Parallel Computing: Implementation of K-Means Algorithm using CUDA

Annalisa Ciuchi
E-mail address
annalisa.ciuchi@edu.unifi.it

Abstract

In this work, the implementation of K-Means algorithm using CUDA is presented. The main objective is to evaluate the effectiveness of the parallelized version on a GPU in improving performance compared to a sequential implementation. After a brief introduction to the functioning of K-Means Algorithm, the CUDA implementation is described. Results were then obtained on datasets of various sizes and complexities. The performance was analyzed, achieving improvements in terms of speedup.

1. Introduction to K-Means Algorithm

K-Means is an unsupervised clustering algorithm aimed at partitioning a dataset into k clusters, minimizing the within-cluster variance. The goal is to divide a dataset into K groups (clusters), where each data point belongs to the cluster with the nearest centroid. This process helps organize the data to identify underlying patterns or structures. K-Means is based on an iterative approach and can be summarized by the following pseudocode.

1.1. Pseudocode

Algorithm 1 K-Means

1: Select K points as initial centroids.
2: **repeat**
3: Form K clusters by assigning each point to the nearest centroid.
4: Recalculate the centroids of each cluster.
5: **until** Centroids changes becomes negligible.

1.2. Fundamental steps of the K-Mean algorthim

The K-Means algorithm involves the following fundamental steps:

1. **Initialization:** Randomly select K data points from the dataset as the initial centroids.
2. **Assignment:** For each data point in the dataset:
   - Calculate the distance to all centroids (e.g., Euclidean distance).
   - Assign the data point to the cluster represented by the nearest centroid.
3. **Update:** Recalculate the centroid of each cluster by computing the mean of all data points assigned to that cluster. The new centroid represents the updated cluster center.
4. **Convergence Check:** Repeat the *Assignment* and *Update* steps until one of the following conditions is met:
   - A predefined maximum number of iterations is reached.
   - Clusters centroids changes becomes negligible.

2. Sequential Implementation

The implementation is written in C++ and in the following sections it's been described.

2.1. Representation of Data Points

Data points are represented using a custom-defined structure called Punto (Point), which encapsulates the following:

- **Coordinates**: Stored as a std::vector<double> to represent multidimensional data.
- **Cluster ID**: An integer (int) that indicates the cluster to which the point is assigned.

2.2. Representation of Datasets and Centroids

- **Dataset**: A std::vector<Punto> is used to store the dataset.
- **Centroids**: Represented as a separate std::vector<Punto> initialized randomly.

2.3. Defined Functions

The sequential version of the K-Means algorithm includes several functions, each with a specific purpose in the clustering process. Below is a description of each function along with its implementation:

2.3.1 Sequential function

The function `sequential_kmeans` implements the K-Means clustering algorithm in a sequential manner. Below is a detailed explanation of its structure and logic.

Input Parameters:

- ds: A vector of Punto objects representing the dataset to be clustered.
- centroidi: A vector of Punto objects representing the initial centroids of the clusters.
- k: The number of clusters.

Steps:

- Initialization:
  - The iteration counter (iter) is set to 0.
  - A convergence flag (convergenza) is initialized to false.
  - A copy of the dataset from the previous iteration (precedente_ds) is created for comparison.
  - The number of points and their dimensionality are determined.
- Main Loop: The algorithm iterates until convergence.
  - Each point in the dataset is assigned to the nearest centroid based on the minimum distance using the function `assegnamento()`
  - Centroids are recalculated based on the mean position of the points assigned to each cluster using the function `aggiornamentoCentroidi()`
  - Convergence check: The algorithm checks whether the cluster centroids have changed compared to the previous iteration. If no changes occur, or if the changes are less than a specified tolerance threshold, the algorithm converges. If there is no convengence, the previous centroids is updated to the current centroids for future comparisons.

These functions are described in the next sections.

- Result: The function terminates when the cluster assignments stabilize (no changes between iterations) and it returns a tuple containing the updated dataset and final centroids.

Code:

```
std::tuple<std::vector<Punto>, std::vector<Punto>>
      sequential_kmeans(std::vector<Punto> ds, std::vector<Punto>
      centroidi, int k, int maxIter, double tol) {
    int iter = 0;
    bool convergenza = false;
    bool prima_iter = true;
    std::vector<Punto> centroidi_precedenti=centroidi;
    std::vector<Punto> precedente_ds;

    const auto numPunti = ds.size();
    const auto dimPunti = ds[0].dimensioni.size();
    std::vector<int> count(k);

    while (!convergenza) {
        assegnamento(ds, centroidi, k, numPunti, dimPunti);
        aggiornamentoCentroidi(ds, centroidi, k, numPunti, dimPunti,
      count);

        iter++;

        if (iter >= maxIter || controlloCluster(centroidi,
      centroidi_precedenti, k, tol)) {
            convergenza = true;
        } else {

            centroidi_precedenti = centroidi;
            precedente_ds = ds;

        }
    }

    return {ds, centroidi};
}
```

Listing 1. Sequential K-Means Function

2.3.2 Cluster Assignment (`assegnamento`)

This function assigns each data point in the dataset to the nearest centroid.

Description:

- Iterates through all data points and centroids.
- Computes the Euclidean distance between each point and each centroid.
- Updates the cluster ID of each point based on the closest centroid.

Code:

```
void assegnamento(std::vector<Punto> &ds, std::vector<Punto> &centroidi,
                  int k, const unsigned long numPunti,
                  const unsigned long dimPunti) {

    int etichetta_cluster = -1;
    double distanza = std::numeric_limits<double>::max();
    double minDistanza;
    for (int i = 0; i < numPunti; i++) {
        minDistanza = std::numeric_limits<double>::max();
        for (int j = 0; j < k; j++) {
            distanza = 0;
            for (int h = 0; h < dimPunti; h++) {
                distanza += pow(ds[i].dimensioni[h] -
                centroidi[j].dimensioni[h], 2);
            }
            distanza = sqrt(distanza);
            if (distanza < minDistanza) {
                minDistanza = distanza;
                etichetta_cluster = j;
            }
        }
        ds[i].cluster_id = etichetta_cluster;
    }
}
```

Listing 2. Cluster Assignment Function

2.3.3 Centroid update (`aggiornamentoCentroidi`)

This function updates the positions of centroids recalculating each centroid's coordinates based on the mean of all data points assigned to the corresponding cluster.

Steps:

- **Initialize the count vector**: All elements of `count` (recalling the number of points assigned to each centroid) are set to 0 using std::fill (to reset the point counts for each cluster).
- **Reset Centroid Coordinates**: For each centroid, all dimensions of its `dimensioni` vector are set to 0 in order to prepare to calculate new centroid positions.
- **Accumulate Data Points into Centroids**: Iterate through each point in the dataset (`ds`): For each dimension h of the point, add the coordinate to the corresponding centroid's dimention (based on the point's `cluster_id`). Simultaneously, increment the count of points assigned to the centroid (`count[cluster_id]`).
- **Calculate New Centroid Positions**: For each centroid, calculate the mean for each dimension by dividing the sum of coordinates by the number of points assigned to the cluster (stored in `count`).

Code:

```
void aggiornamentoCentroidi(std::vector<Punto> &ds, std::vector<Punto>
      &centroidi, int k, const unsigned long numPunti, const unsigned
      long dimPunti, std::vector<int> &count) {

    std::fill(count.begin(), count.end(), 0);

    for (int j = 0; j < k; j++) {
        std::fill(centroidi[j].dimensioni.begin(),
       centroidi[j].dimensioni.end(), 0);
    }

    for (int i = 0; i < numPunti; i++) {
        for (int h = 0; h < dimPunti; h++) {
            centroidi[ds[i].cluster_id].dimensioni[h] +=
       ds[i].dimensioni[h];
        }
        count[ds[i].cluster_id]++;
    }

    for (int j = 0; j < k; j++) {
        for (int h = 0; h < dimPunti; h++) {
            centroidi[j].dimensioni[h] /= count[j];
        }
    }
}
```

Listing 3. Centroids Updates Function

2.4. Convergence Check `controlloCluster`

During the convergence check phase this function checks whether the K-Means clustering algorithm has converged by comparing the current cluster centroids with the previous ones. The convergence is determined based on a tolerance threshold (`tol`).

Description:

- The function Iterate through all clusters (k).
- For each cluster, compute the Euclidean shift between the new centroid (`centroidi[i]`) and the previous centroid (`centroidi_precedenti[i]`)
- Check if the shift (normalized by the number of dimensions of the centroid) exceeds the tolerance
- If the normalized shift is greater than the specified tolerance, it means the clusters are still changing, so the function returns false (not yet converged).
- If all clusters are stable (i.e., their shifts are within tolerance), the function returns true, indicating convergence.

Code:

```
bool controlloCluster(const std::vector<Punto>& centroidi, const
      std::vector<Punto>& centroidi_precedenti, int k, double tol) {

    for (int i = 0; i < k; i++) {
        double shift = 0.0;
        for (size_t d = 0; d < centroidi[i].dimensioni.size(); d++) {
            double diff = centroidi[i].dimensioni[d] -
       centroidi_precedenti[i].dimensioni[d];
            shift += diff * diff;
        }
        if ((std::sqrt(shift)/centroidi[i].dimensioni.size()) > tol) {
            //std::cout << "Cluster non ancora convergente.\n";
            return false;
        }
    }

    return true;
}
```

Listing 4. Centroids Convergence check

3. CUDA Parallel Implementation

The sequential implementation of K-Means can be computationally expensive, especially for large datasets. To improve performance, the implementation of a parallelized version of K-Means using CUDA is presented, leveraging GPU acceleration to achieve significant speedup over the sequential approach. The CUDA implementation exploits GPU parallelism by executing the most computationally expensive operations in parallel.

3.1. Data Preparation for CUDA

Before executing K-Means on the GPU, the data is organized into 1D (linearized) arrays on the CPU, for example:

```
std::vector<double> h_punti(numPunti * dim);
for (int i = 0; i < numPunti; i++) {
    for (int d = 0; d < dim; d++) {
        h_punti[i * dim + d] = dataset[i].dimensioni[d];
    }
}
```

Listing 5. Data preparation

This code converts the points from a data structure into a onedimensional vector h_punti, and the same is done for the centroids as shown below.

```
std::vector<double> h_centroidi(numCentroidi * dim);
for (int c = 0; c < numCentroidi; c++) {
    for (int d = 0; d < dim; d++) {
        h_centroidi[c * dim + d] = centroidiIniziali[c].dimensioni[d];
    }
}
```

Listing 6. Data preparation

Now the points and centroids are in a GPU-compatible format.

3.2. GPU Memory Allocation

Memory areas are allocated on the GPU (usign CudaMalloc) for the data:

- d_punti: Linear matrix of points.
- d_centroidi: Linear matrix of centroids.
- d_assegnamenti: Vector to store the assignment of each point to a cluster.

Code:

```
double* d_punti;
double* d_centroidi;
int* d_assegnamenti;
CUDA_CHECK(cudaMalloc(&d_punti, numPunti * dim * sizeof(double)));
CUDA_CHECK(cudaMalloc(&d_centroidi, numCentroidi * dim *
      sizeof(double)));
CUDA_CHECK(cudaMalloc(&d_assegnamenti, numPunti * sizeof(int)));
```

Listing 7. GPU Memory Allocation

The CUDA Error Handling Macro is also defined. This macro ensures that all CUDA function calls are checked for errors, making debugging easier.

```
#define CUDA_CHECK(call) \
    do { \
        cudaError_t err = call; \
        if (err s cudaSuccess) { \
            std::cerr << "CUDA␣error␣in␣" << __FILE__ << "␣line␣" <<
      __LINE__ << ":␣" \
                << cudaGetErrorString(err) << std::endl; \
            exit(EXIT_FAILURE); \
        } \
    } while (0)
```

Listing 8. CUDA error Handling Macro

3.3. Copying Data from CPU to GPU

The data is then transferred to the GPU using cudaMemcpyHostToDevice that copies the data from the CPU vectors (h_punti, h_centroidi) to GPU memory.

```
CUDA_CHECK(cudaMemcpy(d_punti, h_punti.data(), numPunti * dim *
      sizeof(double), cudaMemcpyHostToDevice));
CUDA_CHECK(cudaMemcpy(d_centroidi, h_centroidi.data(), numCentroidi *
      dim * sizeof(double), cudaMemcpyHostToDevice));
```

Listing 9. GPU Memory Allocation

3.4. Executing the K-Means Algorithm

To execute k-means algorithm the function kmeans_cuda, which implements K-Means on the GPU, was defined.

```
kmeans_cuda(d_punti, d_centroidi, d_assegnamenti, numPunti,
      numCentroidi, dim, 1000, 0.001, h_centroidiPrecedenti,
      h_centroidiCorrenti);
CUDA_CHECK(cudaDeviceSynchronize());
```

Listing 10. K-means cuda function

Main parameters are:

1. numPunti, numCentroidi, dim: Define the problem size.
2. 1000: Maximum number of iterations.
3. 0.001: Convergence criterion (if centroids change less than this threshold, the algorithm stops).

cudaDeviceSynchronize() is used to ensure that all GPU computations are completed before proceeding. Then computed results are transferred back to the CPU usign cudaMemcpyDeviceToHost:

```
CUDA_CHECK(cudaMemcpy(h_centroidi.data(), d_centroidi, numCentroidi *
      dim * sizeof(double), cudaMemcpyDeviceToHost));
CUDA_CHECK(cudaMemcpy(h_assegnamenti.data(), d_assegnamenti, numPunti *
      sizeof(int), cudaMemcpyDeviceToHost));
```

Listing 11. K-means cuda function

h_centroidi will contain the final centroids and h_assegnamenti will store the final assignment of each point.
Then GPU Memory Cleanup is done using cudaFree to prevent memory leaks.

```
CUDA_CHECK(cudaFree(d_punti));
CUDA_CHECK(cudaFree(d_centroidi));
CUDA_CHECK(cudaFree(d_assegnamenti));
```

Listing 12. Memory Cleanup

3.5. kmeans_cuda function

This function orchestrates the K-Means iterations, executing the kernels in a loop until convergence. The first step allocates memory for cluster sizes, new centroids, and define gridSize and blockSize. Then, initialize variables and loop until convergence or maxIter is reached. After that, inside the loop, the kernels assegnaClustes, aggiornaCentroidi and normalizzaCentroidi are executed. Then convergence is checked by comparing the shift in centroids against tol. In the end this function copy results back to the host and frees allocated memory.

```
void kmeans_cuda(double* d_punti, double* d_centroidi, int*
      d_assegnamenti,
    int numPunti, int numCentroidi, int dimensioni, int maxIter, double
      tol,
    std::vector<double>& h_centroidiPrecedenti, std::vector<double>&
      h_centroidiCorrenti) {

    int* d_grandezzeCluster;
    double* d_nuoviCentroidi;
    CUDA_CHECK(cudaMalloc(&d_grandezzeCluster, numCentroidi *
      sizeof(int)));
    CUDA_CHECK(cudaMalloc(&d_nuoviCentroidi, numCentroidi * dimensioni *
      sizeof(double)));

    int blockSize = 256;
    int gridSizePunti = (numPunti + blockSize - 1) / blockSize;
    int gridSizeCentroidi = (numCentroidi + blockSize - 1) / blockSize;

    // Inizializza numero di iterazioni e convergenza
    bool convergenza=false;
    int iter = 0;

    while(!convergenza){

        if (iter >= maxIter) {
            break;
        }
```

```
        CUDA_CHECK(cudaMemset(d_grandezzeCluster, 0, numCentroidi *
       sizeof(int)));
        CUDA_CHECK(cudaMemset(d_nuoviCentroidi, 0, numCentroidi *
       dimensioni * sizeof(double)));

        assegnaClusters << <gridSizePunti, blockSize >> > (d_punti,
       d_centroidi, d_assegnamenti, numPunti, numCentroidi, dimensioni);
        CUDA_CHECK(cudaDeviceSynchronize());

        aggiornaCentroidi << <gridSizePunti, blockSize >> > (d_punti,
       d_nuoviCentroidi, d_assegnamenti, numPunti, numCentroidi,
       dimensioni, d_grandezzeCluster);
        CUDA_CHECK(cudaDeviceSynchronize());

        normalizzaCentroidi << <gridSizeCentroidi, blockSize >> >
       (d_centroidi, d_nuoviCentroidi, d_grandezzeCluster, numCentroidi,
       dimensioni);
        CUDA_CHECK(cudaDeviceSynchronize());

        CUDA_CHECK(cudaMemcpy(h_centroidiCorrenti.data(), d_centroidi,
       numCentroidi * dimensioni * sizeof(double),
       cudaMemcpyDeviceToHost));


        //**********************CONTROLLO
       CONVERGENZA*****************************
        for (int c = 0; c < numCentroidi; ++c) {
            double shift = 0.0;
            convergenza = false;

            for (int d = 0; d < dimensioni; ++d) {
                double diff = h_centroidiCorrenti[c * dimensioni + d] -
       h_centroidiPrecedenti[c * dimensioni + d];
                shift += diff * diff;
            }

            if ((std::sqrt(shift) / dimensioni) > tol) {
                //printf("Non convergente\n");
                break;
            }
            else {
                convergenza = true;
            }
        }


        iter++;

        // Aggiorno i vecchi centroidi per confronto successivo
        h_centroidiPrecedenti = h_centroidiCorrenti;



    }

    // Stampa il numero totale di iterazioni eseguite
    //std::cout << "Numero di iterazioni per convergenza: " << iter << "
       \n";

    CUDA_CHECK(cudaFree(d_grandezzeCluster));
    CUDA_CHECK(cudaFree(d_nuoviCentroidi));
}
```

3.6. Kernels

The kernel functions are defined using the __global__ keyword (runs on the GPU but is called from the CPU). Inside the kernel, each thread determines its unique ID based on its block and thread index, allowing it to process a specific subset of data.

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

Listing 14. Thread Indexing

3.6.1 Cluster Assignment Kernel

The kernel `assegnaClusters` computes the Euclidean distance between each data point and all centroids, assigning the closest centroid to each point.

```
__global__ void assegnaClusters(const double* punti, const double*
       centroidi, int* assegnamenti,
    int numPunti, int numCentroidi, int dimensioni) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= numPunti) return;

    double minDist = INFINITY;
    int bestCluster = -1;

    for (int c = 0; c < numCentroidi; ++c) {
        double dist = 0.0;
        for (int d = 0; d < dimensioni; ++d) {
            double diff = punti[idx * dimensioni + d] - centroidi[c *
       dimensioni + d];
            dist += diff * diff;
        }
        dist = sqrt(dist);
        if (dist < minDist) {
            minDist = dist;
            bestCluster = c;
        }
    }
    assegnamenti[idx] = bestCluster;
}
```

Listing 15. Cluster Assignments

3.6.2 Updating Centroids Kernel

In kernel `aggiornaCentroidi` each thread accumulates the sum of points assigned to a cluster using atomicAdd to prevent race conditions.

```
__global__ void aggiornaCentroidi(const double* punti, double*
       nuoviCentroidi, int* assegnamenti,
    int numPunti, int numCentroidi, int dimensioni, int*
       grandezzeCluster) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= numPunti) return;

    int cluster = assegnamenti[idx];
    if (cluster == -1) return;

    for (int d = 0; d < dimensioni; ++d) {
        atomicAdd(&nuoviCentroidi[cluster * dimensioni + d], punti[idx *
       dimensioni + d]);
    }
    atomicAdd(&grandezzeCluster[cluster], 1);
}
```

Listing 16. Thread Indexing

3.6.3 Normalizing Centroids Kernel

This kernel `normalizzaCentroidi` calculates the new centroid positions by averaging the assigned points.

```
__global__ void normalizzaCentroidi(double* centroidi, double*
       nuoviCentroidi, int* grandezzeCluster, int numCentroidi, int
       dimensioni) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= numCentroidi) return;

    int n = grandezzeCluster[idx];
    if (n > 0) {
        for (int d = 0; d < dimensioni; ++d) {
            centroidi[idx * dimensioni + d] = nuoviCentroidi[idx *
       dimensioni + d] / n;
        }
    }
}
```

Listing 17. Thread Indexing

4. Tests and Results

The evaluation of the K-Means algorithm parallel versions using CUDA, was conducted on NVIDIA rtx 4070 laptop. Initial centroids were randomly selected and identical for both versions to ensure a fair comparison.

4.1. Datasets

The tests were performed on datasets generated both randomly and synthetically, as well as on real-world datasets.

1. The random datasets were generated by varying the number of points and dimensions, while the algorithm

was tested with varying numbers of clusters (10, 100, and 1000).

2. The real-world dataset used was letters. This dataset represents the 26 letters of the English alphabet.

3. synthetic datasets included a1, a2, a3, birch1, and s1.

   (a) a1, a2, a3: Synthetic 2-d data with increasing number of clusters (k). There are 150 vectors per cluster.

   (b) birch1: Synthetic 2-d data with N=100,000 vectors and k=100 clusters in regular grid structure.

   (c) s1: s-set contains synthetic 2-d data with N=5000 vectors and k=15 Gaussian clusters with different degree of cluster overlap. s1 contains well-separated clusters with uniform density.

4.2. Results

The execution time and speed-up relative to the sequential version were measured.

Dataset	Dimensions	k	Speedup
s1set	5000×2	15	29,4828
a1	150×2	20	25,3333
a2	150×2	35	54,8780
a3	150×2	50	82,6987
letters	20000×16	26	49,5579
birch1	100000×2	100	273,8437

Table 1. Synthetic and real datasets speedup

File	Tempo Seq (s)	Tempo Parallelo (s)
s1set.txt	0.085479	0.002926
a1.txt	0.091160	0.003609
a2.txt	0.224975	0.004115
a3.txt	0.603714	0.007272
letter.txt	11.388423	0.229793
birch1.txt	95.461874	0.348642

Figure 1. Synthetic and real datasets execution time

Datasets:	1000x10			1000x100			10000x10			10000x100			100000x10			100000x100		
k	Seq (s)	Par (s)	Speedup	Seq (s)	Par (s)	Speedup	Seq (s)	Par (s)	Speedup	Seq (s)	Par (s)	Speedup	Seq (s)	Par (s)	Speedup	Seq (s)	Par (s)	Speedup
10	0,0745	0,0066	11,2879	0,1659	0,0081	20,4815	0,6237	0,0060	103,95	0,7656	0,0481	15,9168	5,4037	0,1047	51,6113	7,6078	0,1894	40,1679
100	0,3130	0,0139	22,5180	1,5275	0,0963	15,8619	11,1820	0,3096	36,1176	24,4609	0,3191	76,6560	174,1373	0,3995	435,8881	153,9778	0,3783	407,0256
1000	0,2448	0,0180	13,6000	2,3278	0,2826	8,2371	34,2272	0,3844	89,0406	379,2973	1,0595	357,9965	-	-	-	-	-	-

Table. random generated datasets results

The results show a significant acceleration when using the GPU:

1. Even for smaller datasets with a low value of k that were tested (s1set, a1), the speedup is in the range of 25x to 30x, demonstrating the benefits of parallelizing distance computations and centroid updates. Additionally, for very small datasets in terms of points and dimentions, such as a1, a2, and a3, an experiment with blocksize=128 showed a slight improvement in speedup. This suggests that reducing the blocksize for small datasets can improve GPU efficiency.

Dataset	Dimensions	k	Speedup
a1	150×2	20	26,0533
a2	150×2	35	57,4053
a3	150×2	50	83,0288

Table 2. small datasets speedup with blocksize=128

File	Tempo Seq (s)	Tempo Parallelo (s)
a1.txt	0.108356	0.004159
a2.txt	0.292193	0.005090
a3.txt	0.639073	0.007697

Figure 2. small datasets tests with blocksize=128

However, for larger datasets, blocksize=256 was selected as it showed better results.

2. As the dataset size increases, the speedup improves. For example, birch1 dataset ($100{,}000 \times 2$, k = 100) achieves 273x speedup, highlighting the efficiency of CUDA in handling large-scale clustering tasks.

3. High-dimensional datasets, such as letters ($20{,}000 \times 26$), also benefit from GPU acceleration, achieving a speedup of approximately 49.56x