Accelerating K-means Clustering Through CUDA Parallel Computing

Lorenzo Benedetti
lorenzo.benedetti5@edu.unifi.it

Abstract

This paper presents a comparative analysis of sequential and parallel implementations of the K-means clustering algorithm, leveraging NVIDIA's CUDA architecture for parallel computation. The study focuses on evaluating the performance benefits of GPU acceleration across varying dataset dimensions, cluster sizes and number of features. Our implementation demonstrates significant speedup over the traditional sequential approach, particularly for high-dimensional datasets. The analysis includes detailed performance metrics, scaling behavior, and optimization strategies employed in the CUDA implementation. Results show that the parallel implementation achieves substantial performance improvements, with speedup factors varying based on data characteristics and dimensionality.

1. Introduction

The advent of big data and the growing demand for high-performance computing have amplified the need for efficient clustering algorithms, with K-means remaining a cornerstone in data analysis and machine learning. Despite its widespread use and conceptual simplicity, the computational cost of K-means can become a bottleneck when applied to large, high-dimensional datasets. Accelerating its execution is thus a critical challenge, especially for applications requiring real-time processing.

Building upon advancements in parallel computing, this work explores the use of NVIDIA's CUDA framework to significantly enhance the performance of K-means clustering. CUDA enables the execution of massively parallel computations by leveraging the capabilities of Graphics Processing Units (GPUs). Unlike traditional CPU-based approaches, CUDA's parallel architecture is particularly suited for data-intensive algorithms like K-means, where large-scale matrix operations and memory transfers play a crucial role.

The primary contributions of this work include:

- Development and evaluation of a CUDA-based parallel implementation of K-means clustering
- Analysis of performance gains compared to a sequential implementation across varying dataset characteristics
- Discussion of optimization strategies, including efficient memory usage and thread management in GPU computation

2. K-Means

2.1. Description

K-means clustering is an unsupervised learning algorithm that aims to partition n observations into k clusters, where each observation belongs to the cluster with the nearest mean (centroid). The algorithm operates in multidimensional space, making it suitable for a wide range of applications including image processing, customer segmentation, and pattern recognition. The algorithm's objective is to minimize the within-cluster sum of squares (WCSS), also known as the inertia. Mathematically, given a set of observations $(x_1, x_2, ..., x_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k sets

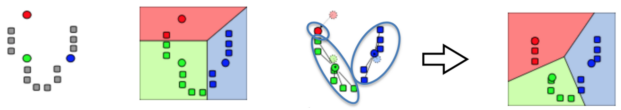

Figure 1. K-Means visualization

$S = S_1, S_2, ..., S_k$ to minimize:

$$\sum_{i=1}^{k} \sum_{x \in S_i} |x - \mu_i|^2 \quad (1)$$

where μ_i is the mean of points in S_i.

2.2. Algorithm

The K-means algorithm follows an iterative refinement approach:

Algorithm 1: K-Means Clustering

Input:

- Dataset $X = \{x_0, \ldots, x_{\text{N\_SAMPLES-1}}\}$
- N_SAMPLES
- N_FEATURE_LIST list of dimensions
- N_CLUSTERS
- MAX_ITER number of epochs

Output: Set of N_CLUSTERS

Assing each observation to the cluster with the nearest centroid Calculate the Euclidean distance

repeat

  Calculate the new centroid of each cluster;
  Update the mean for each cluster;

until *MAX_ITER is reached*;

The algorithm is guaranteed to converge to a local optimum, though this may not be the global optimum. The quality of the final clustering depends significantly on the initial centroid selection, the number of clusters k and the number of epochs.

3. Implementation Details

This section provides a detailed overview of the implementation of the sequential and CUDA-based k-means clustering algorithms. Key aspects of the implementation include the design of data structures, kernel functions, and strategies for memory management and computation. These components were designed to leverage the parallel capabilities of NVIDIA GPUs effectively.

3.1. Data Structures

Efficient data handling is critical to the performance of k-means clustering, especially in a GPU environment. To optimize memory access patterns, we implemented the following structures:

3.1.1 DatasetSoA

The `DatasetSoA` structure uses the Structure of Arrays (SoA) layout to store data points. This layout organizes all values of a specific dimension contiguously in memory, improving memory coalescing and cache utilization during GPU operations. Each point is accessed as follows:

$$\text{Index} = \text{dim\_index} \times \text{n\_points} + \text{point\_index}$$

This layout minimizes memory divergence, a critical factor in GPU performance.

3.1.2 CentroidsSoA

The `CentroidsSoA` structure, similar to `DatasetSoA`, stores centroids in an SoA format. It allows efficient updates and retrievals of centroid coordinates during the k-means iterations, using the same indexing scheme as the dataset.

3.2. Kernel Functions

The CUDA implementation of k-means employs two main kernel functions: `assign_points_kernel` for the assignment phase and `update_centroids_kernel` for updating the centroids. Both kernels are designed to exploit GPU parallelism and optimize memory usage.

3.2.1 Assignment Phase: `assign_points_kernel`

This kernel assigns each data point to the nearest centroid by calculating the squared Euclidean distance between the point and all centroids. Each

thread processes one data point, performing the following steps:

1. Load the point's coordinates from global memory.
2. Iterate over all centroids to compute distances.
3. Identify the nearest centroid and update the assignment array.

To minimize latency, centroid coordinates are stored in constant memory, and shared memory is used for temporary data. The kernel uses a 1D grid where each thread corresponds to a data point. The distance computation follows this pattern:

```
float diff = data[d * n_points + point_idx] -
    centroids[d * k + centroid_idx];
dist += diff * diff;
```

Listing 1. Distance Computation in Assignment Kernel

3.2.2 Update Phase: `update_centroids_kernel`

This kernel recalculates centroids by summing the coordinates of all assigned points in parallel and normalizing the result by the cluster size. Unlike the previous implementation, the updated version employs *shared memory* and *parallel reduction* to optimize performance, minimizing global memory accesses and improving scalability.

The kernel operates on a 2D grid:

- The first dimension corresponds to the cluster index.
- The second dimension corresponds to the dimensions of each centroid.

Each block computes the sum for a specific cluster and dimension. Threads within the block process subsets of points in parallel, storing intermediate sums in shared memory. The partial sums are then reduced using an efficient parallel reduction algorithm.

The main steps of the kernel are:

- Each thread computes a partial sum for its assigned data points and writes it to shared memory.
- A parallel reduction is performed within the shared memory to combine these partial results.
- The normalized centroid values are written back to global memory by the first thread in the block.

The normalization step ensures the final centroid coordinates are the average of all assigned points. This step is performed directly within the kernel, eliminating the need for post-processing on the CPU and further improving overall execution efficiency.

3.3. kmeans_sequential

This function serves as the baseline implementation of the k-means algorithm and is used for comparison against the CUDA implementation. It processes each data point sequentially, iterating through centroids to compute distances and assign points to the nearest cluster.

3.4. kmeans_cuda

The `kmeans_cuda` function is the core of the GPU-accelerated implementation. It leverages CUDA's parallel processing capabilities to assign data points to clusters and compute new centroids concurrently. Key features include:

- **Distance Computation:** Each thread computes the squared Euclidean distance between a single data point and all centroids. The results are stored in shared memory to minimize global memory access.
- **Cluster Assignment:** After computing distances, each thread determines the closest centroid and updates a global array with the assignment results.
- **Centroid Updates:** Atomic operations are used to accumulate the new centroid positions and cluster sizes across threads. This approach ensures correctness while avoiding race conditions.

3.5. Memory Management

Efficient memory management is crucial for GPU performance. Our implementation makes use of:

- **Global Memory:** Used for storing datasets, centroids, and cluster assignments. Memory access patterns are optimized to minimize divergence.
- **Shared Memory:** Allocated for storing intermediate values (e.g., distances) during kernel execution. This reduces latency compared to global memory.
- **Constant Memory:** Centroid coordinates are stored in constant memory during the distance computation phase, as they remain unchanged for most of the algorithm's execution.

3.6. Shared Memory in `updateCentroids`

The `updateCentroids` kernel utilizes shared memory to efficiently compute cluster centroids without requiring atomic operations. Shared memory is allocated dynamically as a 1D array using the `extern` keyword, but it is conceptually organized as a 2D array where rows correspond to clusters (`blockIdx.x`) and columns to dimensions (`blockIdx.y`).

3.6.1 Thread Responsibilities

Each thread (`threadIdx.x`) computes a partial sum for a specific cluster and dimension by iterating over assigned data points. These partial sums are stored in the shared memory array:

```
shared_sums[threadIdx.x] = sum;
```

Listing 2. updateCentroids partial sums

Afterward, threads synchronize to ensure all partial results are available.

3.6.2 Reduction within Blocks

Partial sums are combined using a parallel reduction within the shared memory:

```
for (int stride = blockDim.x / 2; stride > 0;
    stride >>= 1) {
    if (threadIdx.x < stride) {
        shared_sums[threadIdx.x] += shared_sums[
    threadIdx.x + stride];
    }
    __syncthreads();
}
```

Listing 3. updateCentroids parallel reduction

At the end of the reduction, the total sum for the cluster and dimension is stored in `shared_sums[0]`.

3.6.3 Shared Memory Visualization

To better understand the structure and utilization of shared memory in the `updateCentroids` kernel, consider how threads process data points assigned to clusters and dimensions. Each thread computes partial sums for points assigned to a specific cluster (`blockIdx.x`) and dimension (`blockIdx.y`). The following loop illustrates how threads handle their respective subsets of data points:

```
for (int i = threadIdx.x; i < n_points; i +=
    blockDim.x) {
    if (assignments[i] == cluster) {
        sum += data[dim * n_points + i];
    }
}
```

Listing 4. Loop for partial sums calculation

In this loop:

- `threadIdx.x` determines the starting point for each thread.
- `blockDim.x` ensures threads skip over chunks of data to evenly distribute work across the dataset.
- The check `assignments[i] == cluster` filters points belonging to the correct cluster, ensuring partial sums are computed accurately.

The partial sums computed by each thread are stored in shared memory for further reduction. Below, we provide a visualization of how threads operate within a block and their corresponding memory layout.

To further clarify the thread assignment and workload distribution, consider the following example. Suppose we have `blockDim.x = 8` (8 threads per block) and `n_points = 32` (32 total data points). The threads will process data points in the following manner:

- `threadIdx.x = 0` processes points: 0, 8, 16, 24.
- `threadIdx.x = 1` processes points: 1, 9, 17, 25.
- `threadIdx.x = 2` processes points: 2, 10, 18, 26.

⋮

- `threadIdx.x = 7` processes points: 7, 15, 23, 31.

The "stride" between consecutive points processed by the same thread is determined by `blockDim.x`. This ensures that the workload is evenly distributed among threads within the block. Each thread computes a partial sum for its assigned points and stores it in shared memory. These partial sums are then reduced within the block using a parallel reduction strategy.

C	D	T	Partial Sum
0	0	0	Sum of points 0, 8, 16, 24 assigned to c0
0	0	1	Sum of points 1, 9, 17, 25 assigned to c0
0	0	2	Sum of points 2, 10, 18, 26 assigned to c0
0	1	0	Sum of points 0, 8, 16, 24 assigned to c0, d1
⋮	⋮	⋮	⋮
1	0	0	Sum of points 0, 8, 16, 24 assigned to c1
1	0	1	Sum of points 1, 9, 17, 25 assigned to c1
⋮	⋮	⋮	⋮

Table 1. Visualization of shared memory structure in the `updateCentroids` kernel. Each thread computes a partial sum for a specific subset of points assigned to its cluster and dimension. The jump in points processed is determined by `blockDim.x`.

This structure ensures efficient parallel computation by dividing the workload among threads. The use of shared memory reduces global memory access latency during reduction, significantly improving performance.

3.6.4 Avoiding Atomic Operations

This approach avoids costly atomic operations on global memory by reducing all partial results locally within each block. The thread leader (`threadIdx.x == 0`) writes the final centroid value back to global memory after normalizing it by the cluster's point count:

```
if (threadIdx.x == 0 && counts[cluster] > 0) {
    centroids[dim * k + cluster] = shared_sums[0]
     / counts[cluster];
}
```

Listing 5. updateCentroids normalization

By leveraging shared memory for both intermediate computations and the reduction process, the kernel minimizes global memory accesses and ensures high computational efficiency.

3.7. Testing Infrastructure

All experiments were conducted on a laptop equipped with an AMD Ryzen 9 3900 Desktop Processor (3.10GHz) and an NVIDIA GeForce RTX 2070 (Notebook Edition). A Python-based framework was used to automate testing and analyze the performance of the sequential and CUDA implementations. Furthermore, it was used for dataset creation and performance metric visualization.

3.7.1 Dataset Generation

The framework utilizes scikit-learn's `make_blobs` function to generate synthetic datasets with controlled characteristics. For our experiments, we generated datasets with the following parameters:

```
x, y = make_blobs(
    n_samples=10000000,    # 10M data points
    # Different dimensionalities
    n_features=[2, 3, 8, 16, 32, 64, 128, 256],
    centers=64,            # Number of clusters
    random_state=42        # For reproducibility
)
```

This approach allows us to test our implementations across different data dimensionalities while maintaining consistent cluster characteristics. The generated datasets are saved in CSV format for subsequent use by the CUDA implementation.

3.7.2 Performance Metrics

Performance was measured using CUDA's built-in timers to record the execution time of each kernel. Metrics such as speedup and execution time were compared between the sequential and CUDA implementations. Visualizations were created using Python's `matplotlib` library to highlight trends across different dataset configurations.

4. Experimental Results

This study primarily focuses on speedup as the key performance metric while also considering raw execution times to provide a comprehensive performance analysis. The speedup is calculated as the ratio between sequential and parallel execution times for both Array of Structures (AoS) and Structure of Arrays (SoA) implementations.

4.1. Performance Metrics

Our experimental evaluation encompasses several dimensions of analysis:

- Dataset dimensionality (varying from 2D to high-dimensional data)
- Number of data points (scaling from small to very large datasets)
- Number of clusters (varying k-means parameter)

For each analysis, we measure and compare:

- Sequential execution time (T_{seq})
- Parallel execution time (T_{par})
- Speedup ratio ($S = T_{seq}/T_{par}$)

All measurements are averaged over multiple runs (10) to ensure statistical significance, tested under identical conditions. The following sections present detailed analyses of these measurements, with particular emphasis on the scalability characteristics across different problem dimensions.

4.2. Speedup Analysis

This section presents a detailed analysis of the speedup achieved across various experimental dimensions using CUDA parallelization. All experiments were conducted with a maximum of 30 iterations to ensure convergence. Unless otherwise specified, tests were performed using 1,000,000 data points, 32 clusters, and 2 dimensions.

4.2.1 Impact of Data Points

The impact of dataset size on speedup was evaluated by varying the number of points from 1,000 to 100 million, using 32 clusters. CUDA's implementation shows significant scaling advantages, with a clear relationship between dataset size and speedup efficiency:

Points	Speedup
1'000	0.808
10'000	6.504
100'000	31.732
1'000'000	37.661
10'000'000	51.524
100'000'000	**55.897**

Table 2. Speedup with varying number of points

- For small datasets (1,000 points), the overhead of kernel launches and memory transfers dominates, resulting in speedup below 1.
- Performance improves significantly with datasets exceeding 10,000 points, where CUDA's parallelization begins to demonstrate clear benefits.
- At 100 million points, the maximum observed speedup of 55.897x highlights CUDA's scalability for large-scale datasets.
- The results confirm that CUDA's efficient handling of massively parallel tasks and optimized memory access patterns excel as dataset size grows.

4.2.2 Impact of Dimensionality

Dataset dimensionality significantly influences CUDA's performance:

Dim	Speedup
2D	65.419
3D	55.673
8D	58.672
16D	65.313
32D	**75.996**

Table 3. Speedup with varying dimensions

- The speedup improves with increasing dimensionality, peaking at 32 dimensions with a maximum of 75.996x. This suggests effective parallelization of higher-dimensional data processing on the GPU.
- A slight dip in speedup is observed at 3D and 8D, possibly due to thread underutilization or suboptimal memory coalescing at these dimensionalities.
- Beyond 16D, the results show stable performance, indicating CUDA's ability to manage higher memory throughput requirements for larger feature vectors.

4.2.3 Impact of Cluster Count

The number of clusters (k) significantly impacts computational workload and speedup:

Clusters	Speedup
2	2.644
4	4.545
8	8.366
16	18.169
32	37.604
64	65.508
128	68.146
256	76.834
512	**84.936**

Table 4. Speedup with varying number of clusters

- Speedup improves as k increases, with the highest performance observed at $k = 512$ (84.936x). This indicates that CUDA effectively manages the increased computational complexity of cluster assignments.
- For lower cluster counts ($k = 2$ and $k = 4$), the speedup is limited due to lower parallel workload per thread.
- At $k \geq 128$, performance stabilizes as GPU resources are fully utilized.
- The results suggest that CUDA handles the additional memory and compute demands of large k values more efficiently than traditional CPU-based approaches.

4.3. Execution Times

The execution time analysis examines the performance of the CUDA-based K-means implementation compared to the CPU implementation, with a focus on the effect of varying dataset dimensionality. The dimensions tested include [2, 3, 4, 8, 16, 32, 64, 128, 256], using 1M data points and 32 clusters.

- The GPU consistently outperforms the CPU across all tested dimensionalities, with execution times that are approximately two orders of magnitude faster than those of the CPU implementation.
- The relative difference in execution times between the GPU and CPU remains consistent across dimensionalities, demonstrating the scalability and efficiency of the CUDA implementation for higher-dimensional datasets.
- The GPU execution times exhibit a gradual increase with higher dimensionalities, reflecting the additional computational workload, but the increase is minimal compared to the CPU, which scales less efficiently.

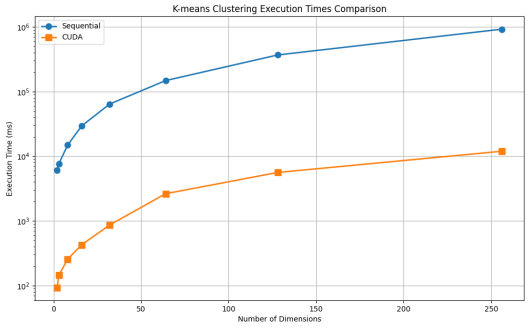

Figure 2. K-means execution times

5. Summary

This comprehensive analysis highlights the significant performance advantages of the CUDA-based K-means implementation over its CPU counterpart, particularly for large-scale and high-dimensional datasets.

The results demonstrate that the GPU implementation scales effectively with increasing workload, maintaining stable execution time differentials and achieving superior speedup ratios. These findings confirm the suitability of GPU acceleration for modern clustering applications, leveraging its parallel computing capabilities to achieve substantial reductions in execution time and improved scalability.

6. Appendix

6.1. Nvidia Nsight Compute

The use of Nvidia Nsight Compute proved instrumental in identifying a bottleneck in the initial implementation of the `updateCentroids` kernel. This bottleneck was caused by the inefficient logic used to sum the data points for each cluster and dimension, leading to suboptimal performance due to excessive global memory access and lack of parallel reduction.

The previous implementation of `updateCentroids` iterated over all data points in a single thread per dimension and cluster, as shown below:

```
__global__ void updateCentroids(const float* data
    , size_t n_points, size_t n_dims,
                                float* centroids,
    size_t k,
                                const int*
    assignments, const int* counts) {
    int c = blockIdx.x;
    int d = threadIdx.x;
    if (c >= k || d >= n_dims) return;
    float sum = 0.0f;
    for (int i = 0; i < n_points; i++) {
        if (assignments[i] == c) {
            sum += data[d * n_points + i];
        }
    }

    if (counts[c] > 0) {
        centroids[d * k + c] = sum / counts[c];
    }
}
```

Listing 6. Previous `updateCentroids` Kernel

By analyzing the kernel's performance profile with Nsight Compute, it became evident that the bottleneck stemmed from the serialized accumulation of data for each cluster and dimension, which failed to fully exploit the GPU's parallel processing capabilities.

To address this issue, the kernel logic was restructured to use *shared memory* and parallel reduction, as described in Section 3.6. The updated implementation, shown below, distributes the workload across threads and utilizes shared memory for efficient reduction:

```
__global__ void updateCentroids(const float* data
    , size_t n_points, size_t n_dims,
                                    float*
    centroids, size_t k,
                                    const int
    * assignments, const int* counts) {
    extern __shared__ float shared_sums[];

    int cluster = blockIdx.x;
    int dim = blockIdx.y;
    if (cluster >= k || dim >= n_dims) return;

    float sum = 0.0f;
    // Parallelize over points
    for (int i = threadIdx.x; i < n_points; i +=
    blockDim.x) {
        if (assignments[i] == cluster) {
            sum += data[dim * n_points + i];
        }
    }

    // Parallel reduction in shared memory
    shared_sums[threadIdx.x] = sum;
    __syncthreads();

    // Reduce within block
    for (int stride = blockDim.x / 2; stride > 0;
     stride >>= 1) {
        if (threadIdx.x < stride) {
```

```
            shared_sums[threadIdx.x] += shared_sums[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Write result
    if (threadIdx.x == 0 && counts[cluster] > 0)
    {
        centroids[dim * k + cluster] = shared_sums[0] / counts[cluster];
    }
}
```

Listing 7. Updated `updateCentroids` Kernel with Shared Memory

This new implementation leverages shared memory to store intermediate sums and employs a parallel reduction pattern to combine partial results efficiently. By minimizing global memory accesses and enabling concurrent computations, the updated kernel significantly improves execution time and scalability, especially for large datasets with many dimensions and clusters.

6.2. Source Code Availability

The complete source code for this K-means clustering implementation is publicly available and can be accessed through the GitHub repository at:
`github.com/lorenzo-27/kmeans-cuda`