K-means Clustering: C++ and Cuda

Alessandro Soci
alessandro.soci@stud.unifi.it

Abstract

The paper presents the K-means clustering method in two different implementation: a sequential implementation in C++ and a parallel implementation using CUDA. It compares the performances between these ways, across images of various size and number of cluster.

1. Introduction

Clustering involves the grouping of similar objects into a set known as cluster. Objects in the same group are more similar (in some sense) to each other than to those in other groups. It is one of the main tasks in exploratory data mining, and it is a common technique for statistical data analysis, too. It is applied in many fields, including machine learning, pattern recognition, image analysis, information retrieval, bio-informatics, data compression, and computer graphics.

In particular, the goal of the project is to study the K-means clustering method that is a vector quantization method. k-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. the final result of the partitioning will be subdivided into space in Voronoi cells.

The problem is computationally NP-hard; however, efficient heuristic algorithms converge quickly to a local optimum.

2. Theoretical formulation

K-means is a iterative method, that run repeatedly some its phase, and it is composed of:

1. Initialization
2. Assignment of the cluster
3. Update of centroid position

2.1. Initialization

Initially, the method choose: the width of the dataset and *K* initial randomly arranged centroid. Choosing the number of centroids, you choose the number clusters of which the dataset will be composed and then the groupings that you want to generate.

2.2. Assignment of the cluster

In this phase, the algorithm analyzes each data points and assigns them to the nearest centroid.

Then the Euclidean distance is calculated between each data points and each centroid. Every data point will then be assigned to the centroid whose has minimum distance from that point.

In mathematical terms:

$$\underset{c_i \in C}{\operatorname{argmin}}\, dist(c_i, x) \tag{1}$$

where c_i is a centroid in the set C (which includes all centroids), x are the data points and dist () is the standard euclidean distance

2.3. Update of centroid position

The average position of each centroid is calculated. The new value of a centroid will be the average of all data points that have been assigned to the new cluster.

In mathematical terms:

$$c_i = \frac{1}{S_i} \sum_{x_i \in S_i} x_i \tag{2}$$

where S_i represents the sum of the data points assigned to the i-th cluster. The new centroid position is obtained from the average of all the data points assigned to the cluster in the previous step.

The algorithm repeat the steps 2 and 3 until it reaches a point of convergence such that there will not be more changes of the centroids.

A drawback of k-means clustering is that it is not guaranteed to always converge to the same clustering [3]. Indeed in this work I use always the same rule of distance[1] and the same initial centroids for every test.

3. Implementation

This paper explain two different ways to perform the K-means method: *sequential* and *parallel*.

From the images, the algorithms create a dataset containing points in 3D. These points are obtained from *RGB channels*, i.e. every pixel corresponds to one point in the 3D space. The domain of the coordinates is $[0 - 255]$ for each axis. Therefore the first part of the algorithms is to manage and create the dataset from an image.

3.1. Sequential K-means

The algorithm implement a *while loop* controlled by the variable *error*, that is the distance between the new values of the centroids and the "old" values of the centroid of the previous round, if it is '0', the loop ends. Within the *while loop* there are two concatenated *for loop* that represent respectively the point [2] and the point [3] of the theory.

Below a pseudo algorithm of the sequential way is shown:

[1] Euclidean distance

Algorithm 1 Sequential K-means

```
1: procedure K-MEANS(R_c,G_c,B_c, centroids, cluster)
2:     while error != 0 do
3:         clusters ← assignment(R_c, G_c, B_c, centroids)
4:         centroids_old ← centroid
5:         centroids ← update(R_c, G_c, B_c, centroids, clusters)
6:         error ← EuclideanDistance(centroids, centroids_old)
```

Algorithm 2 Assignment

```
1: procedure ASSIGNMENT(R_c,G_c,B_c, centroids)
2:     for i=0 to size_image do
3:         for j=0 to size_cluster do
4:             distance ← EuclideanDistance(R_c[i], G_c[i], B_c[i], centroids[j])
5:             if j==0 then
6:                 distance_old ← distance
7:                 cluster[i] ← j
8:             else if distance ≤ distance_old then
9:                 distance_old ← distamce
10:                cluster[i] ← j
```

Algorithm 3 Update Centroids

```
1: procedure UPDATE(R_c,G_c,B_c, centroids, clusters)
2:     for i=0 to size_cluster*3 step 3 do
3:         R ← 0; G ← 0; B ← 0; count ← 0
4:         for j=0 to size_image do
5:             if Clusters[j] == i/3 then
6:                 R ← R + R_c[j]
7:                 G ← G + G_c[j]
8:                 B ← B + B_c[j]
9:                 count ← count + 1
10:        if count != 0 then
11:            centroids[i + 0] ← R/count
12:            centroids[i + 1] ← G/count
13:            centroids[i + 2] ← B/count
```

An simple scheme of sequential algorithm is shown in 1.

3.2. Parallel K-means

The second approach to perform K-means method is CUDA, a parallel computing platform and application programming interface (API) model created by Nvidia. It allows to use a

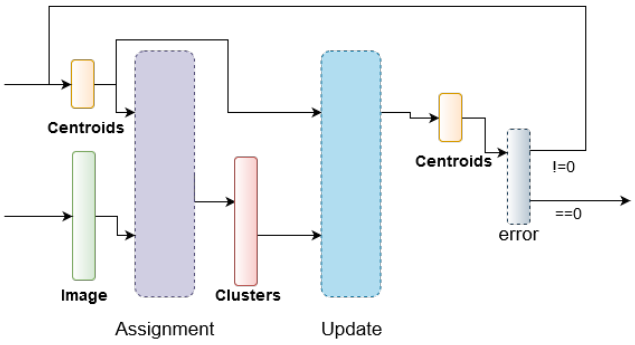

Figure 1. Scheme of sequential method.

CUDA-enabled GPU for general purpose processing. The platform is designed to work with programming languages such as C, C++, and Fortran. This implementation has been written and developed using the C++11 programming language.

The main idea is to divide all possible combination in a greater number of subset, depending on number of threads. So each thread compute a part of all possible iterations, reducing the computational cost considerably.

This work proposes three different parallel implementations: the first is called *shared*, the second *intuitive* and the third *final*.

Shared CUDA

In Cuda is necessary to preallocate every variable with the function *cudaMalloc* and then to copy the variable[2] from the CPU (Host) to GPU (Device) with the function *cudaMemcpy*.

Two global functions are implemented: one for the assignment of the cluster and one for the update of centroid; and one device function to calculate the euclidean distance between 2 points in 3D.

In the first global function every thread compute and update the assignment of the clusters for particular area of image; *e.g.* 4 threads and an image composed by 160000 pixel, every thread makes the assignment for 4000 pixels.

The second global function calls K (number of centroids) blocks and n thread. The reason I use this type of approach is that every block consider only a centroid and because the update function is run one time for *while loop*. The *reduction sum* for Cuda is implemented as describe in [2] to sum every channel of the image. The function use *share* variable and the function *__syncthreads*, so it can erase the problem of race condition.

Every call to the global functions is follow by *cudaThreadSynchronize()* to wait the process of GPU. This slow down the performance but it is necessary.

At the end we calculate the distance between the 'old' and the new centroid, if it is '0' the algorithm will stop.

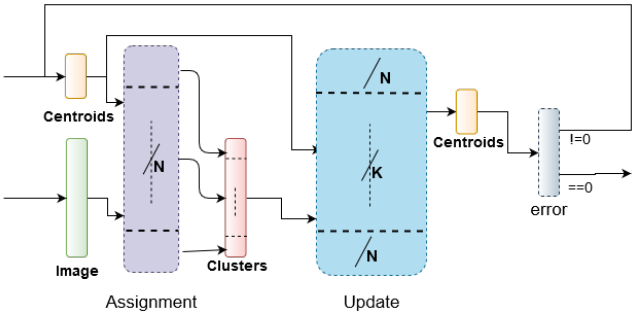

Figure 2. Scheme of shared parallel method.

Intuitive CUDA

Two global functions are implemented like in *shared* version: the first for the assignment of the cluster and the second for updating centroids.

The first function is the same like before, but the second doesn't use shared memory, moreover it calculates only a centroid for time. So it is necessary a *for loop* in the main that call the update function for every centroid. Scheme in figure [3].

Final CUDA

If the number of clusters raises, the speed up between *shared* version and *sequential* version doesn't increase. So *final* parallel version was implemented with the purpose to solve this problem.

The idea of implementation is the same of *shared* parallel version, they have the same scheme [2], but it does not use the shared memory. The low space of shared memory is the biggest

[2] same size of the allocated memory

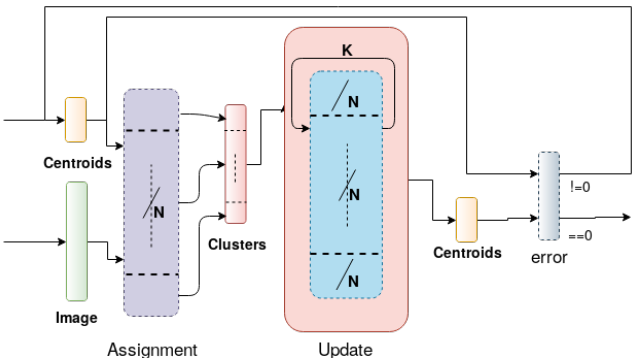

Figure 3. Scheme of intuitive parallel method.

limitation of *shared* version. So the *final* version use a preallocation that depends on the number of cluster and on the number of threads.

4. Results

All the results, in terms of optimal number of threads and execution time, are obtained on a machine with Intel(R) Core(TM) i7-6700 CPU @3.40GHz (total 24 cores) and a Nvidia Titan X GPU.

The dataset is composed by the same image which is resized 7 times, so every resized image has the same distribution of pixels in the RGB space. Also the same initial centroids has been used.

Every single test is run 5-4 times, and the results are the average of all test.

The "comand" *deviceQuery* shows the capacity and the limitation of Titan X. In particular it has a limitation on the shared memory: 49152 bytes. The space is very large but the second global function of *shared* version uses *dynamic shared memory* that depends on the number of clusters and on number of threads. This motivates the *intuitive* and the *final* version.

4.1. Shared version

Below it shown the time of *shared* version when the number of threads changes, with the number of clusters and the size image fixed.

The computational cost decreases quickly until around 100 threads and then decrease slowly, because the dynamic allocation of shared memory takes a lot of time. Higher is the number of threads, four times higher will be the requested space.

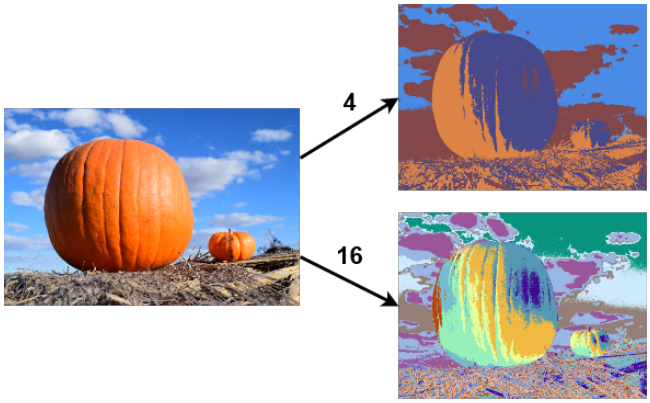

Figure 4. Two images where the pixel belong to the same cluster have the same color: above with 4 clusters; below with 16 clusters.

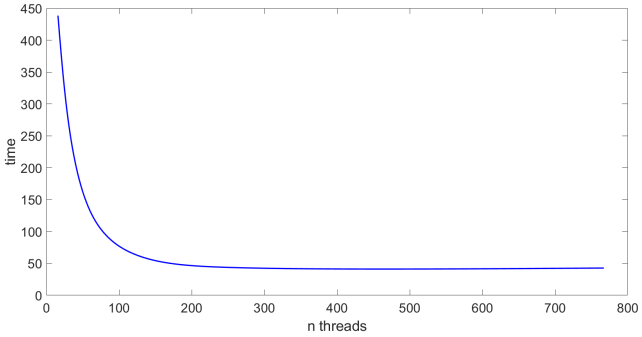

Figure 5. Graphic of the time when number of threads change.

4.2. Shared vs Intuitive

This experiment focus on the difference of speed between *shared* and *intuitive* parallel versions. In the figure below, it is fixed the number of cluster, 4, and the number of threads, 767 (max number of threads for 4 clusters in Titan X).

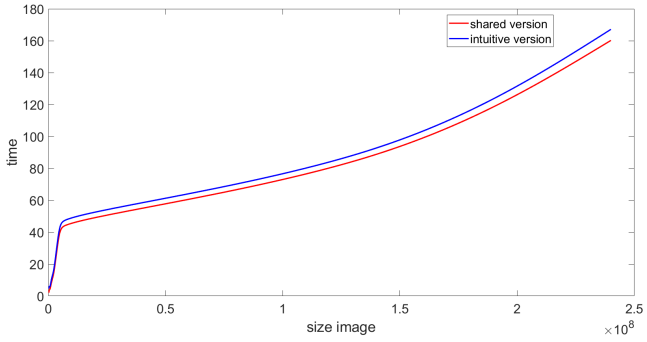

Figure 6. Shared vs Intuitive. Different size image.

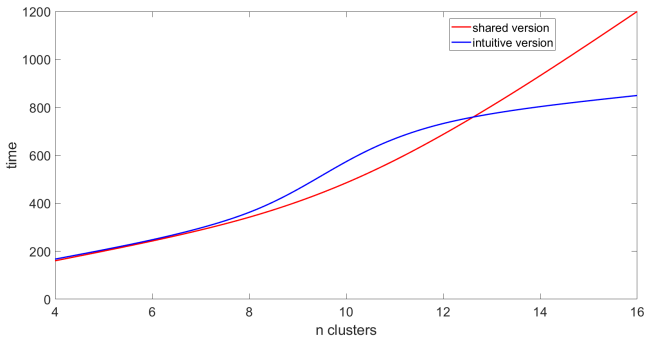

Figure 7. Shared vs Intuitive. Different size image.

In the graphic 7, the number of threads in *intuitive* version is 1024 (max thread per block for Titan X) and the number of threads in *shared* version changes every number of tried clusters due to the limited shared space. The graphic shows how the limitation of *shared* version is really heavy when the number of clusters increase.

4.3. Shared vs Final

To solve the high execution time, I have implemented the *final* version. Below in figure [8] and in [9] it is shown the advantages to not use the dynamic shared memory when it is necessary a lot of memory space.

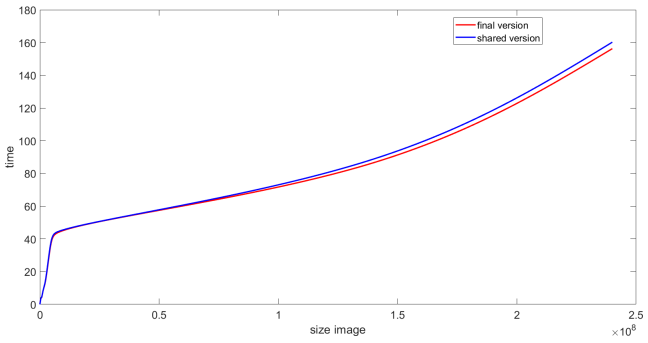

Figure 8. Shared vs Final. Execution time on different size image.

4.4. Sequential vs Final Parallel version

From previous test, the best implementation in parallel is the *final* version. So the next experiment focus on the execution time between *sequential* and *final* parallel version. It is fixed the number of clusters, 4, and the number of threads, 1024 (maximum number of thread per block).

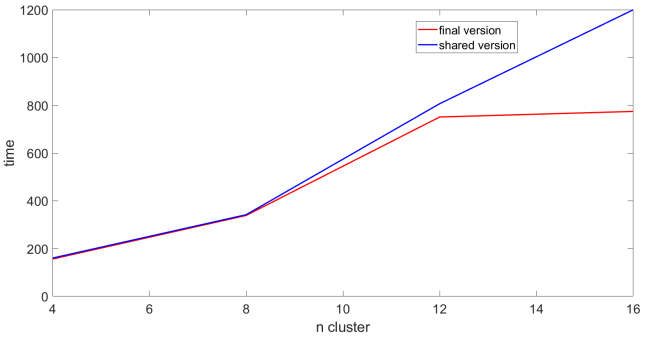

Figure 9. Shared vs Final. Execution time on number of clusters.

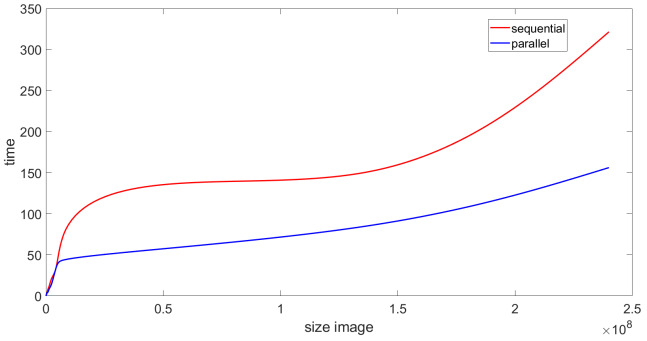

Figure 10. Serial vs Parallel. Execution time on different size image.

As shown in figure number 10, in general the sequential approach is slower compared to the parallel approach. In particular the sequential implementation is just a bit faster than the parallel with small size image, but after around 4500000 pixels the situation is reversed.

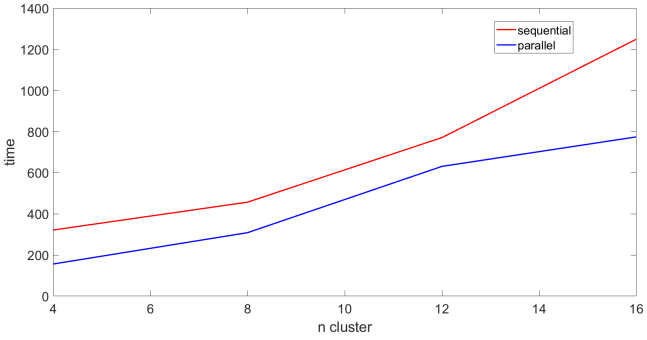

Figure 11. Serial vs Parallel. Different number of clusters.

In figure 11 is shown the difference of speed between different number of clusters on the same image. The size of the image is the highest tested, 24000000 pixels. The parallel version is always better than sequential version. Moreover the se-

quential increase faster than the parallel.

4.5. Speedup

At this stage particular emphasis was given to the speedup concept to figure out if the taken multi-threading approach brought benefits. The speedup S_p is defined as $S_p = \frac{t_s}{t_p}$ where, P is the number of processors, t_s is the completion time of the sequential algorithm and t_p is the completion time of the parallel algorithm. In figure 12 and 13 is shown the speed up obtained with the *final* parallel approach in relation to the sequential algorithm.

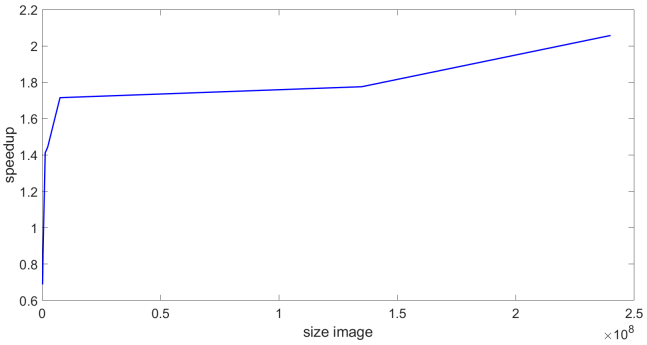

Figure 12. Speedup with different size image and fixed number of thread.

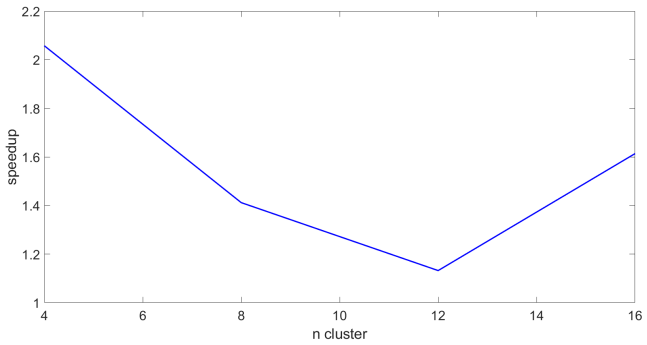

Figure 13. Speedup with different number of threads on the same image.

In figure [13],counter-intuitively, the speed up decreases. Maybe this particularity is due to few iteration of *while* loop.

5. Conclusion

In this paper are described three type CUDA based methods. The *shared* is a bit faster than *intuitive* version with few clusters (from 1 to 12). To solve this problem I have implemented third and *final* version, because there is a relevant limitation about space of shared memory. Indeed the *final* version is better in every test.

However the computational time between sequential and parallel is well shown on the speedup graphics 12 and 13. When the number of data points increases, the speedup can reach value 2 or more; so 2 times faster.

The obtained results are more different than my previous work: it focused on the search of DES-password[1]. The results showed a big speed up between sequential and parallel method. Moreover there was only a call to the kernel. Conversely in this paper, the kernel functions are called 2 times per loop until the convergence, and this particular situation reduces considerably the speed.

References

[1] A. C. Alessandro Soci. Pc-2017/18: Des brute-force attack, 2018. `https://github.com/ciolo/ParallelComputingExam`.

[2] M. Harris. Optimizing parallel reduction in cuda. `Optimizing Parallel REduction in CUDA.pdf`.

[3] P. Pala. K-means clustering. `hierarchical-kmeans.pdf`.