update the class map $M[i]$ to the $nearestCentroid$
11: if the class of $D[i]$ has changed then
12:     Perform an atomic addition to $g$
13: end if
```

2.1.1 Identifying the Bottleneck

The primary bottleneck in this kernel is the atomic add operation, which is shared across all blocks. This operation can cause contention, slowing down the kernel significantly. To address this, one optimization approach is to use a shared ‘changes‘ variable, denoted as sh_g, within each block. This way, each block performs atomic operations on its local copy of sh_g, reducing the need to access the global memory for every update.

2.1.2 Further Optimizations

Another potential optimization is to share the centroids across threads. However, the large size of the centroid matrix ($K \times$ dimPoints) may cause memory issues in shared memory. Additionally, centroids are often sparse, meaning not every data point contributes to every centroid. Therefore, careful consid-

Algorithm 2 Kernel: assignDataToCentroidsKernel with Shared Changes

```
Require: Data points to classify $D$, centroids $C$, class map $M$ of $D$, changes
    counter $g$
 1: __shared__ $sh_g$
 2: Initialize $minDist$ and $nearestCentroid$
 3: for each centroid $c$ in $C$ do
 4:     Calculate the distance between $D[i]$ and centroid $c$
 5:     if the calculated distance is less than $minDist$ then
 6:         Update $minDist$ to the new distance
 7:         Update $nearestCentroid$ to the current centroid $c$
 8:     end if
 9: end for
10: Update the class map $M[i]$ to the $nearestCentroid$
11: if the class of $D[i]$ has changed then
12:     Perform an atomic addition to $sh_g$
13: end if
14: if thread.index == 0 then
15:     AtomicAdd to $g$ using $sh_g$
16: end if
```

eration is required when deciding which data to store in shared memory and when it is beneficial to do so.

2.2 Kernel 2: updateCentroidsKernel

The second kernel, ‘updateCentroidsKernel‘, updates centroids based on the assigned data points. Each thread corresponds to a single data point, and the kernel performs atomic operations to sum data points to auxiliary centroids and count the number of points per class.

Algorithm 3 Kernel: updateCentroidsKernel

```
Require: Data points $D$, centroids $C$, class map $M$ of $D$, auxiliary centroids $A$, points per class $P$
1: Retrieve the class $c$ of the current data point
2: Atomically add the data point to the $c$-th auxiliary centroid
3: Atomically increment $P[c]$ by 1
```

While not much can be optimized here, particularly due to the atomic operations on shared memory and the need to ensure data consistency, it’s important to note the potential limitations of using shared memory for complex operations like centroid updates.

2.3 Kernel 3: atomicMaxFloat

The final kernel is designed to compute the maximum distance for centroid updates, and it is optimized by using atomic operations. The goal is to minimize the computational overhead by leveraging shared memory for efficient maximum distance computation.

Algorithm 4 Kernel: atomicMaxFloat

```
Require: Address $a$, float value $v$
1: Compute the distance between centroid $C[tId]$ and auxiliary centroid $A[tId]$
2: Atomically update the maximum distance
```

2.3.1 Atomic Max Implementation

To optimize the atomic maximum operation, we can use shared memory to reduce the number of global memory accesses. Here’s how the ‘atomicMaxFloat‘

function can be implemented using atomic compare-and-swap (CAS):

Algorithm 5 atomicMaxFloat kernel

```
Require: Pointer to address $address$, float value $val$
1: $address\_as\_int \leftarrow$ cast($address, $int)
2: $old \leftarrow *address\_as\_int$
3: $assumed \leftarrow 0$
4: repeat
5:     $assumed \leftarrow old$
6:     $old \leftarrow$ atomicCAS($address\_as\_int, assumed,$ float_to_int(max($val,$ int_to_float($assumed$))))
7: until assumed = old
8: return int_to_float(old)
```

While alternative implementations, such as using reduction, could be explored, they may not provide significant performance gains. As we will observe, this kernel has little impact on scalability, with the ‘assignDataToCentroidsKernel‘ being the most computationally expensive operation in the overall process.

3 K-Means with MPI

Building upon the high-level description of the K-Means algorithm and its sequential implementation, we now explore how MPI can be leveraged to achieve parallelism. The primary approach involves **data parallelism**, where the dataset is partitioned, and each MPI process is assigned a subset of the data. Each process computes the nearest centroids for its assigned data points and collaborates with others to update the centroids globally.

The MPI implementation of K-Means follows this structure:

Algorithm 6 K-Means Algorithm with MPI

```
1: repeat
2:     Local computation: Assign centroids to data points
3:     Local computation: Compute auxiliary centroids and count points per class
4:     Global synchronization: Perform MPI_Reduce to aggregate centroid updates
5:     Global synchronization: Perform MPI_Reduce to sum all changes $g$
6:     Update centroids: Compute new centroids based on aggregated results
7: until convergence criteria is met
```

Beyond partitioning the dataset, additional considerations include memory allocation for distributed data structures and result retrieval.

3.1 Hybrid Parallelization with OpenMP

Given that the `assignCentroidsToData` function accounts for over 95% of the total computation, integrating OpenMP within this function can significantly improve performance. The following is its parallelized implementation:

Algorithm 7 Parallel Assign Data to Centroids

```
Require: Data array $D$, centroids array $C$, class map $M$, number of points $numPoints$, dimensionality $dimPoints$, number of centroids $K$, pointer to changes counter $g$
1: $localChanges \leftarrow 0$
2: #pragma omp parallel reduction(+:localChanges)
3: {
4: threadId $\leftarrow$ omp_get_thread_num()
5: Start time $s \leftarrow$ omp_get_wtime()
6: for each point $i = 0$ to $numPoints - 1$ do
7:     $minDist \leftarrow +\infty$, $newClass \leftarrow -1$
8:     for each centroid $k = 0$ to $K - 1$ do
9:         Compute squared Euclidean distance between $D[i]$ and $C[k]$
10:        if distance $< minDist$ then
11:            Update $minDist$ and assign $newClass \leftarrow k$
12:        end if
13:    end for
14:    if $M[i] \neq newClass$ then
15:        $M[i] \leftarrow newClass$
16:        Increment $localChanges$
17:    end if
18: end for
19: printf("Thread: %d, Time: %lf", threadId, omp_get_wtime() - s)
20: }
21: $g \leftarrow localChanges$
```

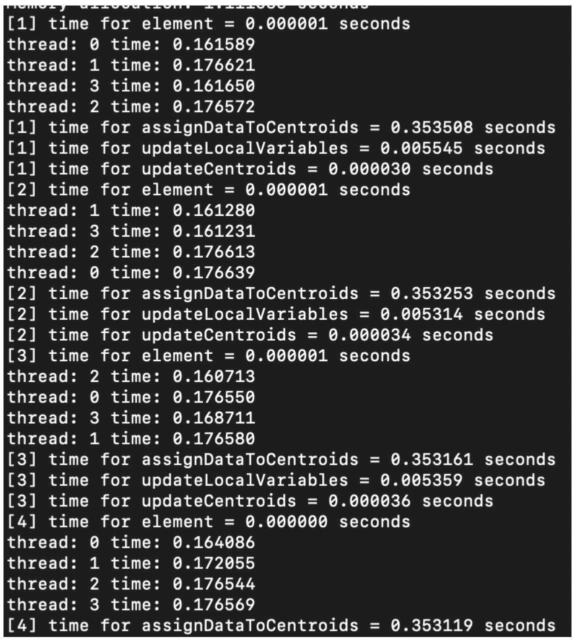
3.2 Scalability Challenges

In this implementation, each thread handles a subset of the data assigned to its MPI process. One potential issue is **false sharing**, where multiple threads modify adjacent memory locations, leading to cache inefficiencies. However, in our case, false sharing is expected to have minimal impact since the computation is dominated by **distance calculations** rather than memory writes (except for the reduction step).

A possible optimization involves caching shared data in private variables before updating shared memory, though implementing this efficiently is non-trivial.

3.3 Performance Bottlenecks

Despite OpenMP integration, scalability remains an issue. To diagnose the problem, we analyzed execution times:



```
[1] time for element = 0.000001 seconds
thread: 0 time: 0.161589
thread: 1 time: 0.176621
thread: 3 time: 0.161650
thread: 2 time: 0.176572
[1] time for assignDataToCentroids = 0.353508 seconds
[1] time for updateLocalVariables = 0.005545 seconds
[1] time for updateCentroids = 0.000030 seconds
[2] time for element = 0.000001 seconds
thread: 1 time: 0.161280
thread: 3 time: 0.161231
thread: 2 time: 0.176613
thread: 0 time: 0.176639
[2] time for assignDataToCentroids = 0.353253 seconds
[2] time for updateLocalVariables = 0.005314 seconds
[2] time for updateCentroids = 0.000034 seconds
[3] time for element = 0.000001 seconds
thread: 2 time: 0.160713
thread: 0 time: 0.176550
thread: 3 time: 0.168711
thread: 1 time: 0.176580
[3] time for assignDataToCentroids = 0.353161 seconds
[3] time for updateLocalVariables = 0.005359 seconds
[3] time for updateCentroids = 0.000036 seconds
[4] time for element = 0.000000 seconds
thread: 0 time: 0.164086
thread: 1 time: 0.172055
thread: 2 time: 0.176544
thread: 3 time: 0.176569
[4] time for assignDataToCentroids = 0.353119 seconds
```

Figure 1: Observed Execution Time Distribution

As shown in Figure 1, the **execution time appears to be the sum of all thread times** instead of benefiting from parallel speedup. This suggests a potential issue such as: - **False sharing**, although no excessive memory writing (aside from reduction) is present. - **Hardware limitations**, such as running on a cluster with single-core CPUs (unlikely but possible).

Since the other MPI functions have a negligible impact on execution time, further optimization efforts will focus on enhancing the **scalability of MPI + OpenMP**. A fully optimized version will be presented during the oral exam. The issue was related to how I measured time. Using clock() results in a time value that represents the sum of the times across all cores.

4 Results and Performance Analysis

In this section, we evaluate the performance of our **CUDA** and **MPI+OpenMP** implementations of the K-Means algorithm. We analyze execution times, compute speedups, and visualize the benefits of parallelism across different configurations of **MPI processes** and **OpenMP threads**.

4.1 Execution Time and Speedup Analysis

We will analyze the speedups and execution times of the different implementations.

For the CUDA implementation, each configuration was executed 20 times using the following command:

```
./kmeans <input_file> 20 500 0 0
```

where:

- `20` is the number of clusters,
- `500` is the maximum number of iterations,
- `0` represents the number of changes,
- `0` is the minimum distance threshold.

This setup ensures that the algorithm runs for 500 iterations with the given input file.

For the MPI+OpenMP version, we tested the following combinations:

- Scaling on threads [(4, 1), (4, 2), (4, 4), (4, 8), (4, 16), (4, 32)]
- Scaling on MPI processes [(1, 4), (2, 4), (4, 4), (8, 4), (16, 4)]

Since the number of combinations is higher, we executed each configuration **10 times** instead of 20 (also some one just one considering some problems with the clusters).

The sequential version followed the same approach as the CUDA version, running **20 iterations per execution**.

The method used for the execution can be seen in the file: `cluster_run.py`

4.1.1 Execution Time Comparison (CUDA vs. Sequential)

Table 1 illustrates the execution times for **sequential**, **CUDA**, and **cudaV2** across various datasets. The substantial reduction in execution time highlights the effectiveness of GPU acceleration.

Model	**100D2**	**100D2X2**	**100DX4**	**100DX8**
Sequential	158.16	316.43	633.70	1268.90
CUDA	3.09	6.87	14.53	29.84
cudaV2	2.68	6.77	14.55	29.79

Table 1: Execution time (in seconds) for Sequential, CUDA, and CUDA V2 implementations.

4.1.2 Speedup Analysis for CUDA Implementations

To further quantify the performance improvements, **Table 2** presents the speedup achieved by `cuda` and `cudaV2` compared to the sequential baseline. Speedup is computed as:

$$\text{Speedup} = \frac{\text{Execution Time (Sequential)}}{\text{Execution Time (Parallel)}} \tag{1}$$

A higher speedup indicates greater efficiency. As shown, both CUDA implementations significantly outperform the sequential approach.

Model	**100D2**	**100D2X2**	**100DX4**	**100DX8**
CUDA	51.06	46.04	43.59	42.51
cudaV2	59.26	46.84	43.55	42.59

Table 2: Speedup of CUDA implementations relative to the sequential model.

4.1.3 Scalability Analysis of MPI+OpenMP Implementations

To evaluate the scalability of our MPI+OpenMP implementation, we analyze and speedups by varying:

- The number of OpenMP threads while keeping the number of MPI processes fixed.
- The number of MPI processes while keeping the number of OpenMP threads fixed.

Model	100D2	100D2X2	100DX4	100DX8
Sequential	158.16	316.43	633.70	1268.90
1_4	40.1	81.0	162.06	331.71
2_4	21.37	41.41	85.92	165.05
4_4	10.54	22.12	43.31	86.97
8_4	6.67	11.91	23.89	46.61
16_4	4.22	6.24	13.91	26.04
4_1	40.58	80.33	160.52	323.37
4_2	20.47	40.58	81.02	166.57
4_4	10.54	22.12	43.31	86.97
4_8	6.83	11.73	21.13	47.25
4_16	3.52	7.01	13.63	25.81
4_32	2.19	3.96	7.73	14.51

Table 3: Execution time for different OpenMP thread counts with fixed MPI processes.

Execution times per model for Different OpenMP Thread Counts

Speedup for Different OpenMP Thread Counts In this table, we fix the number of MPI processes to 4 and measure speedups as we increase the number of OpenMP threads.

Num threads	100D2	100D2X2	100DX4	100DX8
1	3.89	3.93	3.91	3.82
2	7.68	7.79	7.39	7.67
4	15.00	14.28	14.64	14.63
8	23.15	26.58	26.53	27.18
16	44.93	50.68	45.53	48.67
32	72.21	79.88	82.01	87.34

Table 4: Speedup for different OpenMP thread counts with fixed MPI processes.

Speedup for Different OpenMP Thread Counts with Fixed MPI Processes

Speedup for Different MPI Process Counts with Fixed OpenMP Threads
In this table, we fix the number of OpenMP threads and measure speedups as we increase the number of MPI processes.

MPI processes	100D2	100D2X2	100DX4	100DX8
1	3.91	3.92	3.92	3.91
2	7.73	7.80	7.72	7.67
4	15.00	14.28	14.64	14.63
8	22.97	26.71	27.17	27.12

Table 5: Speedup for different MPI process counts with fixed OpenMP threads.

As demonstrated in the tables, the parallel version exhibits excellent scalability, particularly in terms of both strong and weak scaling.

5 Correctness of the Program

The correctness of the program is verified using the assets provided in the `compare.py` file. This file ensures that the output of the parallel implementations (CUDA, MPI, MPI + OpenMP) matches the sequential version of the K-Means algorithm. Key aspects verified include:

- **Cluster Assignments:** The assignment of data points to clusters is consistent between the sequential and parallel implementations.
- **Execution Integrity:** Multiple runs show consistent results, ensuring the parallelism doesn't introduce errors.