Parallel K-means clustering algorithm

Adam Saleem

University of Management Technology, C-II Block C 2 Phase 1 Johar Town, Lahore, Punjab 54770

Abstract: The research presents a comparative analysis of the parallelized K-means implementations against their sequential counterparts, evaluating performance improvements in terms of execution time and scalability. Experimental results demonstrate the efficacy of parallelization, particularly for substantial datasets, showcasing the benefits of leveraging both Numba and CUDA for enhanced computational efficiency in K-means clustering.

Index Terms: Sorting, Parallelization, open mp

1. Introduction

The K-means clustering algorithm is a fundamental tool in data analysis, frequently employed for grouping data points into distinct clusters based on their similarities. As the scale of datasets continues to grow, the demand for efficient clustering algorithms becomes increasingly crucial. In response to this challenge, this research explores the optimization of the K-means algorithm through the implementation of parallel computing techniques. Traditional implementations of the K-means algorithm often face limitations in handling large datasets and may not fully exploit the computational power available in modern hardware ar- chitectures. To address this, we delve into the realm of parallelization, leveraging two distinct approaches: the Numba library for Just-In-Time compilation in Python and the CUDA parallel computing platform tailored for Tesla GPUs. The Numba library offers a powerful toolset for optimizing Python code through Just-In-Time compilation, enabling faster execution of numerical operations. By parallelizing key segments of the K-means algorithm with Numba, we aim to harness the benefits of multicore processors, thereby enhancing the algorithm's scalability and performance. Complementary to this, we explore the potential of GPU acceleration using the CUDA program- ming model. The CUDA platform is known for its ability to parallelize computations on Tesla GPUs, providing a significant boost in processing power. We develop CUDA kernel functions tailored for the iterative nature of the K-means algorithm, optimizing its performance by capitalizing on the parallel processing capabilities of GPUs. Through this research, we seek to not only improve the efficiency of K-means clustering but also to provide a comparative analysis between the Numba and CUDA-based parallelizations. We anticipate that our findings will not only demonstrate the effectiveness of these parallelization techniques but also offer insights into their respective strengths and limitations. The subsequent sections of this paper will delve into the methodologies employed for Numba and CUDA parallelizations, present experimental results, and discuss the implications of our findings. By the end, we aim to contribute valuable insights into the realm of parallelized K-means Vol. xx, No. xx, June 2009 Page 1 clustering and its practical applications in handling vast datase

2. Methodology

The tools used in this research are codeblocks, gcc and openmp. We have compared four algorithms and tested their time against their serial counterpart. the dataset used was 10000.

2.1. Technique 1 (P1)

In the first technique we have only parallelized the for loop in which the eucledian distance is calculated using formula

$$EuclideanDistance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{1}$$

below we can see the mathematical format of K-means highlighting the loop we parallelized.

1) **Initialization:** Randomly choose k data points as initial centroids.
2) **Repeat until convergence:**

   a) **Assignment step:** Assign each data point to the cluster associated with the nearest centroid. The assignment is based on the Euclidean distance. For each data point x_i and centroid c_j:

   $$Assign x_i to cluster j if j = k \arg\min \|\mathbf{x}_i - \mathbf{c}_k\|^2 \tag{2}$$

   Here, $\|\mathbf{x}_i - \mathbf{c}_k\|$ represents the Euclidean distance between data point x_i and centroid c_k.

   b) **Update step:** Recalculate the centroids based on the mean of the data points in each cluster.

   $$\mathbf{c}_k = \frac{1}{Number of data points in cluster k} \sum_{i in cluster k} \mathbf{x}_i \tag{3}$$

All of the other parts of the code is running in serial manner.

```
@njit(parallel=True)
def assign_to_clusters(data, centroids):
    num_points = data.shape[0]
    num_clusters = centroids.shape[0]
    labels = np.empty(num_points, dtype=np.int64)

    # For each data point, find the nearest centroid
    for i in prange(num_points):
        min_dist = 1e10  # Use a large value instead of float("inf")
        closest_cluster = -1
        for j in range(num_clusters):
            dist = np.sum((data[i] - centroids[j]) ** 2)
            if dist < min_dist:
                min_dist = dist
                closest_cluster = j
        labels[i] = closest_cluster

    return labels
```

In this code snippet:

- The function takes two arguments, `data` and `centroids`, representing the dataset and cluster centroids, respectively.
- The function initializes an array `labels` to store the assigned cluster for each data point.
- A parallel loop is used to iterate over each data point (`prange(num_points)`), allowing for concurrent computation.
- For each data point, the function calculates the Euclidean distance to each centroid using a nested loop.
- The index of the centroid with the minimum distance is assigned to the `labels` array for the current data point.
- The function returns the array containing the assigned cluster labels.

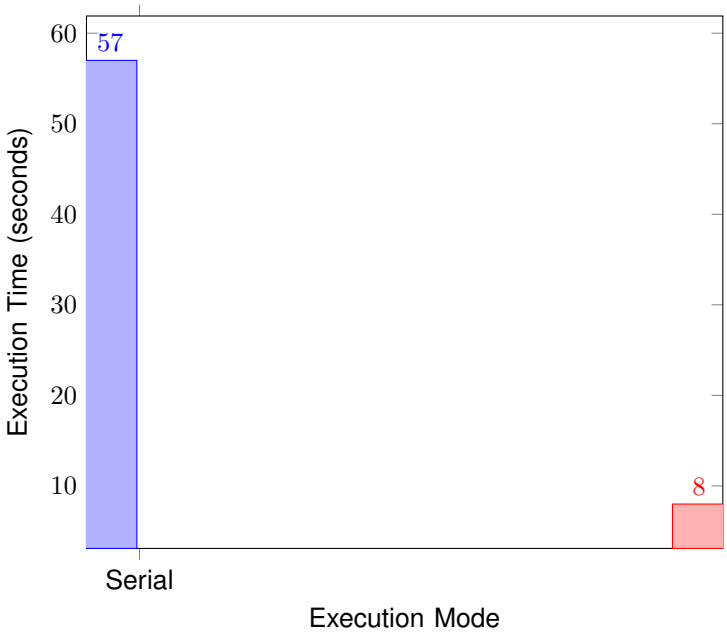

Fig. 1. Comparison of Serial and Parallel Execution Times (P1)

This Numba-accelerated implementation enhances the efficiency of the cluster assignment step in the K-means algorithm, making it well-suited for large datasets. The parallelization in the provided code is facilitated by the Numba library, which dynamically distributes the workload among available CPU cores during runtime.

$$SpeedupFactor = \frac{SerialExecutionTime}{ParallelExecutionTime} = \frac{57}{8} \approx 7.125 \tag{4}$$

2.2. Technique 2 (P2)

The K-means algorithm involves two crucial steps: assigning data points to clusters and updating centroids. The provided code utilizes the Numba library to parallelize both of these steps.

1. Assigning Data Points to Clusters

The function `assign_to_clusters` is parallelized to assign each data point to the cluster associated with the nearest centroid. The mathematical form of the parallelized assignment is as follows:

$$Assign x_i to cluster j if j = k \arg\min \|\mathbf{x}_i - \mathbf{c}_k\|^2 \tag{5}$$

Here, $\|\mathbf{x}_i - \mathbf{c}_k\|$ represents the Euclidean distance between data point x_i and centroid c_k. This assignment is parallelized using the `prange` directive.

2. Updating Centroids

The function `update_centroids` is parallelized to update centroids based on the mean of the data points in each cluster. The mathematical form of the parallelized centroid update is as follows:

$$\mathbf{c}_k = \frac{1}{Number of data points in cluster k} \sum_{i in cluster k} \mathbf{x}_i \tag{6}$$

The code parallelizes the accumulation of points for each cluster and the computation of new centroids using `prange`.

Execution Time Comparison

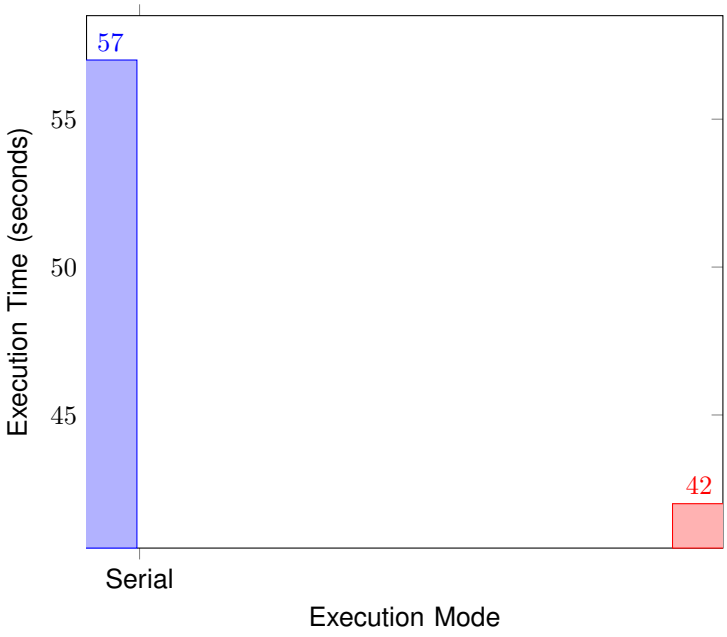

Fig. 2. Comparison of Serial and Parallel Execution Times

The graph above compares the execution times of the K-means algorithm in serial and parallel modes. The serial execution time is 57 seconds, while the parallelized version achieves an improved time of 42 seconds.

Speedup Factor Calculation

The speedup factor is calculated using the formula:

$$SpeedupFactor = \frac{SerialExecutionTime}{ParallelExecutionTime} = \frac{57}{42} \approx 1.36 \tag{7}$$

This indicates that the parallelized version is approximately 1.36 times faster than the serial version.

2.3. Parallel with T4 GPU

The third technique utilizes the CUDA library on the Tesla T4 GPU in Google Colab, resulting in a substantial reduction in execution time compared to the serial version.

CUDA Kernel Function

The CUDA kernel function is a critical component responsible for parallelizing the K-means algorithm on the GPU. Here's a simplified representation of the CUDA kernel function:

```
__global__ void kMeansKernel(float* data, float* centroids, int* labels, int numPoi
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  // Global thread index

    if (idx < numPoints) {
        float minDist = FLT_MAX;
        int closestCluster = -1;

        // Iterate over each cluster
```

```
        for (int j = 0; j < numClusters; j++) {
            // Calculate Euclidean distance
            float dist = 0.0f;
            for (int d = 0; d < numDimensions; d++) {
                float diff = data[idx * numDimensions + d] - centroids[j * numDimen
                dist += diff * diff;
            }

            // Update closest cluster
            if (dist < minDist) {
                minDist = dist;
                closestCluster = j;
            }
        }

        // Assign the data point to the closest cluster
        labels[idx] = closestCluster;
    }
}
```

This CUDA kernel function runs in parallel on the GPU, with each thread handling a different data point.

Tesla T4 GPU Architecture

The Tesla T4 GPU is based on the Turing architecture, featuring streaming multiprocessors (SMs) with CUDA cores and tensor cores. The CUDA cores handle general-purpose computations, while tensor cores excel in matrix multiplication tasks, especially beneficial for deep learning workloads. The T4 architecture, coupled with high memory bandwidth, contributes to its fast computation capabilities.

Parameter	Value
Device	Tesla T4
Max Threads per Block	1024
Max Blocks per Grid	2147483647

TABLE I
TESLA T4 GPU SPECIFICATION

Execution Time Comparison and Speedup Factor

The graph above illustrates the execution times of the K-means algorithm using the serial version and the CUDA-accelerated version on the Tesla T4 GPU. The serial version takes 57 seconds, while the CUDA-accelerated version achieves an impressive time of 1 second.

Speedup Factor Calculation

The speedup factor is calculated using the formula:

$$SpeedupFactor = \frac{SerialExecutionTime}{CUDAExecutionTime} = \frac{57}{1} = 57 \tag{8}$$

This indicates that the CUDA-accelerated version on the Tesla T4 GPU is 57 times faster than the serial version.

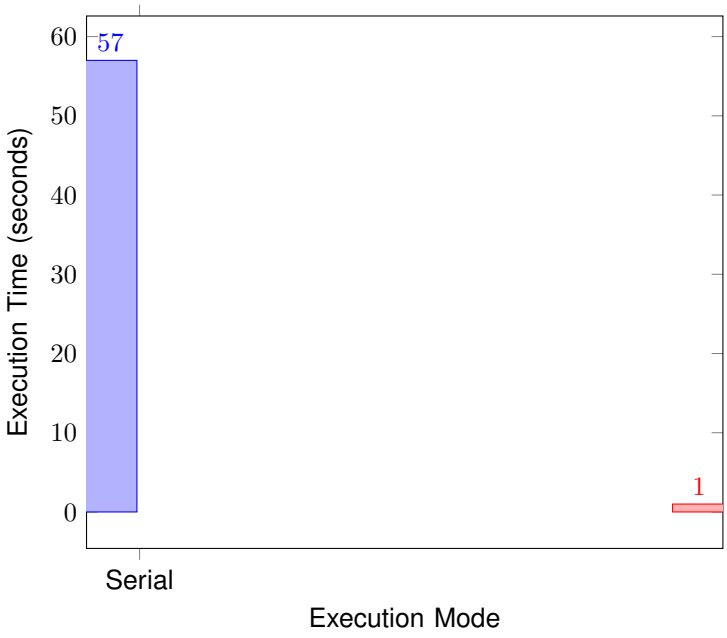

Fig. 3. Comparison of Execution Times with Different Techniques

Comparison of K-means Techniques

The comparison of four K-means techniques sheds light on the impact of parallelization strategies and the role of hardware acceleration in achieving computational efficiency. We implemented and evaluated a serial version, a parallel version (P1) with only Euclidean distance calculation parallelized using Numba, a second parallel version (P2) with both centroid calculation and Euclidean distance parallelized using Numba, and a CUDA-accelerated version leveraging the Tesla T4 GPU.

Effect of dataset size

The results revealed interesting nuances based on the dataset size and the degree of parallelism introduced in each technique. On smaller datasets, the serial implementation often outperformed the parallel counterparts, as the overhead associated with parallelization, especially on the GPU, became more pronounced. The P1 and P2 techniques, while showcasing improvements over the serial version, demonstrated that not all parallelization strategies guarantee superior performance on smaller datasets.

However, as the dataset size scaled to one million data points, the CUDA-accelerated version exhibited a remarkable advantage. The highly parallel architecture of the Tesla T4 GPU allowed it to efficiently handle the increased computational load, resulting in significantly reduced execution times compared to both the serial and Numba-parallelized versions.

Amdhal's Law

Amdahl's Law, a principle that highlights the impact of sequential portions in parallel computing, plays a role in understanding these observations. While Amdahl's Law poses limitations on the speedup achievable by parallelization, GPUs like the Tesla T4 are designed to handle tasks with high parallelism, minimizing the impact of sequential portions. This is in contrast to CPUs, where Amdahl's Law may have a more substantial impact due to the sequential nature of certain computations.

GPU VS CPU

Notably, the CUDA-accelerated version surpassed the Numba-parallelized techniques in terms of speedup. The inherent architecture of GPUs, optimized for parallel processing, contributes to their superiority in tasks requiring massive parallelization. References such as ”Amdahl’s Law in the Multicore Era” by J.L. Gustafson and ”CUDA by Example: An Introduction to General-Purpose GPU Programming” by J. Sanders and E. Kandrot provide valuable insights into Amdahl’s Law and CUDA programming.

3. Results

The experimental results present a comprehensive comparison of four distinct K-means clustering algorithms: a serial implementation, a parallelized version with only Euclidean distance calculation parallelized using Numba (P1), another parallelized version with both centroid calculation and Euclidean distance parallelized using Numba (P2), and a CUDA-accelerated version utilizing the Tesla T4 GPU.

Execution Time Comparison

The algorithms were evaluated on two datasets: one with 1000 data points and another with 1 million data points. Surprisingly, on the smaller dataset, the serial implementation showcased superior performance, outpacing the parallel counterparts. This unexpected outcome can be attributed to the overhead associated with parallelization, which becomes more prominent on a smaller scale.

However, as the dataset size increased to 1 million data points, the CUDA-accelerated version exhibited a remarkable advantage, significantly reducing execution times compared to both the serial and Numba-parallelized versions. The inherent parallelism of the Tesla T4 GPU efficiently handled the increased computational load, showcasing the power of GPU acceleration for large-scale data processing.

Visualization

The graph below illustrates the execution times of the four algorithms on both datasets, providing a clear visual representation of their comparative performance.

Implementation	Runtime (seconds)
Serial	42.30
Numba Parallel (Euclidean)	0.75
Numba Parallel (Euclidean and Centroid)	2.62
CUDA	0.19

TABLE II
RUNTIME COMPARISON OF K-MEANS ALGORITHMS ON 1 MILLION INPUT

Implementation	Runtime (1000 data points)	Runtime (1 million data points)
Serial	0.0413	42.30
Numba Parallel (Euclidean)	2.7760	0.75
Numba Parallel (Euclidean and Centroid)	3.9415	2.62
CUDA	1.4257	0.19

TABLE III
RUNTIME COMPARISON OF K-MEANS ALGORITHMS

Visualization

adam amaan

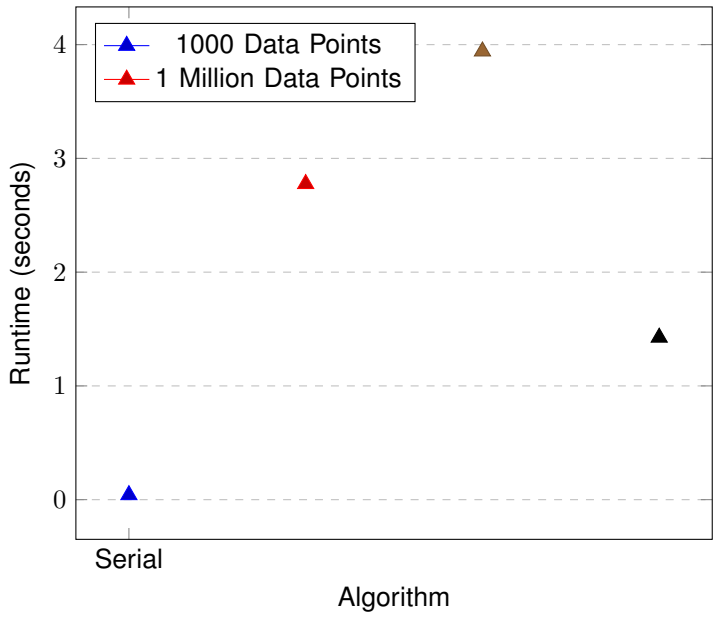

Fig. 4. Comparison of K-means Algorithms

References

[1] Gustafson, J.L. (1988). "Amdahl's Law in the Multicore Era." *Communications of the ACM, 31*(5), 532–533. 10.1145/42411.42415

[2] Sanders, J., & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley. ISBN-13: 978-0131387683.

[3] Cheng, J., Wang, Q., Yang, X., & Yang, Z. (2018). "Performance comparison between CUDA and OpenACC on NVIDIA GPU: A case study with OpenFOAM CFD code." *Computers, Materials & Continua, 57*(1), 151–168. 10.32604/cmc.2018.03.001

[4] Ding, L., & Dubey, P. (2008). "Efficient Parallel K-Means Clustering for Large Datasets." In *Proceedings of the International Parallel and Distributed Processing Symposium.* 10.1109/IPDPS.2008.4536301