Grenoble INP – ENSIMAG
École Nationale Suprieure d'Informatique et de Mathematiques Appliques

Report

GP-GPU Computing Project
The Parallelized K-Means Clustering Algorithm

BUKHARAEV Alim, MSIAM Data Science track

supervised by PICARD Christophe

27 February 2022

1 Summary

Python is the most popular language for data analysis today. For this reason, it is especially important to learn how to use the GPU power right from the Jupyter Notebook. This report presents a parallelized implementation of the K-Means algorithm with K-Means++ initialization, which shows an order of magnitude better runtime than the well-optimized CPU version from the Scikit-learn library.

2 Background

K-means (Lloyd, 1957; MacQueen, 1967) [3], [4] remains one of the most popular clustering methods. As many other clustering algorithms, it can benefit immensely from the parallel computation. Here is the diagram that describes the algorithm (see fig. 1):

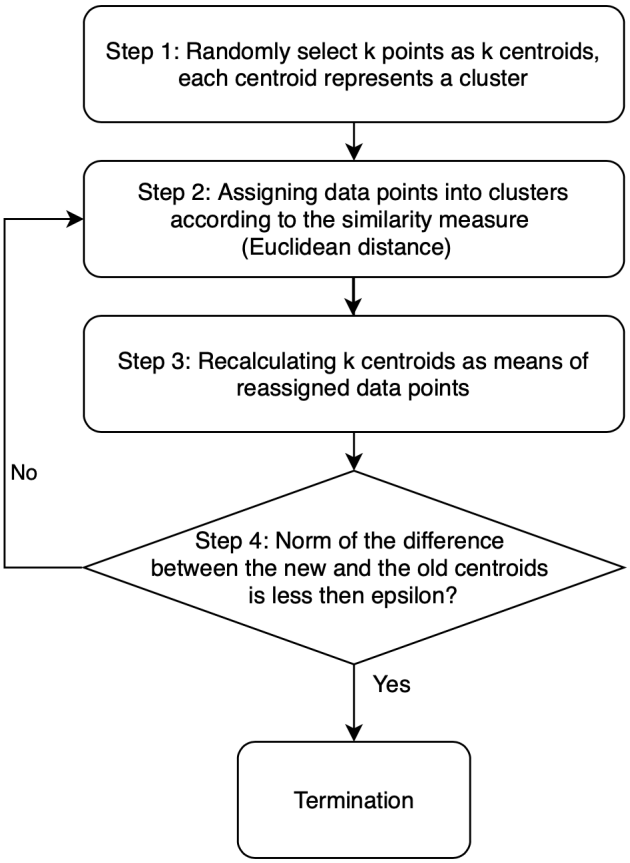

Figure 1: The diagram of the k-means algorithm

Obviously, it is the second and the third steps which need to be parallelized to speed up the algorithm. Computing the distance from each point to each of the cluster centroids and then computing the means of the corresponding points is quite computationally intensive.

Moreover, the performance of k-means algorithm depends heavily on initialization of centroids. One of the ways to find the optimal starting centroids is to use the K-Means++ initialization algorithm [1], which can be described as follows (see fig. 2):

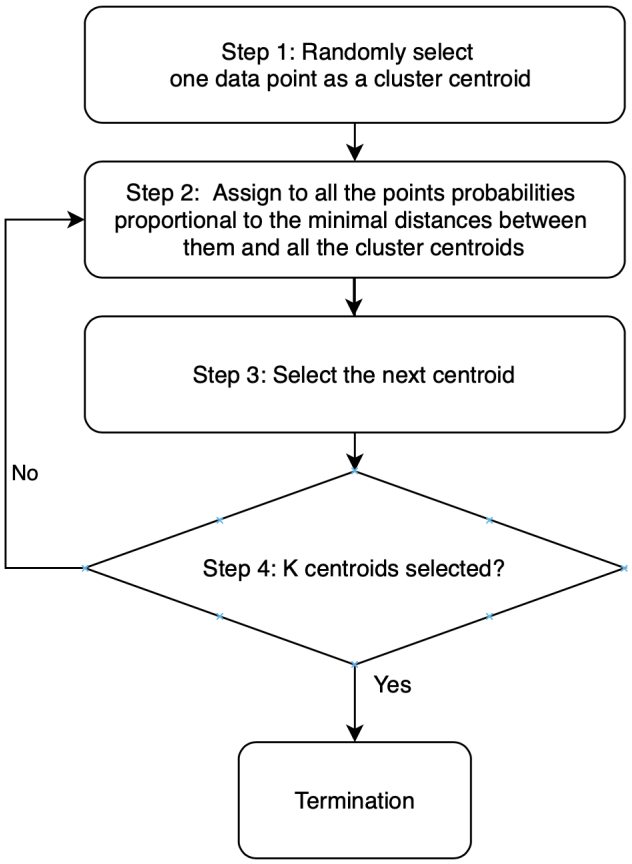

Figure 2: The diagram of the k-means++ initialization

Here, computing distances and choosing the next cluster centroid could be parallelized, too.

3 Challenge

Unfortunately, parallelism is not a magical tool that speeds up any computational task in a number of times. First of all, each algorithm may still have some non-parallelizable operations which would be a bottleneck restricting performance. Secondly, it has to be taken into account that GPU cores are considerably slower then CPUs.

When I just started the project, I thought that parallelizing the distance computation will be enough to achieve a considerable speed-up. However, recomputing cluster centroids turned out to be slow and required parallelization, too (see section 4).

By doing this project, I expected to gain experience in implementing a classic data science algorithm with a GPU, to make it as optimal as possible and to see what would be the gain in performance.

To measure performance, the artificially generated datasets of various sizes (points in 2d to be clustered, which form five clusters) were used. The GPU used was NVIDIA Tesla K80.

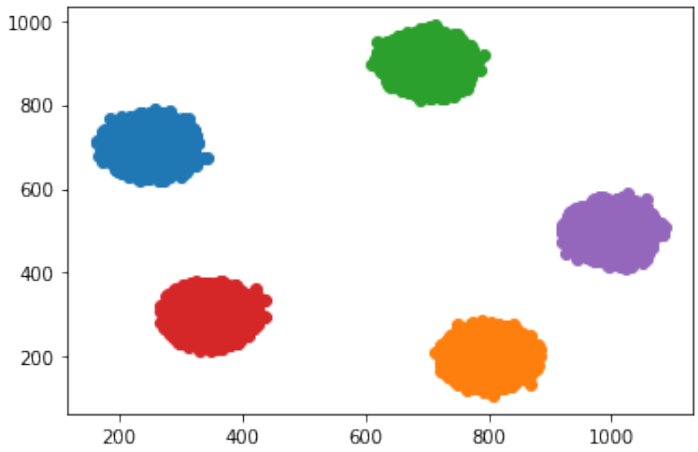

Figure 3: An example of data used

4 Approach

4.1 CPU-implementation

I started the project with implementing K-Means without any use of GPU. A more detailed explanation of its steps would look like this (see 1):

Step 2: For each point find the closest centroid and write the index of the corresponding cluster to a special array.

Step 3: Based on the indexes stored in this array, recompute coordinates of the new k centroids.

As expected, such a naïve implementation turned out to be painfully slow (see section 5).

4.2 In parallel: Step 2

It was quite straightforward to parallelize step 2. Simply put, each of the threads could compute the distance from a single point to all the cluster centroids. Then, it would choose the closest cluster and write its index to global memory.

As expected, this resulted in an immense speed-up in step 2. However, it turned out that step 3 itself was quite computationally intense.

4.3 In parallel: both the steps

So it was decided to parallelize step 3 of the algorithm, too. To compute the new cluster centers as means of corresponding points means to compute the sum of their coordinates and divide it by their quantity. To compute the sum, one can use the reduce algorithm. In this project, the most optimal implementation of the reduce algorithm of the ones learnt in class (see the website [2]) was implemented (the one where memory accessed are coalesced, see fig. 4).

The naïve approach would be to implement the kernels for steps 2 and 3 separately. However, it would be more optimal to implement them in one kernel, so that the information obtained from the global memory during step 2 could be stored in shared memory and was ready to be utilized during step 3. K-Means may require lots of iterations, so this reduction of number of accesses to the global memory is crucial.

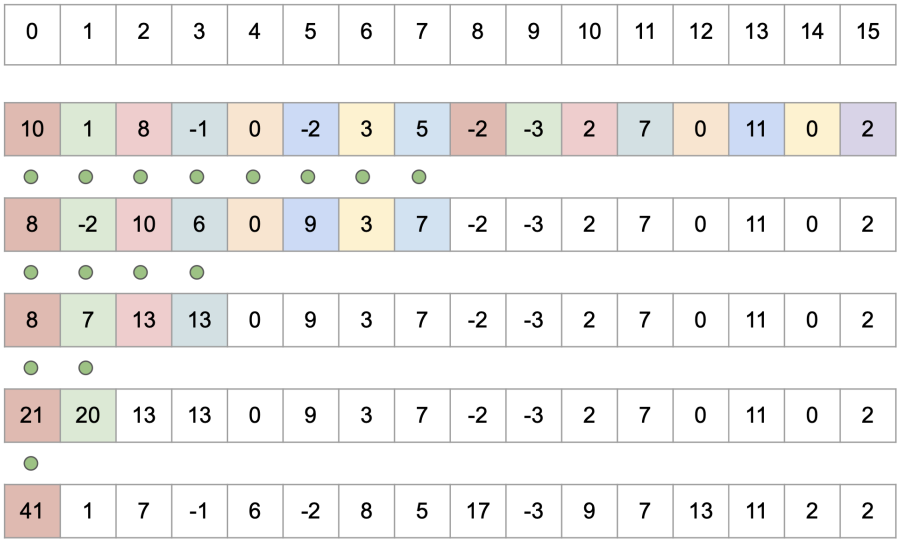

Figure 4: The picture illustrating the k-means algorithm used

4.4 Parallel: K-Means++

For better initialization of the centroids, K-Means++ algorithm was used. Instead of using the probabilistic approach, at each step the point which was the most remote from all the selected centroids was simply picked up as the next cluster center.

Both of the steps (computing the distances from each of the points to the centroids and selecting the most remote point) were implemented as separate kernels (because initialization does not require as many iterations as K-Means itself may).

5 Results

In the three pictures below the results of a naïve CPU implementation (green), the proposed GPU implementation (blue) and the highly optimized Scikit-learn CPU implementation (red) are compared. The dataset was comprised of 5 clusters (see fig. 3), the quantity of each is represented on the x-axis, in logarithmic scale. On the y-axis, time is presented in logarithmic scale, too. The first of the pictures compares time spent on initialization of K-Means++ algorithm (fig, 5). The second picture (fig, 6) compares time spent on average on each iteration of the K-Means algorithm, and the last one (fig, 7) compares the overall time spent by the proposed GPU implementation and the Scikit-learn version.

It can be seen that the proposed GPU implementation outruns the naïve CPU implementation. Also, it is clear that on large datasets (of millions of points and bigger) the proposed implementation surpasses the Scikit-learn version of K-Means, too (running about 10 times faster). This is very important, because waiting dozens of seconds instead of several minutes for clustering to finish could really help.

Strangely, the GPU version of K-Means++ turned out to be a little slower than expected. One of the solutions to that might be merging the kernels of its steps in order to minimize the number of acceses to global memory. Nevertheless, on large datasets the initialization part takes just a very small part of the total running time of the algorithm, so its further optimization would not play a huge role.

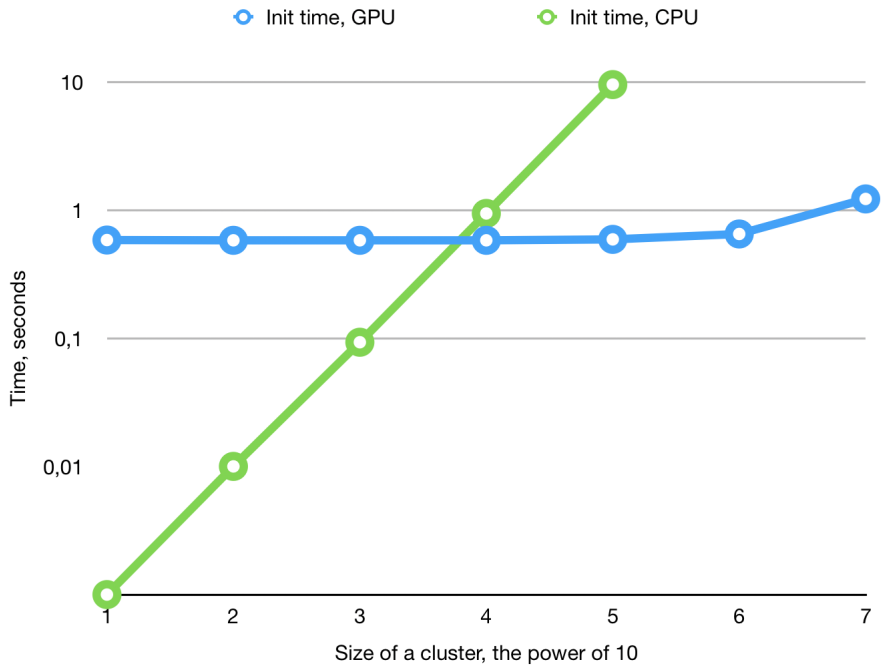

Figure 5: K-Means++ initialization time of GPU and CPU implementations.

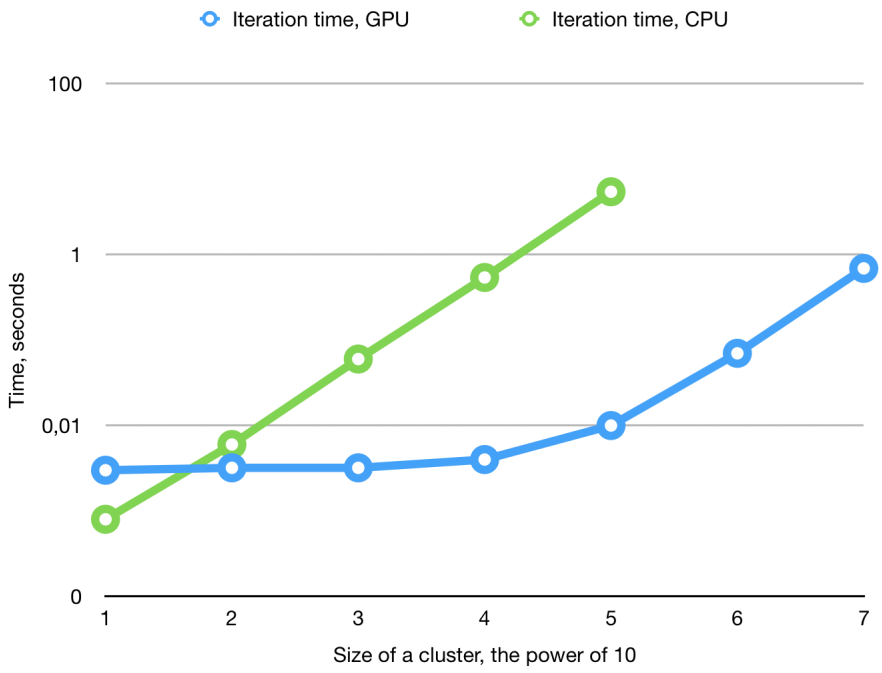

Figure 6: Average iteration time of GPU and CPU implementations.

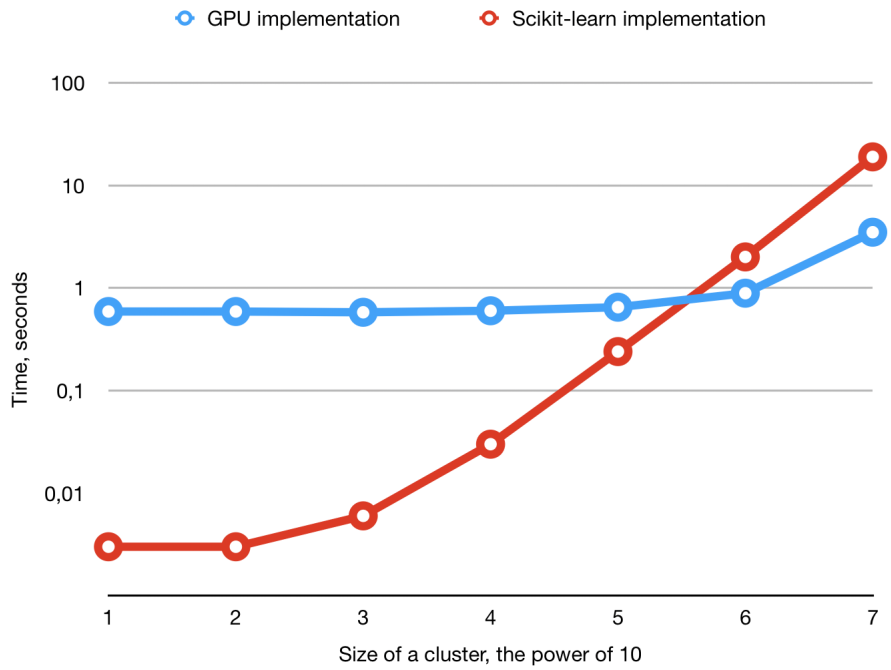

Figure 7: Average times for GPU and Scikit-learn implementations.

It is truly amazing is that with knowledge of CUDA one can run a clustering algorithm right in their Jupyter Notebook 10-100 times faster than the CPU library version. One more good reason to study GPU-Computing.

References

[1] David Arthur and Sergei Vassilvitskii. "k-means++: The Advantages of Careful Seeding". In: *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035. ISBN: 978-0-898716-24-5.

[2] *GP-GPU and High Performances Computing*. `https://christophe.picard.pages.ensimag.fr/courses/course/gp-gpu/`.

[3] S. P. Lloyd. "Least squares quantization in PCM". In: *Technical Report RR-5497, Bell Lab* (1957).

[4] J. B. MacQueen. "Some Methods for Classification and Analysis of MultiVariate Observations". In: *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*. Ed. by L. M. Le Cam and J. Neyman. Vol. 1. University of California Press, 1967, pp. 281–297.