Implementation of CUDA K-Means Clustering on Multi-dimensional dataset

PHAN Manh Tung
Course: GPU Computing
Professor: Christophe Picard

February 11, 2025

Contents

1 Overview

K-Means clustering is a fundamental unsupervised machine learning algorithm used for dividing a dataset into k distinct clusters based on feature similarity. It is widely applied in pattern recognition, image segmentation, and data compression. Despite its simplicity, the K-Means algorithm becomes computationally expensive for large-scale, high-dimensional datasets due to the repeated distance computations and centroid updates. Traditional CPU implementations suffer from long execution times, especially when handling millions of data points with high-dimensional feature spaces. To address this, a CUDA-based parallel implementation is used to accelerate K-Means clustering by leveraging the massive parallelism of GPUs. By exploiting these CUDA optimizations, the K-Means algorithm achieves significant speedup compared to sequential CPU-based implementations, making it suitable for large-scale clustering tasks.

2 Code Implementation

The CUDA implementation *(final.cu)* is optimized for parallel execution on the GPU, leveraging thread hierarchy and shared memory for efficiency. The algorithm follows 4 steps:

- **Initialization**: Randomly select k initial centroids from the dataset and copy them to GPU memory.
- **Cluster Assignment** (Parallelized on GPU): Each data point is assigned to its nearest centroid based on Euclidean distance, where each thread computes the distance between a point and all centroids, storing the nearest cluster.
- **Centroid Update** (Parallelized on GPU): New centroids are computed as the mean of all points assigned to each cluster, with threads summing up assigned points using atomic operations and computing the new centroid once all points are processed.
- **Convergence Check** (CPU-GPU Coordination): Centroids are copied back to the CPU to check for convergence, and if not converged, steps 2 and 3 repeat until centroids stabilize or the maximum number of iterations is reached.

This parallelized approach enables significant speedup over CPU implementations, making use of:

- **Thread-level parallelism:** Each thread processes a data point or a centroid update in parallel.
- **Shared memory:** Used to store centroids and reduce redundant global memory accesses.
- **Atomic operations:** Ensures safe updates when multiple threads modify the same centroid.

2.1 Global Parameters

The implementation starts with defining key parameters:

- `MAX_ITER`: Maximum number of iterations before forcing termination.
- `THRESHOLD`: Convergence condition based on the maximum centroid shift. The threshold is set to 0.1 since the testing data is in range (0,1000)

```
int MAX_ITER = 100000;
float THRESHOLD = 0.1f;
```

Listing 1: Global Parameters

2.2 Centroid Initialization

The first step is selecting initial centroids by randomly selecting k initial centroids from the dataset.

```
void initializeCentroids(const float* data, float* centroids,
                          int numPoints, int dimensions, int clusters) {
    srand(42);
    for (int c = 0; c < clusters; ++c) {
        int idx = rand() % numPoints;
        for (int d = 0; d < dimensions; ++d) {
            centroids[c * dimensions + d] = data[idx * dimensions + d];
        }
    }
}
```

Listing 2: initialization

2.3 Distance Computation

To measure similarity between data points and centroids, the *calculateDistance* function computes the squared Euclidean distance in parallel.

- It runs on the device (`__device__`) to minimize memory transfer between CPU and GPU.
- Each thread computes the distance for a single data point, fully utilizing parallelism.

```
__device__ float calculateDistance(const float* point,
                                   const float* centroid,
                                   int dimensions) {
    float dist = 0;
    for (int i = 0; i < dimensions; ++i) {
        dist += (point[i] - centroid[i]) * (point[i] - centroid[i]);
    }
    return sqrtf(dist);
}
```

Listing 3: Distance Calculation Function

2.4 Cluster Assignment Kernel

This kernel assigns each point to its nearest cluster. Since every point's computation is independent, the problem is perfectly suited for parallel execution.

- **Shared memory:** Centroids are loaded into shared memory to reduce slow global memory access.
- **Thread-parallelism:** Each thread handles a single data point.
- **Synchronization:** `__syncthreads()` ensures centroids are fully loaded before computation begins.

```
__global__ void assignCluster(const float* data,
                              float* centroids,
                              int* clusterAssignments,
                              int numPoints,
                              int dimensions,
                              int clusters) {
    extern __shared__ float sharedCentroids[];
    int tid = threadIdx.x;

    for (int d = tid; d < clusters * dimensions; d += blockDim.x) {
        sharedCentroids[d] = centroids[d];
    }
    __syncthreads();

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < numPoints) {
        float minDist = FLT_MAX;
        int bestCluster = -1;

        for (int c = 0; c < clusters; ++c) {
            float dist = calculateDistance(&data[idx * dimensions],
                                           &sharedCentroids[c *
   dimensions],
                                           dimensions);
            if (dist < minDist) {
                minDist = dist;
                bestCluster = c;
            }
        }
        clusterAssignments[idx] = bestCluster;
    }
}
```

Listing 4: Cluster Assignment Kernel

2.5 Centroid Update Kernel

After assigning points to clusters, the new centroid position is computed as the mean of all assigned points.

- **Parallel reduction:** Each thread contributes to computing centroid sums.
- **Atomic operations:** `atomicAdd()` ensures correct accumulation of point coordinates across multiple threads.
- **Shared memory:** Reduces redundant accesses to global memory.
- **Thread synchronization:** `__syncthreads()` ensures threads do not proceed before updates are complete.

```
__global__ void updateCentroids(const float* data,
                                const int* clusterAssignments,
                                float* centroids,
                                int* clusterCounts,
                                int numPoints,
                                int dimensions,
                                        int clusters) {
    extern __shared__ float sharedCentroids[];
    int tid = threadIdx.x;
    int clusterIdx = blockIdx.x;

    if (tid < dimensions) {
        sharedCentroids[tid] = 0.0f;
    }
    __syncthreads();

    for (int i = tid; i < numPoints; i += blockDim.x) {
        if (clusterAssignments[i] == clusterIdx) {
            for (int d = 0; d < dimensions; d++) {
                atomicAdd(&sharedCentroids[d], data[i * dimensions + d
]);
            }
            atomicAdd(&clusterCounts[clusterIdx], 1);
        }
    }
    __syncthreads();

    if (tid < dimensions && clusterCounts[clusterIdx] > 0) {
        centroids[clusterIdx * dimensions + tid] = sharedCentroids[tid]
    / clusterCounts[clusterIdx];
    }
}
```

Listing 5: Centroid Update Kernel

The combination of atomic operations and shared memory makes this kernel highly efficient while preventing race conditions when multiple threads modify the same centroid.

3 Hardware and Execution Environment

The CUDA K-Means implementation was tested on the following platforms:

- **GPU:** NVIDIA Tesla P100 (Kaggle)
- **Platforms:** Kaggle Notebook, Google Colab

The execution was performed using Kaggle's GPU environment with the following command:

```
!./cuda 64 /kaggle/input/my-data/dataset_20000_80_15.txt
```

Example Output:

```
Converged at iteration 52.
Total execution time: 545.391 ms
```

This program achieves convergence in 52 iterations with a total execution time of approximately 545.391 milliseconds on a high-dimensional dataset.

4 Results and Performance Analysis

The CUDA implementation significantly accelerates K-Means clustering due to parallelization. Two tests are conducted as follows:

4.1 Efficiency Test

To evaluate the performance of our CUDA-based K-Means implementation, it is compared against a traditional C-based implementation (kmeans.c) and the widely used `sklearn` library (sklearn.py). The goal is to analyze how well CUDA leverages parallelism for different dataset sizes and dimensions.

4.1.1 Test Setup

- **Thread Configuration:** The CUDA implementation was executed with a fixed thread-per-block setting of 64.
- **Comparison Metrics:** The number of iterations required for convergence and the execution time (in milliseconds).
- **Averaging Runs:** Each test was run **5** times, and the average execution time and number of iterations were recorded (rounded up).
- **Datasets:** The experiments were conducted on five datasets of varying sizes and dimensionalities.

4.1.2 Dataset Overview

The datasets used in the evaluation vary in the number of points, dimensions, and clusters to examine how different factors impact performance.

Table 1: Dataset Characteristics

File Name	Points	Dimensions	Clusters	Description
dataset_2000_3_5.txt	2000	3	5	Small size, low-dimensional
dataset_5000_50_10.txt	5000	50	10	Medium size, high-dimensional
dataset_10000_20_10.txt	10000	20	10	Larger size, moderate-dimensional
dataset_20000_80_15.txt	20000	80	15	Large size, high-dimensional
dataset_50000_5_15.txt	50000	5	15	Massive size, low-dimensional

4.1.3 Result: Performance Comparison

The following table summarizes the results of the efficiency test. The execution time is highlighted in red to emphasize performance differences.

4.1.4 Analysis of Result

- **CUDA's Significant Speed Advantage:** The CUDA implementation consistently achieves faster execution times than both C and sklearn, particularly for larger datasets. While the execution time for C increases exponentially with the number of data points and dimensions, CUDA maintains a near-linear growth due to its parallel processing capabilities.

Table 2: Performance Comparison of CUDA, C, and Sklearn Implementations

File Name	**Iterations**			**Execution Time (ms)**		
Source	**C**	**CUDA**	**sklearn**	**C**	**CUDA**	**sklearn**
dataset_2000_3_5.txt	33	**27**	4	10	**4**	45
dataset_5000_50_10.txt	65	**95**	11	919	**146**	436
dataset_10000_20_10.txt	124	**122**	6	1321	**146**	271
dataset_20000_80_15.txt	160	**52**	13	18822	**544**	2254
dataset_50000_5_15.txt	101	**69**	6	3177	**100**	778

- **Scalability with Larger Datasets:** As dataset size and dimensionality increase, CUDA maintains a clear advantage. In `dataset_20000_80_15.txt`, CUDA executes around **35x faster** than C (544 ms vs. 18822 ms), demonstrating its scalability for high-dimensional data.
- **Iteration Efficiency vs. Execution Time:** While CUDA requires more iterations than sklearn, its parallel processing capabilities result in significantly reduced overall execution time. This highlights CUDA's ability to handle more iterations efficiently.

These results highlight the strengths of CUDA for large-scale, high-dimensional K-Means clustering, demonstrating its effectiveness in accelerating iterative computations.

4.2 Scalability Test

The purpose of this experiment is to evaluate how the CUDA implementation scales with different GPU configurations by varying the number of threads per block. By systematically increasing the number of threads, the test aims to identify the optimal configuration that maximizes performance before encountering resource saturation.

4.2.1 Test Setup

The scalability test was conducted using the dataset `dataset_10000_20_10.txt`. The execution time was measured while varying the number of threads per block across the following configurations: 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024. Each configuration was executed 10 times, and the average execution time was recorded. The expectation was that increasing the number of threads would enhance performance up to a certain threshold, beyond which additional threads would not provide significant speedup due to hardware constraints.

4.2.2 Results and Analysis

The Figure 1 indicates a clear trend in performance scaling:

- **Initial Performance Gain:** Execution time improves as the number of threads increases from 16 to 256. The highest improvement is observed when transitioning from 64 to 256 threads per block, where parallelism effectively reduces the workload per thread.

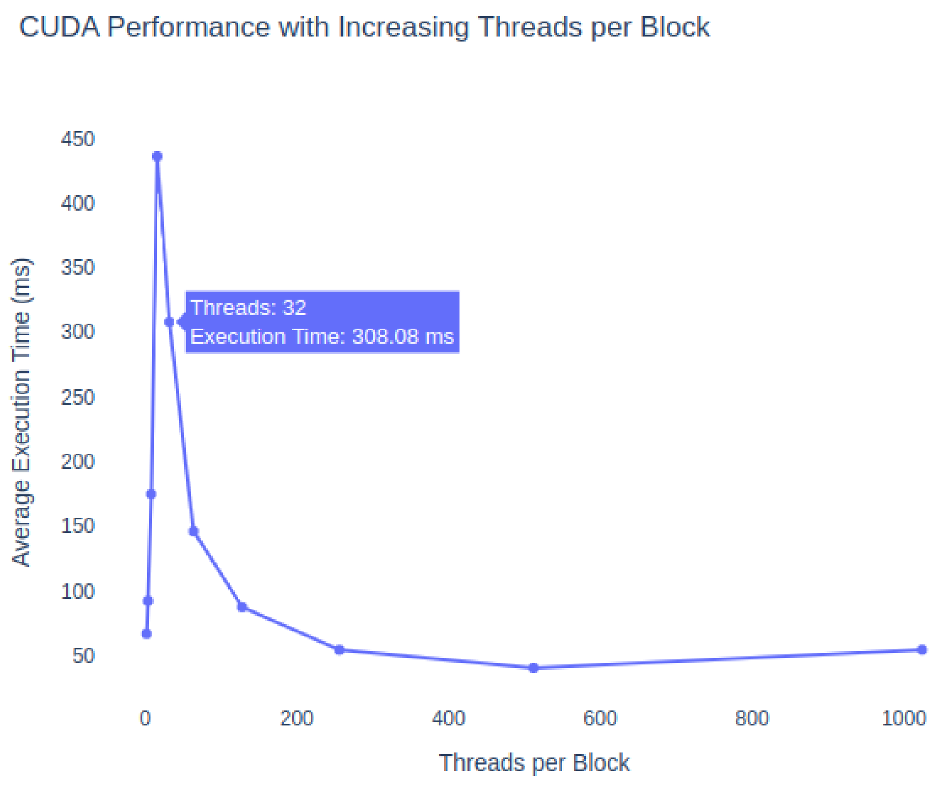

Figure 1: CUDA Performance Scaling with Increasing Threads per Block

- **Plateau Effect Beyond 256 Threads:** After 256 threads per block, further increases show negligible improvements. At 512 threads and beyond, execution time stabilizes, suggesting that GPU compute resources are fully utilized.
- **Diminishing Returns and Saturation:** While theoretically more threads should enhance parallelism, GPU hardware constraints such as shared memory, register availability, and multiprocessor occupancy limit further performance gains.

5 Conclusion

The CUDA-accelerated K-Means clustering implementation significantly improves performance over traditional CPU-based methods by leveraging parallel computation, shared memory optimization, and atomic operations. The efficiency test demonstrated substantial speedups, particularly for large-scale, high-dimensional datasets, with CUDA achieving better results compared to both C and sklearn implementation. These results underscore the effectiveness of GPU acceleration in handling computationally intensive clustering tasks, making CUDA-based K-Means a potential powerful solution for large-scale data analysis applications.