k-Means implementation using OpenMP and CUDA

Niccolò Mazzi - 7076194

February 7, 2023

Abstract

In this report we'll expose the implementation of the k-means algorithm using OpenMP and CUDA and the speedups obtained by executing these implementations with different settings.

1 Introduction

K-means is a clustering algorithm which identifies k *clusters* inside a set of objects. The number k of clusters will be chosen before the execution of the algorithm. For each clusters we'll identify a *centroid* (so we'll have k centroids) which represents the cluster's center.

2 How does k-means algorithm work?

K-means is an iterative algorithm which can be synthesized in the following steps:

1. **Initialization**: we define **K** and the set of objects we'll work on.
2. **Assignment** Each data point gets assigned to the nearest cluster based on the Euclidean distance to the centroid.
3. **Updating the centroid position**: we recalculate the exact position of the centroid based on the "*center of mass*" concept.

A point p_i can only belong to a cluster C_n and **all** the points will be assigned to a cluster.

This algorithm ensures the convergence in a finite time.

3 OpenMP implementation

In this implementation it will be possible to choose the number of cores that will be used.

There are many features of the algorithm that are customizable thanks to the definition of some initial parameters:

```
//Definizione delle variabili iniziali:

#define NUM_THREADS 6
#define NUM_CLUSTERS 10
#define NUM_POINTS 10000
#define ITER 20
```

3.1 Points and clusters creation

Points are implemented as a "**Point**" object composed by the following elements:

- **x**, **y** are the coordinates in the 2D-space
- "**cluster**" is the cluster which the point belongs

The distance between a point and a cluster is given by the euclidean distance between that point and the cluster's centroid. Centroids are implemented as "Points" data type too. We calculate the distance by using a function called `distance()` in which we return the distance calculated in the following way:

```
    sqrt((a.x - b.x) * (a.x - b.x) + (a.y -
↪   b.y) * (a.y - b.y))
```

3.2 Assignment of points to the clusters

To assign each point to a cluster we use the following loop:

```
#pragma omp for
    for (int i = 0; i < NUM_POINTS; i++) {
        double cluster = 0;
        double min_distance =
↪   distance(data[i], centroids[0]);
        for (int j = 1; j < NUM_CLUSTERS;
↪   j++) {
            double dist = distance(data[i],
↪   centroids[j]);
            if (dist < min_distance) {
                min_distance = dist;
                cluster = j;
            }
        }
        data[i].cluster = cluster;
    }
```

As we can see, we use the `#pragma omp for` directive. This is explained by the fact that this loops gets executed in a parallel way.
The **omp** directive allows access of a specific memory location atomically. It ensures that race conditions are avoided through direct control of concurrent threads that might read or write to or from the particular memory location

3.3 Update centroid position

A cluster has to recalculate the centroid position. Concretely, this operation is executed by the following loop:

```
#pragma omp for
            for (int i = 0; i <
↪   NUM_CLUSTERS; i++) {
                int count = 0;
                Point sum = { 0, 0 };
                for (int j = 0; j <
↪   NUM_POINTS; j++) {
                    if (data[j].cluster ==
↪   i) {
                        sum.x += data[j].x;
                        sum.y += data[j].y;
                        count++;
                    }
                }
                if (count > 0) {
                    centroids[i].x = sum.x
↪   / count;
                    centroids[i].y = sum.y
↪   / count;
                }
            }
        }
```

where the effective update is performed trough this operation:

```
    centroids[i].x = sum.x / count;
    centroids[i].y = sum.y / count;
```

3.4 Algorithm execution steps

At the beginning of the algorithm we generate **NUM_POINTS** points and **NUM_CLUSTERS** centroids.

After that we execute the following loop (parallelized with **NUM_OF_THREADS** threads) for at the most **ITER** times:

- We assign all the points to the respective clusters (if NUM_OF_THREADS > 1 this operation will be parallelized)
- Recalculate the clusters'centroids

In conclusion, the operations that have been parallelized in the implementation are the following ones:

- Points assignment(*assignPointsToClusters()*)
- Recalculate clusters'centroid (*recomputeCentroid()*)

3.5 Execution tests and speedups

Using a single core (sequential execution) we obtain the following results:

N. points	N. clusters	time (ms)
100.000	30	1.55
100.000	50	2.48
100.000	100	4.88
1.000.000	30	15.47
1.000.000	50	25.37
1.000.000	100	49.80

By using 6 cores we obtain the following execution times:

N. points	N. clusters	time (ms)	speedup
100.000	30	0.35	4.42
100.000	50	0.58	4.27
100.000	100	1.11	4.39
1.000.000	30	3.36	4.60
1.000.000	50	5.32	4.77
1.000.000	100	10.32	4.82

4 CUDA implementation

In the cuda implementation to represent points and clusters we used linearized matrices. In other words, a linearized matrix is a matrix with m rows and k columns where:

$$M[i][j] = m[i * k + j]$$

This will allow us to manipulate them in a easier way with CUDA.

4.1 Points assignment to cluster

To assign the points to the clusters we follow these steps:

- I map each point p in a CUDA thread
- Each point p looks for his best-fitting cluster
- We proceed by associating p and C

The operation illustrated above is implemented in the following function:

```
    __global__ void assign_clusters(float*
↪   punti, float* clusters) {
        long id_punto = threadIdx.x +
↪   blockIdx.x *
↪   blockDim.x;
        if (id_punto < POINT_NUM)
↪   {
                float x_punto, x_cluster,
↪   y_punto, y_cluster = 0;
                x_punto = punti[id_punto *
↪   POINT_FEATURES + 0];
                y_punto = punti[id_punto *
↪   POINT_FEATURES + 1];
                long best_fit = 0;
                long distMax = LONG_MAX;
                for (int i = 0; i <
↪   CLUSTER_NUM; i++) {
                        int
↪   cluster_index_point = clusters[i *
↪   CLUSTER_FEATURES + 0];
                        x_cluster =
↪   punti[cluster_index_point *
↪   POINT_FEATURES + 0];
                        y_cluster =
↪   punti[cluster_index_point *
↪   POINT_FEATURES + 1];
                        if
↪   (distance(x_punto, x_cluster, y_punto,
↪   y_cluster) < distMax) {
                                best_fit =
↪   i;
                                distMax =
↪   distance(x_punto, x_cluster, y_punto,
↪   y_cluster);
                        }
                }
                //Output, i assign the
↪   results:
                punti[id_punto *
↪   POINT_FEATURES + 2] = best_fit;
        //CRITICAL SECTION
                atomicAdd(&clusters[best_fit
↪   * CLUSTER_FEATURES + 1],
↪   x_punto);
```

```
                atomicAdd(&clusters[best_fit
↪  * CLUSTER_FEATURES + 2],
↪  y_punto);
                atomicAdd(&clusters[best_fit
↪  * CLUSTER_FEATURES + 3],
↪  1);
        }
```

4.2 centroid recalculation

The centroid recalculation is performed trough CUDA.
To perform this operation we'll use a number of threads that will be equal to the number of clusters. Each thread will recalculate his cluster'centroid through the kernel

4.3 Algorithm Execution

To set the variables of the program we can modify the following parameters:

```
#define CLUSTER_NUM 30
#define POINT_NUM 1000000
#define THREADS_BLOCK 512
#define IT_MAX 20
#define EPSILON 0.001
```

The logic of the program is very similar to the OpenMP one, but in this version the assignment of the points to the clusters, the centroids recalculation and the clusters reset are performed through CUDA. Cause of this, after the points and the clusters are created, they will be moved from the *host* memory to the *device* memory.

4.4 Usage of THREAD_BLOCK variable

The **Thread_BLOCK** variable is used to perform the division of threads and procks for the *assign_cluster* operation.

The blocks allocated for this operation will be calculated as

$$\frac{POINT\_NUM + THREAD\_BLOCK - 1}{THREAD\_BLOCK}$$

and each one of these blocks will contain **THREAD_BLOCK** threads

4.5 Execution test and speedups

Running the CUDA version with a GTX 1070 GPU we obtain the following results:

N. points	N. clusters	time (ms)	speedup
100.000	30	0.022	70.45
100.000	50	0.034	72.94
100.000	100	0.059	82.71
1.000.000	30	0.146	105.96
1.000.000	50	0.263	96.46
1.000.000	100	0.44	113.18
10.000.000	30	1.321	115.47
10.000.000	50	2.144	115.37
10.000.000	100	3.83	127.18

5 Final Considerations

In this project we implemented both cuda and OpenMP versions of Kmeans algorithm.
As result we obtained that the CUDA version permorms a valuable speedup in comparson both to OpenMP and sequential version.