Parallelization of the K-means Clustering Algorithm using MPI, CUDA, OpenMP and MPI+OpenMP

Zanoni Leila (2033176) Martinelli Munera Daniel (2049054)

September 4, 2025

1 Introduction

K-means is an unsupervised learning algorithm that operates by categorizing data points into clusters using a mathematical distance measure, usually Euclidean, from the cluster center. The goal is to minimize the sum of distances between data points and their assigned clusters. Data points that are nearest to a centroid are grouped together within the same category. Unlike supervised learning, where we train models using labeled data, K-means is used when we have unlabeled data to uncover hidden patterns or structures.

1.0.1 Algorithm structure

The k-means algorithm can be described with the following pseudo-code, where X is the set of data points, $C = \{\mu_1,\ \mu_2,\ \ldots,\ \mu_k\}$ is the set of centroids and Y is the set of assignments:

Algorithm 1: K-means clustering (Lloyd's algorithm [1])

Input: Dataset $X = \{x_1, \ldots, x_n\}$, clusters K
Output: Centroids $\{\mu_1, \ldots, \mu_K\}$, labels Y

Initialize: randomly choose K centroids μ_j;
while *not converged* **do**
    `// Assignment step`
    foreach *point* $x_i \in X$ **do**
        assign $y_i \leftarrow \arg\min_j \|x_i - \mu_j\|$;
    `// Update step`
    for $j \leftarrow 1$ **to** K **do**
        $\mu_j \leftarrow$ mean of $\{x_i : y_i = j\}$;

1.0.2 Limitations

Despite its usefulness, this algorithm has a few limitations to be aware of, such as:

- **Dependency on initial guess:** The final results of the clustering can be affected by how well we guess the initial positions of the cluster centers;
- **Difficulty in determining K:** One of the drawbacks of the K-means algorithm is that we have to set the number of clusters (K) in advance. Choosing an incorrect number of clusters can lead to inaccurate results;
- **Time complexity:** The time complexity of the algorithm is

$$O(n \times K \times M \times D),$$

where K is the number of clusters, n is the number of data points, D is the number of dimensions, and M is the number of iterations.

1.1 Sequential code analysis

The sequential K-means implementation `K-MEANS.c` serves as a baseline for our parallel implementations and follows the classical Lloyd's algorithm. It accepts six command-line arguments: the input data file, number of clusters K, maximum number of iterations, minimum percentage of class changes, precision in the centroid distance after the update and the output file.

Data points are read from the input file using `strtok` to parse dimensions separated by tab characters. Points and centroids are stored using linear memory (1D arrays) with index calculations to simulate two-dimensional access. Cluster assignment and centroid updates are computed iteratively based on Euclidean distance.

The implementation uses `memcpy` for copying arrays efficiently and manual memory management (`malloc`/`calloc`). While functionally correct, the sequential approach can be computationally expensive for large datasets.

2 MPI Implementation

The Message Passing Interface (MPI) [2] is an Application Program Interface that defines a model of parallel computing where each parallel process has its own local memory, and data must be explicitly shared by passing messages between processes. By default, the k-means clustering program follows a

Globally Parallel, Locally Sequential (GPLS) model. This means that the critical initialization and termination tasks are handled by rank 0, while the looping part of the algorithm is entirely parallelized.

2.1 Workload distribution

This implementation uses a classic data-parallel K-means: every MPI rank holds the full dataset and a full copy of the centroids.
The work is split by rows: each rank processes a contiguous block of points (`local_lines`) and performs local point assignment, local centroid update (per-cluster counts and sum coordination). All ranks run a single collective to obtain global counts and sums; each rank computes new centroids locally from those global sums.
Then the algorithm repeats until stopping conditions, then rank 0 gathers all per-point class id's with `MPI_Gatherv` and finally writes the output.

2.2 Fully Local Operations

Each rank processes a contiguous set of rows. The code computes `local_lines` and (relative) `start_index` per rank. Then each rank loops over its local points to assign them to the closest centroid and computes partial sums that will be used for centroid update.

```
// ================= POINT ASSIGNMENT LOOP =================
for (i = start_index; i < start_index + local_lines; ++i)
{
    class = 1;
    minDist = FLT_MAX;

    for (j = 0; j < K; ++j) {
        dist = euclideanDistance(&data[i * samples], &
             centroids[j * samples], samples);
        if (dist < minDist) {
            minDist = dist;
            class = j + 1;
        }
    }

    if (classMap[i] != class) { changes++; }
    classMap[i] = class;
}

zeroFloatMatriz(pointsPerClass, K, 1);
zeroFloatMatriz(auxCentroids, K, samples);

// ================= PARTIAL SUMS LOOP =================
for (i = start_index; i < start_index + local_lines; ++i)
{
    class = classMap[i];
    pointsPerClass[class - 1] += 1.0f;
    for (j = 0; j < samples; j++)
    {
        auxCentroids[(class - 1) * samples + j] += data[i *
             samples + j];
    }
}
```

2.3 Ranks Synchronization

Each rank has its own `allgather_buffer` which contains:

- `allgather_buffer[0]` : local changes
- `allgather_buffer[1]` : local points-per-class count
- `allgather_buffer[2]` : local auxCentroids coordinates

```
float_changes = (float)changes;
allgather_buffer[0] = float_changes;
memcpy(&allgather_buffer[1], pointsPerClass, K * sizeof(float
     ));
memcpy(&allgather_buffer[1 + K], auxCentroids, K * samples *
     sizeof(float));
```

Once the local buffer is loaded, each rank calls `MPI_Allreduce`.

`MPI_Allreduce` performs element-wise SUM reduction of a single packed float buffer that contains (1) value for changes, (K) values counts and ($K * samples$) values for coordinate sums.
The result is written in place on all ranks. This collective serves as the synchronization step that merges local partial sums into global sums, making them available to all processes.

```
MPI_Allreduce(MPI_IN_PLACE, allgather_buffer,
     allgather_buffer_size, MPI_FLOAT, MPI_SUM,
     MPI_COMM_WORLD);
```

Finally, the local variables are updated with the global data stored inside the buffer.

```
changes = (int)allgather_buffer[0];
memcpy(pointsPerClass, &allgather_buffer[1], K * sizeof(float
     ));
memcpy(auxCentroids, &allgather_buffer[1 + K], K * samples *
     sizeof(float));
```

2.4 Parallel Centroid Calculation

```
for (i = 0; i < K; i++) {
    for (j = 0; j < samples; j++) {
        auxCentroids[i * samples + j] /= pointsPerClass[i];
    }
}

for (i = 0; i < K; i++) {
    distCentroids[i] = euclideanDistance(&centroids[i *
         samples], &auxCentroids[i * samples], samples);
    if (distCentroids[i] > maxDist) maxDist = distCentroids[i
         ];
}
memcpy(centroids, auxCentroids, (K * samples * sizeof(float))
     );
```

No MPI call is needed here; each rank computes new centroids from global sums and global counts.

2.5 Outside Iterative Loop

After meeting the convergence conditions, we need to gather the results of individual ranks, namely the class mapping stored inside each rank's `classMap`. Depending on the size of the dataset, not all ranks can process the same number of lines, so we cannot use `MPI_Gather(...)` directly. Instead we implemented a 3 steps approach:

- **STEP 1:** `MPI_Gather(...)` collects equal-sized blocks from every rank in a single array on the root rank. Here, the gathered data is the number of lines to process for each rank (1 integer per rank).

```
int *recvcounts = (int *)malloc(size * sizeof(int));
int *displs = (int *)malloc(size * sizeof(int));

MPI_Gather(&local_lines, 1, MPI_INT, recvcounts, 1,
     MPI_INT, 0, MPI_COMM_WORLD);
```

- **STEP 2:** Root rank calculates `recvcounts[i]` and `displs[i]` that will be used in step 3.

```
if (rank == 0)
{
    int total_lines = recvcounts[0];
    displs[0] = 0;
    for (int i = 1; i < size; i++)
    {
        displs[i] = displs[i - 1] + recvcounts[i - 1];
        total_lines += recvcounts[i];
    }
// optionally check: sum(recvcounts) == lines
}
```

- **STEP 3:** `MPI_Gatherv(...)` collects blocks of variable size from every rank into a single array on the root rank; you must provide `recvcounts` and `displs` (displacements).

```
MPI_Gatherv(&classMap[start_index], local_lines, MPI_INT
     , classMap, recvcounts, displs, MPI_INT, 0,
     MPI_COMM_WORLD);
```

2.6 MPI Performance analysis

We tested with 1, 2, 4, 8, 16, 32 and 64 MPI processes, evaluating on multiple datasets of increasing size and dimensionality. The results are reported in terms of execution time, speedup, efficiency and scaling quality.

2.6.1 Speedup analysis

For datasets like *input100D2* and *200k_100*, speedup grows linearly up to 32 processes, achieving **27-32x** speedup, and still improves at 64 processes with values around **48x**. Smaller datasets such as *input100D* achieve lower maximum speedups due to communication overhead, reducing speedup.

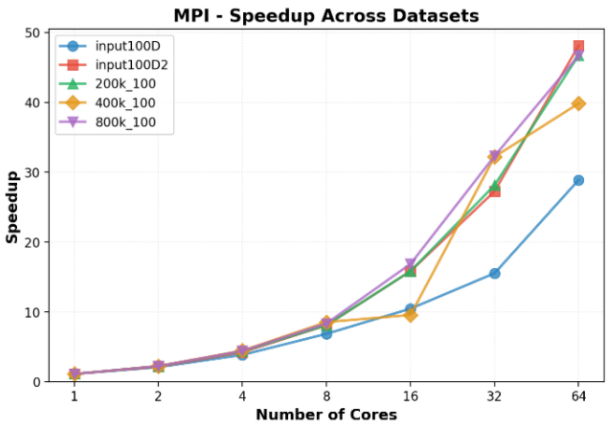

Figure 1: Speedup of MPI K-means across datasets

2.6.2 Execution time across datasets

For small datasets, execution time decreases rapidly at first, but eventually flattens out because of communication overhead. For larger datasets, the execution time continues to fall, showing good scalability. The plot below shows the average execution times across 5 runs:

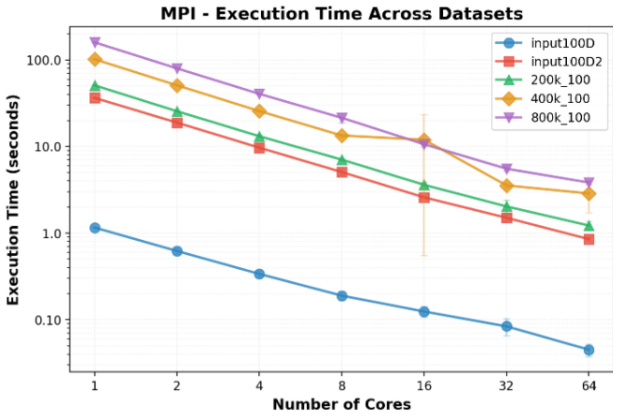

Figure 2: Execution time of MPI across datasets

2.6.3 Parallel Efficiency

Efficiency remains close to **100%** for larger datasets up to 32 processes, indicating near-ideal scaling in that range. At 64 processes efficiency starts to drop, as communication costs outweigh computation. We can also see that parallelization is most effective when the dataset is large enough because for small datasets, the efficiency drops faster, below **50%**.

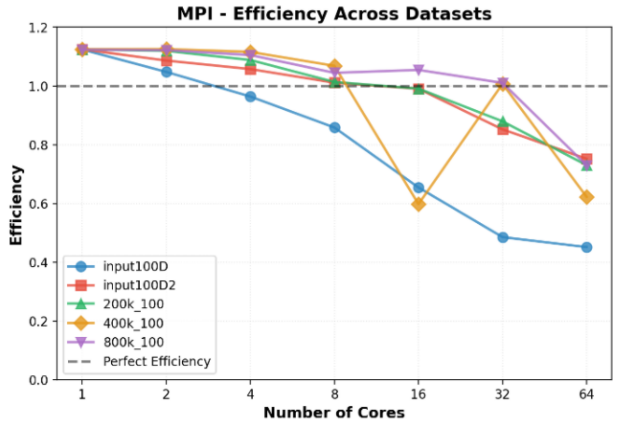

Figure 3: Efficiency across datasets

2.6.4 Scaling Quality

The slopes for medium and large datasets (*input100D2, 200k_100, 800k_100* are between **0.90** and **0.91**, classified as excellent scaling. The smallest dataset achieves only **0.76**, demonstrating that dataset size directly influences scaling behavior.

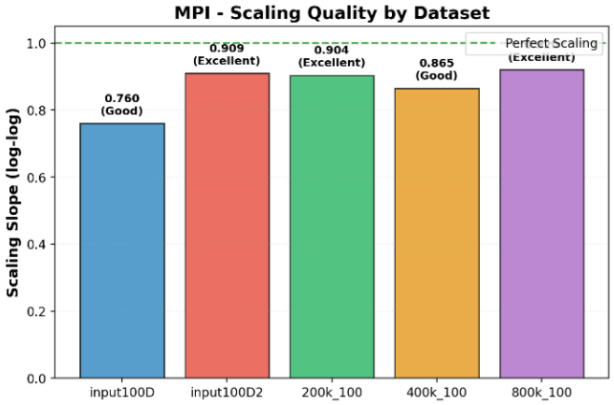

Figure 4: Scaling Quality aceoss datasets

3 OpenMP Implementation

The OpenMP [3] implementation exploits shared memory parallelism through thread-level parallelization. This approach is particularly effective for multicore processors, where threads can efficiently share data through cache hierarchies.

3.1 Euclidean Distance Function

We modified the Euclidean distance function to make it more efficient on data points with higher dimensionality, by employing SIMD vectorization to process multiple arrays with one CPU instruction and then perform a reduction on the variable `dist`. We also used a fused multiply-add function, `fmaf`, from the C math library.

```
float euclideanDistance(float *point, float *center, int
      samples)
{
       float dist=0.0;
       for(int i=0; i<samples; i++)
       {
              dist+= (point[i]-center[i])*(point[i]-center[i
                   ]);
       }
       dist = sqrt(dist);
       return(dist);
}
```

Algorithm 1: Sequential version

```
float euclideanDistance(float *point, float *center, int
      samples)
{
    float dist = 0.0;
    #pragma omp simd reduction(+ : dist)
    for (int i = 0; i < samples; i++) // Loop through all
         dimensions
    {
        dist = fmaf(point[i] - center[i], point[i] - center[i
             ], dist);
    }
    return dist;
}
```

Algorithm 2: OpenMP-parallelized version

3.2 Address mapping for data points

```
float** row_pointers = (float**) malloc(lines * sizeof(float
     *));
#pragma omp parallel for
for (unsigned int row = 0; row < lines; ++row)
{
    row_pointers[row] = &data[row * samples];
}
```

Algorithm 3: Row-pointer mapping for 2D view over 1D storage

We use this address map to improve the performance of the cache and to avoid computing `i * samples` repeatedly in the inner loop.

3.3 STEP 1: Assigning each point to the nearest centroid

```
#pragma omp parallel for private(i, j, class, minDist, dist)
     shared(data, centroids, classMap, lines, samples, K)
     reduction(+ : changes) schedule(dynamic, 128)
for (i = 0; i < lines; i++)
   {
       class = 1;
       minDist = FLT_MAX;

       // Cache-friendly: process all centroids for current
            point
       for (j = 0; j < K; j++)
       {
           dist = euclideanDistance(row_pointers[i], &
                centroids[j * samples], samples);

           if (dist < minDist)
           {
               minDist = dist;
               class = j + 1;
           }
       }

       // Check for changes outside the inner loop
       if (classMap[i] != class)
       {
           changes++;
       }
       classMap[i] = class;
   }
```

Algorithm 4: OpenMP

- `#pragma omp parallel for` creates a parallel region where the for loop is distributed across multiple threads and each thread gets a subset of iterations to execute.
- `private(i, j, class, minDist, dist)`: each thread gets its own copy of these variables, preventing race conditions.
- `shared(data, centroids, classMap, lines, samples, K)`: all threads access the same memory location on which these variables are stored. They are all read-only except `classMap`
- `reduction(+ : changes)`: safely accumulates `changes` across all threads while preventing race conditions on the variable.
- `schedule(dynamic, 128)`: each thread gets 128 consecutive loop iterations at a time, when it finishes a chunk and requests another one.

3.4 STEP 2: Recalculating centroids

3.4.1 Phase 1

```
    #pragma omp parallel
{
    // Each thread gets its own local arrays
    int *local_pointsPerClass = (int *)calloc(K, sizeof(int));
    float *local_auxCentroids = (float *)calloc(K * samples,
         sizeof(float));

    #pragma omp for private(i, j, class) schedule(dynamic, 64)
    for (i = 0; i < lines; i++)
    {
        class = classMap[i];
        local_pointsPerClass[class - 1]++; // Count points per
              cluster (local)

        for (j = 0; j < samples; j++)
        {
            local_auxCentroids[(class - 1) * samples + j] +=
                 data[i * samples + j]; // Sum coordinates (
                 local)
        }
    }
```

Here, each thread gets private arrays: `local_pointsPerClass` and `local_auxCentroids`. `local_pointsPerClass` keeps track of how many points are assigned to each cluster. Note that if it was a shared variable, many threads would try to increment it simultaneously. `local_auxCentroids` contains the partial sums of the coordinates of the new centroids. Again, if this variable was shared, multiple threads could try to write this variable at the same time, since multiple points could belong to the same cluster.

3.4.2 Phase 2

```
#pragma omp critical
    {
        // Combine results efficiently - unroll when possible
        for (i = 0; i < K; i++)
        {
            pointsPerClass[i] += local_pointsPerClass[i];
        }
        for (i = 0; i < K; i++)
        {
            float *dest = &auxCentroids[i * samples];
            float *src = &local_auxCentroids[i * samples];
            for (j = 0; j < samples; j++)
            {
                dest[j] += src[j];
            }
        }
    }

    // Free thread-local memory
    free(local_pointsPerClass);
    free(local_auxCentroids);
    } // <-- exit local section
```

This reduction step must be executed sequentially to avoid race conditions on the global variable: one thread at a time adds its `local_pointsPerClass` to `pointsPerClass`, and then that same thread adds up its `local_auxCentroids` to `auxCentroids`.

3.4.3 Phase 3

```
#pragma omp parallel for private(i, j) collapse(2)
    for (i = 0; i < K; i++)
    {
        for (j = 0; j < samples; j++)
        {
            auxCentroids[i * samples + j] /= pointsPerClass[i];
        }
    }
}
```

In this part, `#pragma omp parallel for private(i, j) collapse(2)` prevents race conditions on loop counter and flattens 2 nested for loops into a single iteration space, enabling parallelization of both outer and inner loops and creating a larger pool of work to distribute among threads.

3.5 OpenMP Performance analysis

3.5.1 Speedup across datasets

Medium and large datasets achieve nearly **linear speedup** up to 64 threads, with speedups above **40x** for **400k_100** and **800k_100**. Smaller datasets achieve lower speedups, for example for **input100D**, we get a speedup around **23x** on 64 threads.
Compared to MPI, OpenMP speedup curves are smoother, because threads share memory directly and avoid explicit communication overhead.

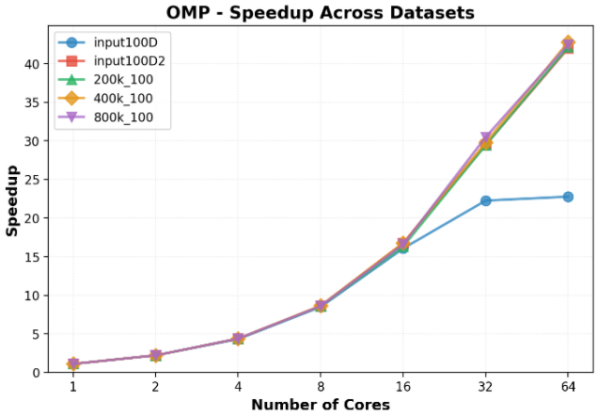

Figure 5: Speedup across datasets

3.5.2 Execution time across datasets

We see on the figure that execution time consistently decreases with more threads for all datasets. Small datasets such as **input100** finish extremely fast, but the runtime reduction flattens because of overhead from thread creation and synchronization. Meanwhile larger datasets show much stronger reductions in execution time.

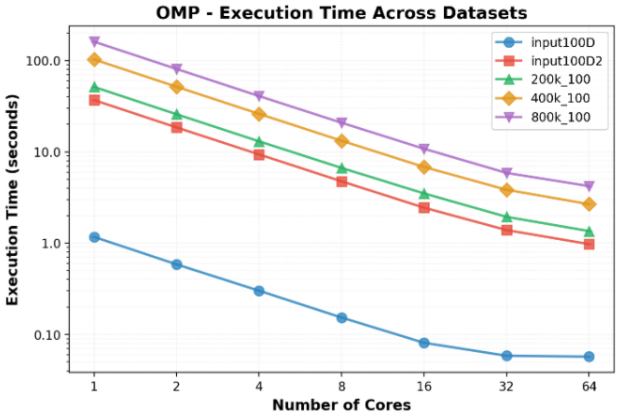

Figure 6: Execution time across datasets

3.5.3 Efficiency across datasets

Larger datasets maintain efficiency close to **100%** up to 16 threads, and still achieve 70-80% at 64 threads. Smaller datasets suffer from rapid efficiency loss, again because of overhead when work per thread is too small. OpenMP efficiency is higher than MPI at low to medium thread counts.

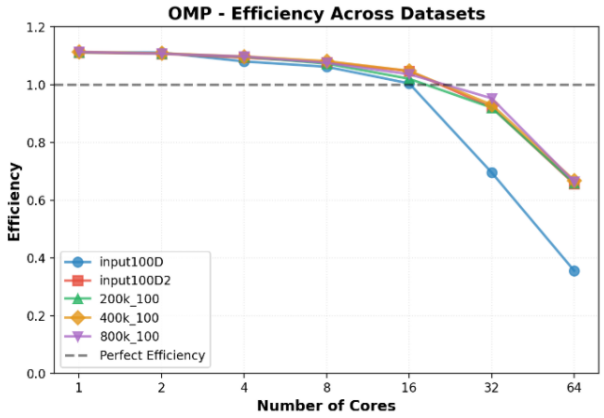

Figure 7: Efficiency across datasets.

4 OpenMP + MPI Hybrid Implementation

4.1 Parallelization Strategy

This program uses a hybrid MPI + OpenMP strategy. MPI splits the dataset across processes and handles inter-process communication of metadata and global aggregates. Inside each MPI rank, OpenMP parallelizes compute-heavy loops (distance computations, local accumulation, and centroid averaging).

We first reproduced a plain MPI implementation and then enhanced it by parallelizing heavy loops inside each rank with OpenMP via the same `#pragma`s used in the pure OpenMP implementation. We also experimented with improving memory efficiency, at the cost of additional communication overhead during the allocation phase.

4.2 Allocation Phase

Only rank 0 reads and loads the dataset. It then broadcasts the dataset's metadata via `MPI_Bcast()` and distributes the actual dataset across ranks via `MPI_Scatterv()`.

Note: `MPI_Scatterv()` is used instead of `MPI_Scatter()` to handle dataset sizes not divisible by the number of processes.

After that, rank 0 initializes centroids and calls `MPI_Bcast()` to broadcast the centroids' initial coordinates to all ranks.

In this way, every rank owns a contiguous subset of data points (except for rank 0, which has the entire dataset) and all initial centroid coordinates.

4.3 Algorithm

Each rank goes through a fully local section in which points are assigned to the closest centroid and partial sums are accumulated for centroid updates. These two phases consist of separate loops, whose iterations are distributed across multiple threads via OpenMP pragmas. In the second loop, a critical section is needed for incrementing `local_pointsPerClass`and `local_auxCentroids` arrays that keep track of how many points are assigned to each cluster and new cluster coordinates.

Afterward, all ranks synchronize via an `MPI_Allreduce()` call. Just like in the pure MPI implementation, an `allgather_buffer` is used to perform a single MPI call to aggregate and broadcast: (1) a value for changes, (K) count values, and (K × samples) values for coordinate sums.

Finally, each rank computes the centroid updates using the aggregated sums and counts, and performs convergence checks.

Note: `classMap` still contains only local point assignments at this stage.

Once convergence is reached, `MPI_Gather()` and `MPI_Gatherv()` are called to collect local `classMaps` into a complete `classMap` in the root rank. This is necessary to write out the results.

4.4 MPI + OpenMP Performance analysis

4.4.1 Efficiency across datasets

This plot compares efficiency between configurations with **2 MPI processes per node** and **4 MPI processes per node**, using OpenMP threads.

- With **2 MPI processes**, efficiency remains high up to 32 cores and gradually decreases at 46 cores.
- With **4 MPI processes**, efficiency is lower, showing that increasing MPI processes brings more communication overhead and reduces resource utilization.

We can clearly see that fewer MPI processes with more OpenMP threads per process generally provides better efficiency.

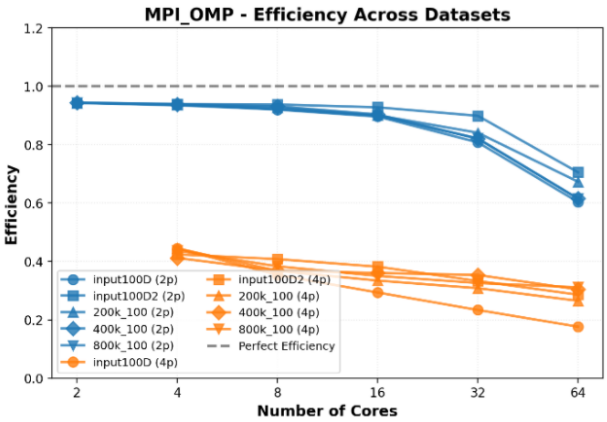

Figure 8: Efficiency across datasets

4.4.2 Execution time across datasets

This figure shows us that:

- With **2 MPI processes**, runtime decreases smoothly with core counts across all datasets. Larger datasets scale better.
- With **4 MPI processes**, runtimes are consistently higher, especially for larger datasets. The communication between more MPI processes leads to computational overhead.

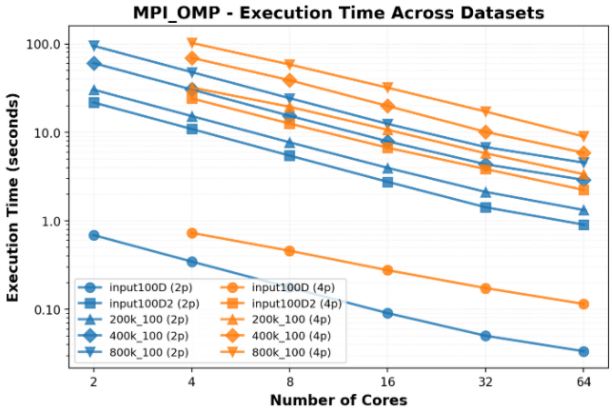

Figure 9: Execution time across datasets

4.4.3 Speedup across datasets

We can clearly see that using **2 MPI processes** achieves strong speedups, scaling linearly up to 64 cores. Meanwhile with **4 MPI processes** achieve lower speedups, rarely exceeding 20x.

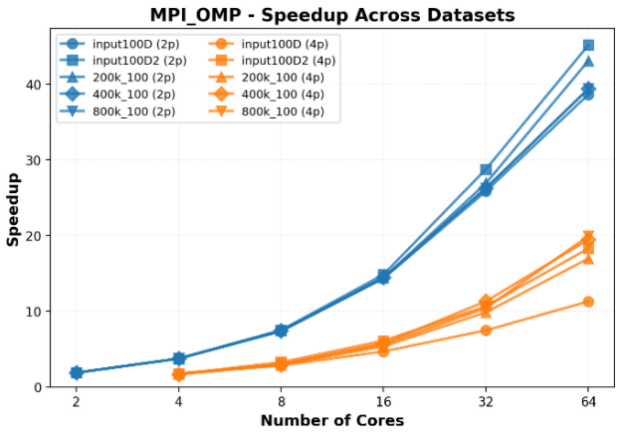

Figure 10: Speedup across datasets

5 CUDA Implementation

5.1 Constant Memory for Frequently Accessed Parameters

Constant memory accesses are **cached** and have **minimal latency** when accessed uniformly. **Read-Only semantics** ensure data consistency across all threads and prevent accidental modifications. These variables are then assigned by the host device.

```
    // Declare constant memory for algorithm parameters -
        cached and read-only
__constant__ int c_numPoints; // Total number of data points
__constant__ int c_dimensions; // Dimensionality of each data
      point
__constant__ int c_K; // Number of clusters
```

5.2 CUDA Streams

CUDA [4] streams enable **asynchronous execution** and **memory transfer overlap** for maximum GPU utilization. Below is how we employed them:

```
    // Stream setup
    cudaStream_t stream;
    CHECK_CUDA_CALL(cudaStreamCreate(&stream));

    // Start timing within the stream
    CHECK_CUDA_CALL(cudaEventRecord(computation_start, stream)
        );
```

```
    // All GPU memory allocated asynchronously for better
        performance
    CHECK_CUDA_CALL(cudaMallocAsync(&d_data, lines * samples *
         sizeof(float), stream));
    CHECK_CUDA_CALL(cudaMallocAsync(&d_centroids, K * samples
        * sizeof(float), stream));
    CHECK_CUDA_CALL(cudaMallocAsync(&d_newCentroids, K *
        samples * sizeof(float), stream));
    CHECK_CUDA_CALL(cudaMallocAsync(&d_classMap, lines *
        sizeof(int), stream));
    // ... more allocations
```

5.3 KERNEL 1 - Point Assignment Kernel

```
    __global__ void assignPointsToCentroids(
    float *points, // Input dataset (all data points) in
        global memory
    float *centroids, // Current cluster centers in global
        memory
```

```
    int *assignments, // Output array storing cluster assignment for each point
    int *changes, // Global counter for points that changed clusters
    bool useSharedMemory) // Flag to enable/disable shared memory optimization
{
    // Thread-to-point mapping (each thread processes exactly 1 data point)
    int pointIdx = blockIdx.x * blockDim.x + threadIdx.x;

    // Safety check: Some threads might not have corresponding data points
    if (pointIdx >= c_numPoints)
        return;

    // ========== PHASE 1 - SHARED MEMORY LOADING ==========

    // Dynamic allocation of shared memory at Kernel Launch
    extern __shared__ float sharedCentroids[];

    // Load centroids into shared memory WITH COLLABORATIVE LOADING (if enabled)
    if (useSharedMemory)
    {
        for (int i = threadIdx.x; i < c_K * c_dimensions; i += blockDim.x)
        {
            if (i < c_K * c_dimensions) // Bounds check
                sharedCentroids[i] = centroids[i];
        }

        // Required synchronization!!
        __syncthreads();
    }

    // ========== PHASE 2 - CORE ALGORITHM ==========

    // Initialization of per-thread variables
    float minDistance = FLT_MAX;
    int bestCentroid = 0;
    int oldAssignment = assignments[pointIdx];

    // Calculate distance to each centroid
    for (int k = 0; k < c_K; k++)
    {
        float distance = 0.0f;

        // Calculate squared Euclidean distance (sqrt not needed for comparison)
        for (int d = 0; d < c_dimensions; d++)
        {
            float centroid_val;
            // Use shared memory if available
            if (useSharedMemory)
                centroid_val = sharedCentroids[k * c_dimensions + d];
            else
                centroid_val = centroids[k * c_dimensions + d];

            float diff = points[pointIdx * c_dimensions + d] - centroid_val;
            distance += fmaf(diff, diff, distance);
        }

        // Update closer centroid
        if (distance < minDistance)
        {
            minDistance = distance;
            bestCentroid = k;
        }
    }
    // Convert to 1-based indexing
    assignments[pointIdx] = bestCentroid + 1;

    // Count changes using atomic operation
    if (oldAssignment != bestCentroid + 1)
    {
        atomicAdd(changes, 1);
    }
}
```

All threads cooperate to load centroids into shared memory, and the total global memory reads K x dimensions per block, resulting in a massive reduction in memory traffic.
Without shared memory, each thread would read centroids K times from global memory, while reading lines x K x dimensions.

5.3.1 Point Assignment Kernel (Launch)

```
// Copy current centroids to GPU
CHECK_CUDA_CALL(cudaMemcpyAsync(d_centroids, centroids, K * samples * sizeof(float), cudaMemcpyHostToDevice, stream));
CHECK_CUDA_CALL(cudaMemcpyAsync(d_changes, &zero, sizeof(int), cudaMemcpyHostToDevice, stream));

// Kernel call
assignPointsToCentroids<<<gridSize, blockSize, sharedMemSize, stream>>>(d_data, d_centroids, d_classMap, d_changes, useSharedMemory);

// Copy results back to host
CHECK_CUDA_CALL(cudaMemcpyAsync(classMap, d_classMap, lines * sizeof(int), cudaMemcpyDeviceToHost, stream));
CHECK_CUDA_CALL(cudaMemcpyAsync(&changes, d_changes, sizeof(int), cudaMemcpyDeviceToHost, stream));
```

```
dim3 blockSize(32);
dim3 gridSize((lines + blockSize.x - 1) / blockSize.x);
```

Algorithm 5: Grid and Block configuration

We experimented with different block sizes, ranging from 32 to 512. In the end, the 32 threads block size was the one bringing the best speed-ups with respect to the sequential code. By analyzing the kernel, we were able to associate this behavior with the fact that each block has to go through a critical section. Knowing that the GPU at our disposal, NVIDIA's RTX Quadro 6000 , has 72 SMs each with a maximum capacity of 1024 threads, choosing a smaller block size makes it possible to run multiple blocks concurrently on the same SM.
Smaller block size also means having a larger number of blocks , especially when dealing with very large datasets. Saturating all SMs becomes easy and since each SM can run at most 1024 threads simultaneously, the blocks that do not fit in any SMs have to be rescheduled. Nonetheless, this scheduling overhead resulted to be less problematic than having a smaller number blocks executing the critical section concurrently.

5.4 KERNEL 2 - Centroid update

```
__global__ void recalculateCentroids(
        float *points,
        int *assignments,
        float *newCentroids,
        int *pointsPerCluster)
{
    int clusterIdx = blockIdx.x;
    int dimIdx = threadIdx.x;

    if (clusterIdx >= c_K || dimIdx >= c_dimensions)
        return;

    float sum = 0.0f;
    int count = 0;

    // ============ Phase 1 - Data Accumulation ============
    // Each thread handles one dimension of one cluster
    for (int i = 0; i < c_numPoints; i++)
    {
        // Convert from 1-based
        if (assignments[i] == clusterIdx + 1)
        {
            sum += points[i * c_dimensions + dimIdx];
            if (dimIdx == 0) // Only count once per point
                count++;
```

```
        }
    }
    // ============ Phase 2 - Point Count Storage ============
    // Store the count (only for dimension 0 to avoid race conditions)
    if (dimIdx == 0)
        pointsPerCluster[clusterIdx] = count;

    __syncthreads();

    // ============ Phase 3 - Centroid Calculation ============
    // Calculate mean for this dimension
    int totalCount = pointsPerCluster[clusterIdx];
    if (totalCount > 0)
        newCentroids[clusterIdx * c_dimensions + dimIdx] = sum / totalCount;
    else
        newCentroids[clusterIdx * c_dimensions + dimIdx] = 0.0f; // Or keep old centroid
}
```

Algorithm 6: Centroid Update in CUDA

Each thread is responsible for computing one dimension of one cluster's centroid. This creates a 2D parallelization:

- **Horizontal parallelism:** Different blocks handle different clusters
- **Vertical parallelism:** Different threads within a block handle different dimensions.

5.4.1 Phase 1 - Data Accumulation

Each thread accumulates values for its assigned cluster-dimension pair.
For example:

```
Thread(0,0): Accumulates dimension 0 values for cluster 0, counts points
Thread(0,1): Accumulates dimension 1 values for cluster 0
Thread(1,0): Accumulates dimension 0 values for cluster 1, counts points
Thread(k,d): Accumulates dimension d values for cluster k
```

Only thread 0 (dimension 0) counts the number of points to avoid duplicate counting.

5.4.2 Phase 2 - Point Count Storage

Since only threads with `dimIdx == 0` are responsible for incrementing the `count` variable, we need to broadcast this information to all other threads in the block. This is done by updating `pointsPerCluster[clusterIdx]` which is visible to all the threads in the block.
Only thread with `dimIdx == 0` writes the count to avoid race conditions.

5.4.3 Phase 3 - Centroid Calculation

Compute final centroid coordinates by dividing accumulated sums by point count.
Each thread reads shared `pointsPerCluster[clusterIdx]`, then computes `newCentroid[d] = sum[d] / totalCount` and finally writes to a unique location in `newCentroids`.
Edge case handling: Empty clusters get zero coordinates.

5.4.4 Centroid Update (Launch)

- **Grid dimensions:** K blocks (one block per cluster)
- **Block dimensions:** sample threads (one thread per dimensions)
- **Shared memory:** 0 bytes (it would not make sense to load the entire dataset on every block shared memory)

```
dim3 centroidGrid(K); // K blocks (one block per cluster)
dim3 centroidBlock(samples); // `samples` threads (one thread per dimension)

// Kernel call
recalculateCentroids<<<centroidGrid, centroidBlock, 0, stream>>>(d_data, d_classMap, d_newCentroids,
    d_pointsPerCluster);

// Copy new centroids back to host for convergence check
    CHECK_CUDA_CALL(cudaMemcpyAsync(auxCentroids,
        d_newCentroids, K * samples * sizeof(float),
        cudaMemcpyDeviceToHost, stream));
    CHECK_CUDA_CALL(cudaMemcpyAsync(pointsPerClass,
        d_pointsPerCluster, K * sizeof(int),
        cudaMemcpyDeviceToHost, stream));

    CHECK_CUDA_CALL(cudaStreamSynchronize(stream)); // Wait for GPU operations to complete
```

5.4.5 Alternative approach

We also thought about assigning a data point to each thread (instead of dimension). In this way each thread would process only one sample and add it to the cumulative sum of its own cluster.
The thread-per-sample approach would have much better memory coalescing and cache efficiency. However, the current approach was chosen because:

1 Simplicity: No race conditions, no complex reductions, no critical sections
2 Predictable performance: Equal work distribution
3 Typical use cases: Works well for moderate dimensions (< 1000)
4 Implementation complexity: Much easier to debug and maintain.

5.5 KERNEL 3 - Custom Reduction [5]

5.5.1 Warp Level Reduction

This is an **ultra-high-performance reduction kernel** that leverages the lowest level of CUDA parallelism - warp-level primitives - and employs a butterfly pattern. Here we describe its general functioning, while in our code we applied little changes to make it suit our purpose: computing max distance convergence check.

```
__device__ __forceinline__ float warp_reduce(float val)
{
    // this bitmask indicates all 32 thread in the warp partecipate to the reduction. Each bit represents a thread
    const unsigned int FULL_MASK = 0xffffffff;
#pragma unroll
```

```
  for (unsigned int i = 16; i > 0; i /= 2)
  {
    val = max(val, __shfl_down_sync(FULL_MASK, val, i));
  }
  return val;
}
```

Algorithm 7: General structure of the function

`__shfl_down_sync()` allows threads within a warp to exchange data directly though registers, bypassing shared memory entirely. As a matter of fact **Register-level communication** is exponentially faster than shared memory access because it operates at the lowest hardware level. It requires perfectly balanced binary tree structure (we need padding to enable the algorithm)

5.5.2 Block Level Reduction

2 important variables are used for indexing:

```
// Position within warp (0-31)
unsigned int lane = threadIdx.x % warpSize;
// Warp ID within block
unsigned int wid = threadIdx.x / warpSize;
```

```
// ==== Phase 1: Warp-level reduction ====
sum = warp_reduce(sum);
if (lane == 0) shared[wid] = sum;

__syncthreads();

// ==== Phase 2: Inter-warp reduction ====
sum = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] :
      0;
if (wid == 0) sum = warp_reduce(sum);
```

- **Phase 1:** Each warp reduces its values to a single result, but only thread 0 of each warp (`lane == 0`) writes to shared memory.
- **Phase 2:** Thread 0 from each warp becomes a representative. All together form a new "virtual warp" for the final reduction.

```
unsigned int lane = threadIdx.x % warpSize; // Position
      within warp (0-31)
unsigned int wid = threadIdx.x / warpSize; // Warp ID within
      block
```

Algorithm 8: Main variables

5.6 CUDA Performance Analysis

5.6.1 Speedup across datasets

input100D achieves around **40x speedup**, while larger datasets achieve between **90x and 105x speedup** relative to their sequential baselines.

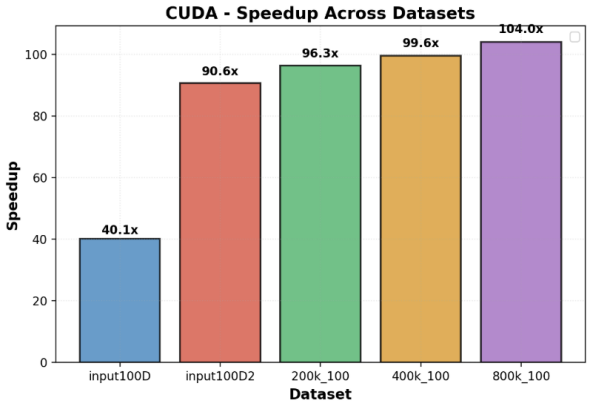

Figure 11: Speedup across datasets

5.6.2 Execution time across datasets

Execution time grows with dataset size, from **0.03s** for *input100D* up to **1.72s** for *800k_100*.
Even for the largest dataset, execution time remains very low compared to MPI and openMP, due to the computational power of GPUs.

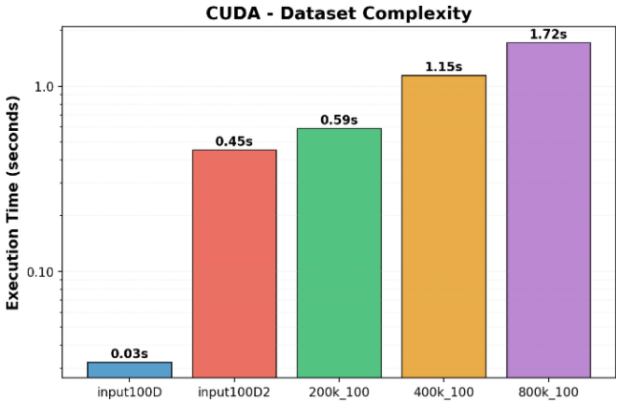

Figure 12: Execution time across datasets

6 Overall Performance

For each configuration, the program was executed five times in order to account for runtime variability. From these runs, we recorded the average, minimum, and maximum execution times, together with the standard deviation. This allowed us to evaluate both central tendency and dispersion of the results.

All experiments were performed using the standard K-means parameters reported in Table 1.

Parameter	Value
Number of clusters (K)	20
Maximum iterations	3000
Threshold	1.0
Tolerance	0.0001
Random seed	42

Table 1: Standard K-means parameters used in all experiments.

6.1 Comparison between smallest and largest datasets

6.1.1 Execution time

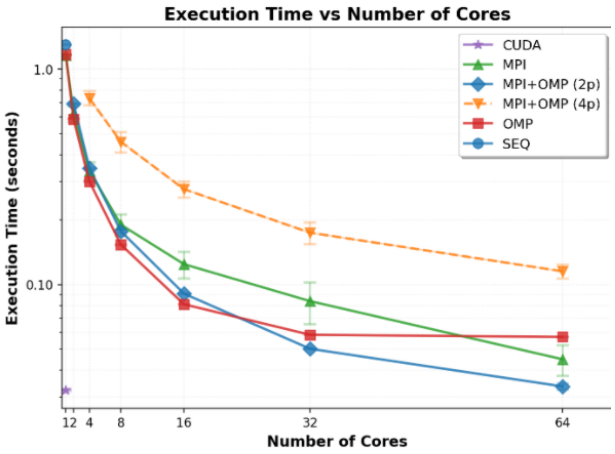

Figure 13: Execution time in *input100D*

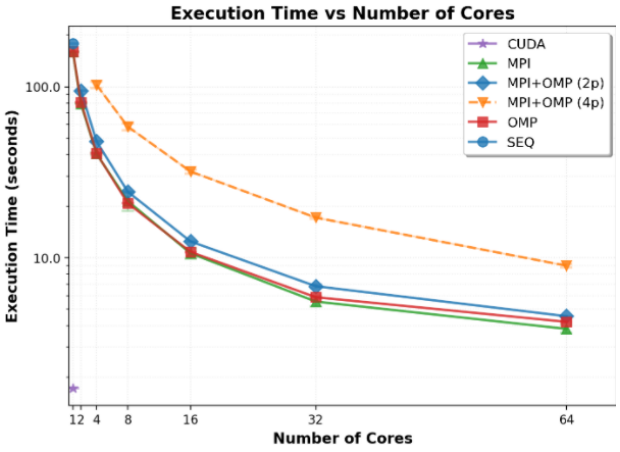

Figure 14: Execution time in *800k_ 100*

For smaller datasets, parallel overhead limits scaling; Meanwhile large datasets achieve very good results in terms of execution time.

6.1.2 Speedup

Larger datasets achieve better speedup , very close to ideal linear scaling because the computational workload per core is significant enough to overshadow the relative cost of communication and synchronization. In contrast, small datasets like the one represented in the plot below saturate early. At this point, communication, synchronization, and thread management overheads become the dominant factors in execution time.

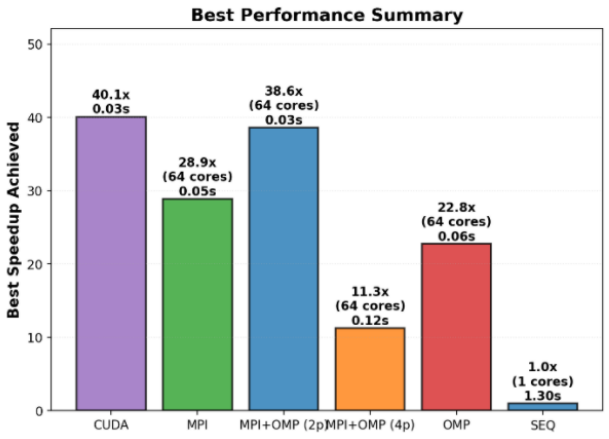

Figure 15: Speedup in *input100D*

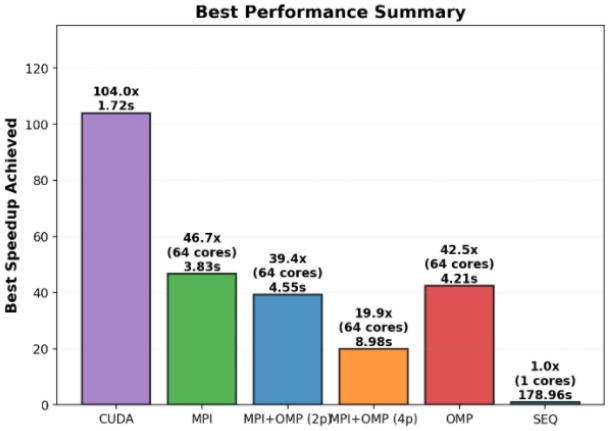

Figure 16: Speedup in *800k_ 100*

6.1.3 Efficiency

For small datasets, efficiency declines as the number of cores increases, where communication overhead dominates. For *800k_ 100*, efficiency remains high up to 64 cores, since computation outweighs communication overhead. CUDA maintains excellent efficiency across all sizes.

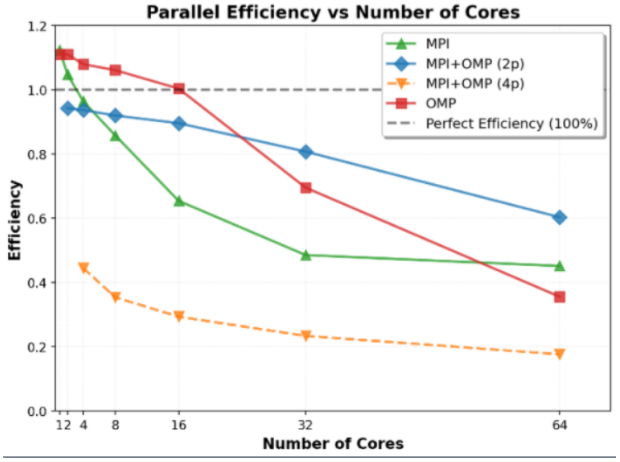

Figure 17: Efficiency in *input100D*

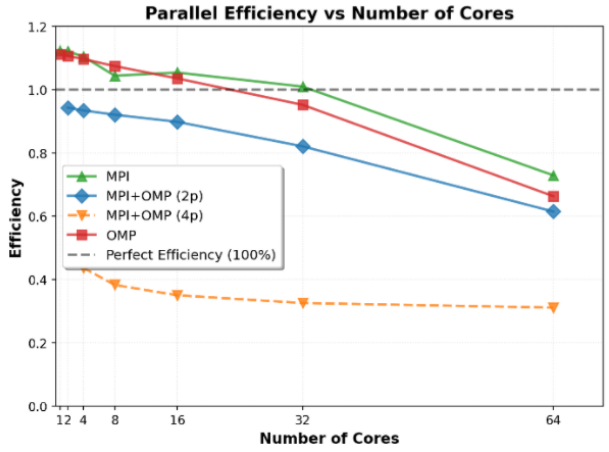

Figure 18: Efficiency in *800k_100*

6.2 Overall comparison across datasets

6.2.1 Best speedup by Implementation

Each bar shows the maximum speedup achieved for each dataset by CUDA, MPI, MPI+OpenMP (2 processes), MPI+OpenMP (4 processes), and OpenMP.

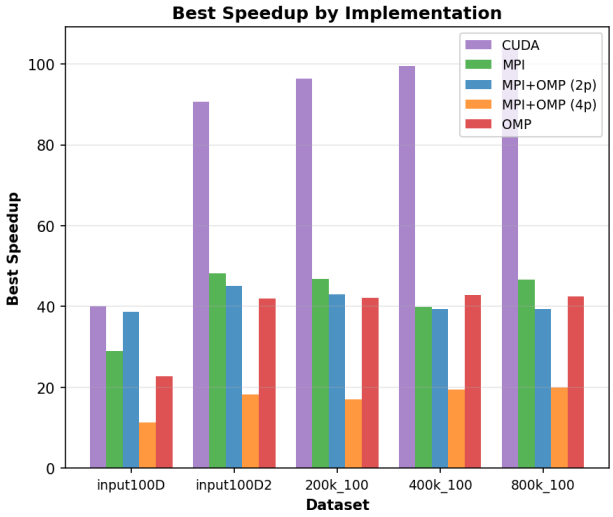

Figure 19: Speedup across all datasets by implementation

- CUDA consistently achieves the highest speedups, regardless of size.
- MPI and OpenMP perform very well, especially on medium and large datasets
- MPI + OpenMP with 2 processes competes with MPI and OpenMP, meanwhile the one with 4 processes is much slower due to MPI overhead.

6.2.2 Dataset complexity Comparison

The increase is proportional to dataset size, showing how computational complexity grows with more points and dimensions.

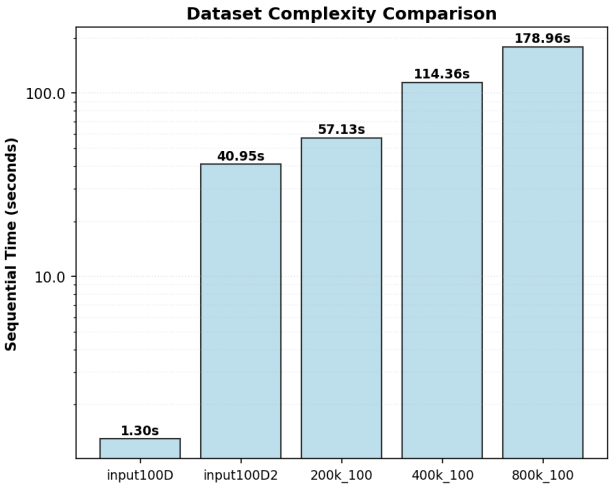

Figure 20: Complexity across all datasets

7 Conclusions

K-means parallelization with CUDA, MPI, OpenMP, and Hybrid MPI+OpenMP demonstrates how dataset size has a significant impact on performance. Parallel overhead restricts speed and efficiency for small datasets, providing only slight gains. However, parallelization is crucial for huge datasets: CUDA achieves sub-second performance with speedups exceeding 100×, while MPI and OpenMP cut runtimes from minutes to seconds. When it comes to CPU approaches, MPI and OpenMP perform equally. If properly adjusted, Hybrid (2p) is competitive, however Hybrid (4p) has communication overhead. For large-scale clustering, MPI and OpenMP are still efficient CPU-based alternatives, even though CUDA offers the greatest acceleration overall.

References

[1] S. P. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[2] Message Passing Interface Forum, "MPI: A message-passing interface standard." `https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/mpi-report.html`, 1994.

[3] OpenMP Architecture Review Board, "OpenMP application programming interface version 5.0." `https://www.openmp.org/specifications/`, 2018.

[4] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide." `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2025.

[5] A. Vardanian, "How to optimize parallel reductions in cuda," 2021. Accessed: 2025-09-04.