Implementazione e risultati del clustering k-means in modalità sequenziale e parallela con CUDA

Alessandro Lemmo
E-mail address
alessandro.lemmo@stud.unifi.it

Abstract

In the following work, k-means clustering was studied and implemented in two ways, one sequential in C ++ and one parallel with CUDA. The latter, acronym for Compute Unified Device Architecture, is a platform for parallel processing, as well as an API, created by NVIDIA.

The developed application is unique. Clustering is performed on the same points taken as input in two ways: sequential and parallel. The performances of the two executions are then evaluated and compared on the basis of different parameters such as the number of threads, the number of points taken as input and the number of clusters.

1. Introduction

k-means clustering is a method of vector quantization that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells.

The software created takes n points in three dimensions as input.k-means clustering is performed on this data in order to obtain k clusters. These clusters will have the centroids in exactly the same points both as regards sequential and parallel execution, what changes are the performance in the calculation. Both executions are implemented within the same source file. As for the sequential application, it was created in C++ while the parallel application was created in CUDA (Compute Unified Device Architecture).

CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. The programming languages available in the CUDA development environment are extensions of the most common languages for writing programs. The main one is 'CUDA-C' (C with Nvidia extensions), which has also been used for the development of this project.

2. Description

On a theoretical level, clustering k-means functions in the following way: 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 ($\leq n$) sets $S = \{S_1, S_2, ..., S_k\}$ so as to minimize the within-cluster distance. Formally, the objective is to find:

$$arg \min_{S} \sum_{i=1}^{k} \sum_{x \in S_i} d(x, u_i)$$

where u_i is the mean of points in S_i and d is the euclidean distance. In practice, the arrangement of the clusters and therefore of their centroids must be found so as to minimize the dis-

tance between each point and the centroid of the cluster to which it belongs. However, the problem remains of how to assign a point to a certain cluster. We will see it in the next paragraph.

3. Algorithm

Given an initial set of k centroids $m_1^{(1)}, ..., m_k^{(1)}$, where the subscript indicates the cluster index and the superscript indicates the iteration, the algorithm proceeds by alternating between two steps:

Assignment step: Assign each observation to the cluster with the nearest centroid: that with the Euclidean distance.

$$S_i^{(t)} = \left\{ x_p : d\left(x_p, m_i^{(t)}\right) \leq d\left(x_p, m_j^{(t)}\right) \ \forall j,\ 1 \leq j \leq k \right\}$$

where each x_p is assigned to exactly one $S^{(t)}$.

Update step: Recalculate centroids for observations assigned to each cluster.

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j$$

The algorithm has converged when the assignments no longer change. The algorithm does not guarantee to find the optimum. The iterative procedure is illustrated in figure 1 at the follow page.

4. Implementation

Now let's see how the implementation of both the sequential and the parallel version is carried out.

4.1. Sequential version

The implemented algorithm can be seen as divided into three main phases: initialization, assignment and updating. Let's see the three phases in detail.

Inizialization: a vector is defined containing for each point to be classified the index of the belonging cluster. A vector of results is then defined which must contain the centroids of the clusters obtained. A centroid is represented by means of its coordinates through a struct.

```
for i = 0 to total num of points {
    clusters[i] = -1
}

for i = 0 to total num of clusters  {
    result[i] = points[i]
    clusters[i] = i
}
```

The result vector, as reported in the code, is initialized by placing the centroids of the k clusters in correspondence of the first k points.

Assignment: a "clusterCounts" vector is defined which will contain for each cluster the number of points belonging to it. A "delta" variable is then defined which at each iteration contains the number of points to which the belonging cluster has been changed. When it turns out nothing ends.

```
for i = 0 to total num of clusters
    clusterCounts[i] = 0

delta = 0

//assignment of each point at one cluster
//for each point it scrolls all the centroids
//and determines the closest one
for i = 0 to total num of points
{
    bestCluster = -1

    for j = 0 to total num of clusters
    {
        dist =
        EuclideanDistance (points[i], result[j])

        if (dist < minDistance)
        {
            minDistance = dist
            bestCluster = j
        }
    }

    //if the bestCluster is not the actual
    //cluster of appartenence
    if (bestCluster != clusters[i])
    {
        clusters[i] = bestCluster
        delta++
    }
    clusterCounts[bestCluster]++
}
```

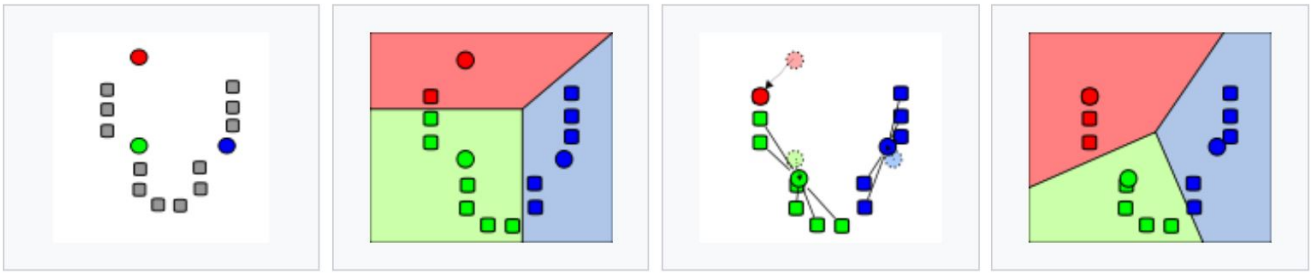

Figure 1:
1. k initial centroids (in this case k=3) are randomly generated within the data domain (shown in color).
2. k clusters are created by associating every observation with the nearest mean. The partitions here represent the Voronoi diagram generated by the means.
3. The centroid of each of the k clusters becomes the new centroids.
4. Steps 2 and 3 are repeated until convergence has been reached.

At the end we will have the vector ”clusters” which will contain for each point the corresponding index of the belonging cluster and the vector ”clusterCounts” which will contain for each cluster the number of points that belong to it.

Update: we define a vector containing the new centroids. Then the sum is made component by component of all the points belonging to the same cluster and all three components are divided by the total number of points belonging to the cluster. In this way the new centroid of the cluster was calculated. The same procedure is done for all clusters.

```
Point* newResult = new Point[k];

for i = 0 to total num of points
{
    //cluster belonging to the i-th point
    index_cluster = clusters[i]

    //sum component for component
    newResult[index_cluster] =
        newResult[cluster] + points[i]
}

for (int i = 0; i < k; i++)
{
    if (clusterCounts[i] == 0)
        continue

    //calculate the new centroid of the i-th
    //cluster
    newResult[i] =
        newResult[i] / clusterCounts[i]
}
result = newResult
```

4.2. CUDA version

The algorithm developed with CUDA has an operating logic almost identical to that developed in sequential mode. The implementation also in this case can be seen as divided into three main phases: initialization, assignment and updating. The initialization phase is the same as the sequential one. Is defined a vector of points to contain the centroids results, initialized with the k centroids coinciding with the first k points, and a vector ”clusters” of integers which will contain for each point the index of the belonging cluster. Instead, the assignment and updating phases, while maintaining the same operating logic, are implemented in parallel through the use of two kernels.

Assignment: below is reported the code of the first kernel function.

```
__global__ findCluster(parameters)
{
    i = blockIdx.x * blockDim.x + threadIdx.x

    //assignment of i-th point at j-th cluster
    for (pointNr=0; pointNr < POINTS_FOR_THREAD
         pointNr++)
    {
        index = i * POINTS_FOR_THREAD + pointNr

        Point p = d_points[index]

        for j = 0 to total num of clusters
        {
            dist = PointDistance(p, d_result[j])
```

```
            if (dist < minDist)
            {
                minDist = dist
                bestCluster = j
            }
        }

        if (bestCluster != d_clusters[index])
        {
            d_clusters[index] = bestCluster
        }
        d_counts[bestCluster * threads + i]++

        Point old_result =
        d_new_result[bestCluster * threads + i]

        d_new_result[bestCluster * threads + i]=
        Point(old_result.X + p.X,
              old_result.Y + p.Y,
              old_result.Z + p.Z)
    }
}
```

The outermost for loop does not perform as many iterations as there are total points but how many points are assigned to a thread. While in the sequential case an i index could be used to refer to the i-th point, in this case we define "index" which considers both the iteration of the loop and the thread that is being executed. The internal for loop, the update of the vector containing the indexes of the best clusters relating to each point and the update of the vector containing for each cluster the number of points assigned remain identical to the sequential case. All the operations indicated in the code are in this kernel because it requires a parallel cycle on all points managed by a number n of threads calculated on the basis of the number of total points and this is already present in this kernel. It was executed with n threads and a certain number of blocks calculates on the basis of number of points and threads.

Update: First, through the use of the thrust library, which allows you to perform addition operations in parallel, the coordinates of the points belonging to the same cluster have been added together, component by component, in order to obtain a vector of points containing as many elements as there are clusters. In a kernel function the point generated by the sum of components is divided by the total number of all points belonging to the same cluster in way to obtain the new updated centroid. This operation is handled in a kernel different from the previous one in that it is working on the number k of clusters and no longer on the number n of points. In fact, this kernel is executed on k threads and a certain number of bloks calculates on the basis of the number of points and threads.

```
__global__ void calculateResult(parameters)
{
    i = blockIdx.x * blockDim.x + threadIdx.x

    Point p = d_new_result[i]
    count = d_counts[i]

    d_result[i] =
        Point(p.X/count, p.Y/count, p.Z/count)

}
```

5. Results

Both the sequential and the parallel versions have been tested to evaluate their performance by varying the number of points, number of clusters and number of threads. In order to have the most accurate tests possible, the points on which to cluster have been stored in a text file. This is to ensure that the same points were always passed at each execution (and not those randomly generated by the program). As for the only variation in the number of threads used, a minimum of two to a maximum of 1024 were used. The results are shown below. It can be seen that the parallel version is always faster than the sequential version and that it has a significant improvement up to 32 threads. From 32 to 128 it still improves, albeit with very small margins, after which the execution times remain approximately constant. This is most likely due to the procedure of creating a very large number of threads, which is very expensive.

Thread number	Sequential (msec)	CUDA (msec)
2	10169	9539,5
4	10145	5403,5
8	10105	3195.2

continued on next page

continued from previous page		
Thread number	Sequential (msec)	CUDA (msec)
16	10214	2063.8
32	10182	1510.2
64	10163	1490.2
128	10107	1483.4
256	10157	1484
512	10132	1494.3
1024	10069	1520.8

Table 1: Results in milliseconds with sequential and CUDA application to the variation of the number of threads

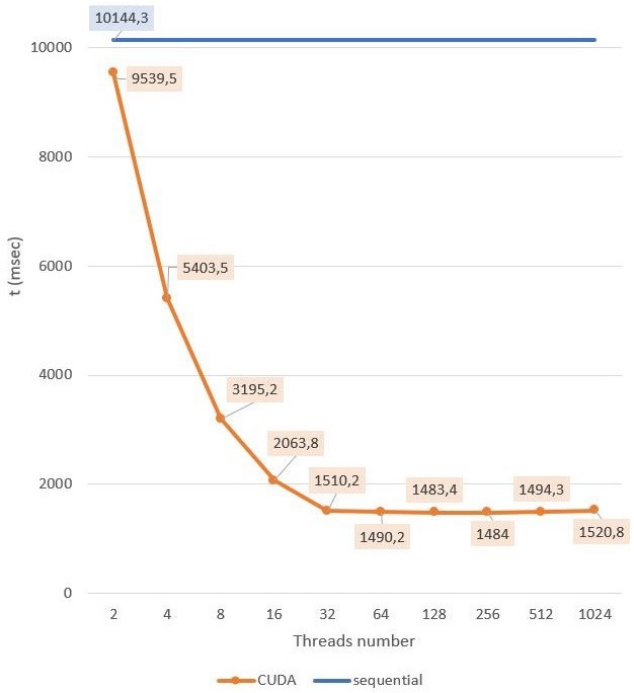

Figure 2: Time difference in execution between sequential and CUDA to the variation of the number of threads

By evaluating the variation of the number of clusters from a minimum of 5 to a maximum of 30, more fluctuating results are obtained. This is due to the fact that the increase in the number of clusters does not necessarily lead to an increase in the number of iterations and therefore to an increase in the execution time. It may happen that more clusters result in fewer iterations. Even in the latter case, the execution time may still slightly increase as having a greater number of clusters the single iteration becomes more expensive. The results are shown below.

Clusters number	Sequential (msec)	CUDA (msec)
5	6852.7	977.6
10	10144.3	1483.4
15	23401.2	2646.7
20	27358.7	3577.1
25	50524.5	6443.8
30	49807.4	6438.2

Table 2: Results in milliseconds with sequential and CUDA application to the variation of the number of clusters

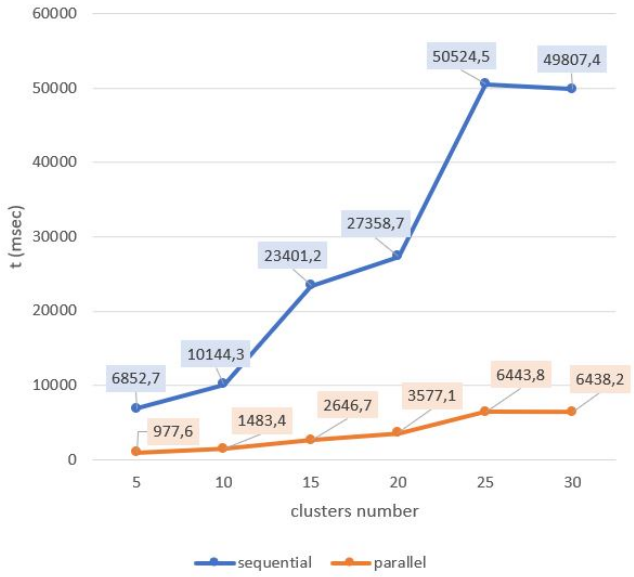

Figure 3: Time difference in execution between sequential and CUDA program to the variation of the number of clusters

Figure 4: Number of iterations to the variation of number of clusters

Finally, the variation in the number of points to be clustered is evaluated. They are taken from a minimum of 100 to a maximum of 1000000 points and compared the times using 10 clusters and 128 threads. Using a fairly high number of threads and initially a low number of points, the parallel version is worse than the sequential one. As the dots increase the use of the parallel version becomes more and more advantageous as evidenced by the intersection of the trajectories in the graph below. The results are shown below.

Points number	Sequential (msec)	CUDA (msec)
100	6.8	346.9
1000	19.7	407.1
10000	30.5	405.3
100000	753.3	797.9
500000	1753.2	802.3
1000000	10144.3	1483.4

Table 3: Results in milliseconds with sequential and CUDA application to the variation of the number of points

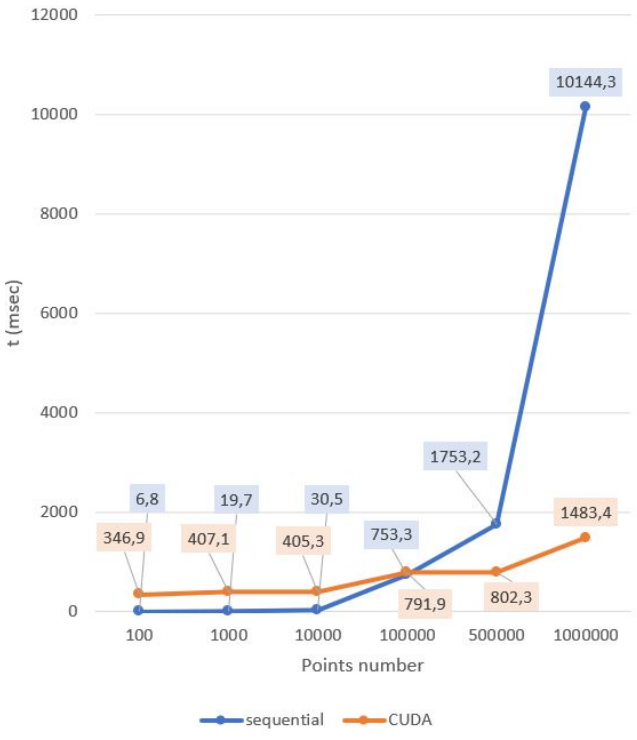

Figure 5: Time difference in execution between sequential and CUDA program to the variation of the number of points

5.1. Speedup

Speedup is a parameter for evaluating the performance of a parallel algorithm defined as

$$S_p = \frac{T_s}{T_p}$$

where T_s is the completation time of the sequential algorithm and T_p is the completation time of the parallel algorithm.

By evaluating the speedup as regards the variation of the number of threads, leaving the number of points and clusters unchanged, the following results are obtained:

Threads number	SpeedUp (msec)
2	1.06
4	1.87
8	3.17
16	4.91
32	6.71
64	6.8
128	6.83
256	6.83
512	6.78
1024	6.67

Table 4: Results in milliseconds of speedup to the variation of the number of threads

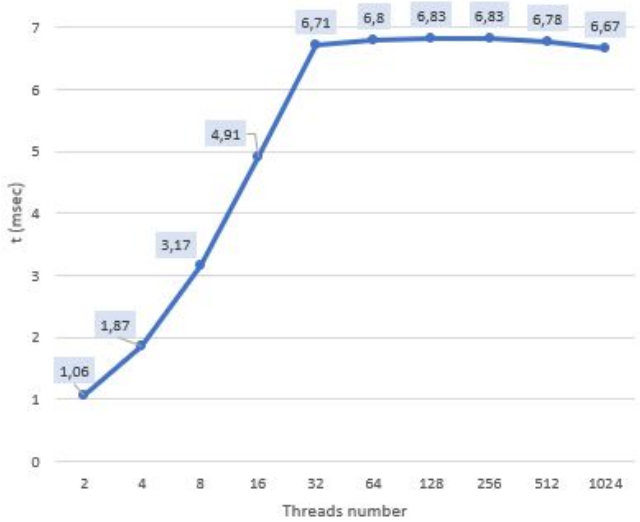

Figure 6: Speedup to the variation of the number of threads

By evaluating the speedup as regards the variation of the number of points and keeping the

number of clusters and the number of threads unchanged, the following results are obtained:

Threads number	SpeedUp (msec)
100	0.01
1000	0.04
10000	0.07
100000	0.95
500000	2.18
1000000	6.83

Table 5: Results in milliseconds of speedup to the variation of the number of points

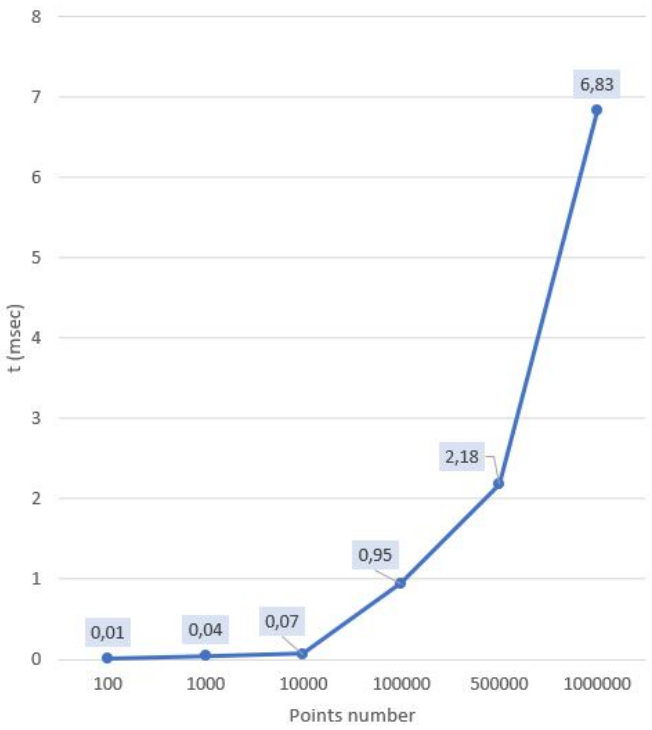

Figure 7: Speedup to the variation of the number of points

6. Conclusion

The presented work allowed to evaluate the performance of a software for the clustering of points in three dimensions which can have applications in various fields. In general we deduce that processing on the GPU is faster than that on the CPU. The cases in which the opposite is true are few and marginal, such as if you have few points and a large number of threads or if you use a parallel version with a very low number of threads. An Nvidia GeForce GTX 1050 Max-Q graphics card was used for processing with CUDA.