UNIVERSITÀ
DI SIENA
1240

University of Siena

Information Engineering and Mathematical Sciences Department

High Performance Computer Architecture

K-Means implementation in CUDA

A.A. 2021/2022

Brisudova Natasa

Graziuso Natalia

Lazzeri Sean Cesare

Contents

1 Introduction

1.1 K-Means Clustering

K-means clustering is the most commonly used unsupervised machine learning algorithm for partitioning a given data set into a set of K groups (i.e. clusters). It classifies objects in multiple groups, such that objects within the same cluster are as similar as possible. In K-means, each cluster is represented by its center (i.e centroid), which is computed by the arithmetic mean of the coordinates of points inside a cluster [1].

K-means algorithm can be summarized as follows:

1. **clusters initialization** - for each of the cluster K we randomly initialize the centroids;
2. **cluster assignment** - assign each observation to their closest centroid, based on the Euclidean distance between the object and the centroid;
3. **centroid update** - for each of the K clusters update the cluster centroid by calculating the new mean values of all the data points in the cluster;
4. **repeating steps 2 and 3** until convergence or for a given number of epochs.

The sequential code has complexity $O(I * K * N * D)$, where I is the number of iterations, K is the number of clusters, N is the number of data points and D is the dimension of each data point.

Each iteration step has complexity $O(N * K * D)$ due to the reassignment of each data point to the nearest center and therefore it is required to compute the distance between each data point with each cluster center for N data points.

To compute the center for each new cluster after the reassignment it implies to compute K groups of means for N data, and the complexity is $O((N + K) * D)$ for each iteration step.

Since the assignment step is more time-consuming we can use its independence to parallelize it.

1.2 Parallelization using CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).

CUDA uses CPU as its HOST, and GPU as its DEVICE. Each GPU node can get access to thousands of threads, and each thread is processing one single data. The threads are grouped into block and shared memory is restricted to each block. HOST and DEVICE do not share memory [2]. Under this configuration, we would have to manually communicate message between HOST and DEVICE.

Our aim is to parallelize the cluster assignment step.

The logic and order of parallel algorithm is the same with original sequential algorithm, however we have to take into account the communication between HOST and DEVICE in parallel algorithm:

1. HOST initialize cluster centers, copy N data coordinates to DEVICE;
2. DEVICE copy data membership and K cluster centers from HOST;
3. In DEVICE, each thread process a single data point, compute the distance between each cluster center and update data membership;
4. HOST copy the new data membership from DEVICE, and recompute;
5. HOST free the allocated memory.

2 Code implementation

Before working on the project we looked at already existing code for K-means parallelization in CUDA [3] for 1D points, but after trying the code we discovered that it did not work properly. However it still gave us a good overlook on how to implement this algorithm in CUDA and we decided to tackle this problem on our own.

Our aim was to implement K-means algorithm for 2D points (for simplicity arranged in 1D array: $[x_1, y_1, x_2, y_2, ...]$) in CUDA and in C++ and later compare the performances of both.

Both of the codes consist of the major functions being:

- **distance( )** - returning the Euclidean distance of two 2D points;
- **centroid_init( )** - randomly choosing one data point from the dataset for each cluster to initialize the centroids at the beggining;
- **kMeansClusterAssignment( )** - finding the closest centroid to each data point and assigning a cluster to it, using the distance function;
- **kMeansCentroidUpdate( )** - updating the new centroids of each cluster according to the mean value of all the assigned data points in that cluster;

In order to make the code more manageable, we add to this basic structure other functions for reading the data points from file, for saving the results in file of '.csv' format and an user interface for choosing the parameter of the execution, like the number of data points N, number of iterations as well as number of clusters K.

We produced all the data sets by means of the python script in the attachment, according to clusters spawned around a random center.

2.1 CUDA

Our main goal was parallelizing the function for the assignment of each 2D points to one cluster, because of its large time consuming.

We called the kernel:

$$kMeansClusterAssignment <<< (N + TPB - 1)/TPB, TPB >>>$$

by creating a number of block equal to $(N + TPB - 1)/TPB$, where N represents the number of points and TPB stands for the threads per block.

The kernel function assigns to each thread the task of computing the distance between one single data point and all the possible centroids in order to find the closest. Then it returns the index of the cluster assigned to that point.

3 Methodology

3.1 Data preparation

We generated 100, 500, 1000, 10 000, 50 000, 100 000, 250 000 and 1 000 000 random data points, belonging to different ranges and clusters.

In our implementation of K-means, the initial centroids of each cluster were chosen randomly among the data points, by selecting a random index (and the next one because of the structure $[x_1, y_1, x_2, y_2, ...]$) according to a fixed seed in order to have the same initialization for both sequential and parallel implementation.

The coordinates of the points and the centroids were then copied to the device so that they can be used later for the kernel. Once the assignment is computed and the results are copied back to the host, the update function aimed to update the new centroids as explained before. The new centroids, computed by a sequential function were sent to the device for the kernel calling of the next iteration.

3.2 Performance analysis

To compare the performance of the CPU and GPU (of both the machines *axm4* and *kronos*) we will vary the size of N and measure the time spent in each ROI of the code.

In the sequential code in C++ we defined the ROI:

- **ROI**$_{while}$ for the duration of one iteration;
- **ROI**$_{assignment}$ for the kMeansClusterAssignment();

In the parallel code in CUDA we defined the ROI:

- **ROI**$_{CP0}$ for transferring initial centroids, datapoints and cluster's sizes from CPU (HOST) to GPU (DEVICE);
- **ROI**$_{while}$ for the duration of one iteration;
- **ROI**$_{assignment}$ for the kMeansClusterAssignment();

- $\mathbf{ROI}_{CP1}$[1] for copying centroids and assignments from GPU (DEVICE) to CPU (HOST);
- $\mathbf{ROI}_{CP2}$[1] for transferring new centroids from CPU (HOST) to GPU (DEVICE) after the kMeansCentroidUpdate();

[1] These two values of copying time will be accumulated in a singular one in the figures of next sections for simplicity.

4 Results

For both the CPU and the GPU we visualized (Fig. 1) all the data points and their final assigned clusters to be sure that the results what we were obtaining were equal and therefore the performance could be compared.

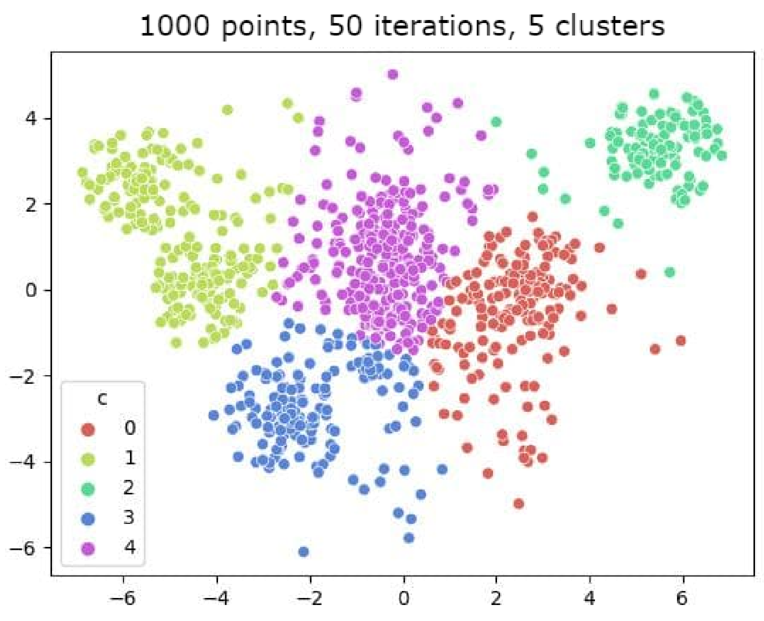

Figure 1: Example of K-means clustering results for both the CPU and GPU.

The results presented in this sections has been computed by the comparison between our sequential and parallel implementation of K-means clustering on all the data set previously mentioned, choosing as parameter $K = 10$ clusters in all the experiments and letting the algorithm run until 50 iterations.

For the following graphs take into consideration that the results of kronos machine are not so reliable, because of the continuous strong occupancy of the machine by other users during our experiments, as you can specially observe in Fig. 6 for $N = 10000$.

The results of the execution time comparison (Fig. 2) prove our assumptions about GPU performing faster thanks to the parallelization of the KmeansClusterAssignment function.

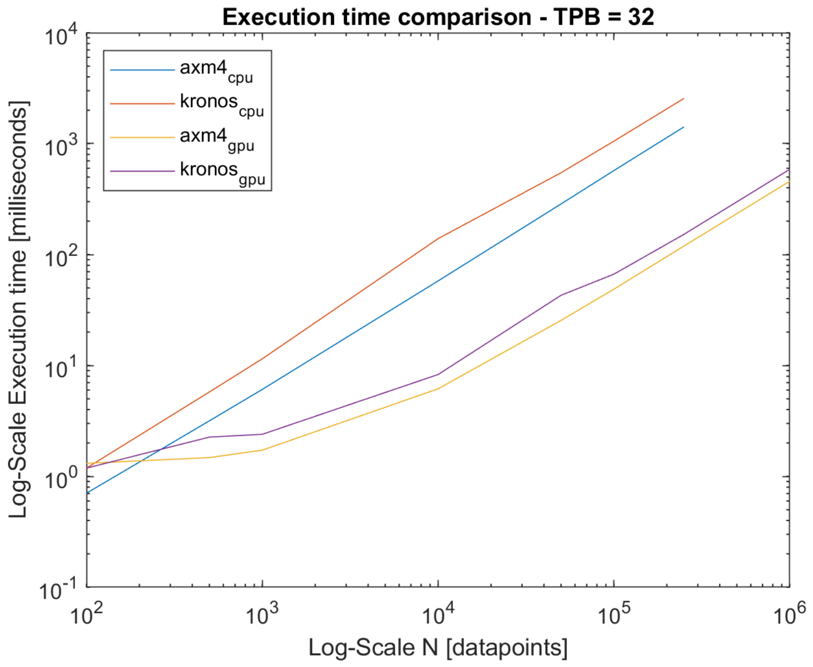

Figure 2: Total execution time comparison for all the machines. The values come from the mean of more experiments, in which we focused on the time spent in the while cycle for 50 iterations.

The histograms in Fig. 3 and 4 summarize the time spent in each one of the most significant parts of the K-means algorithm in milliseconds, taking as sample only one iteration. In further detail, all the time measurements showed in the histograms are the mean of 5 trials of 50 iterations of the while.

It is important to point out how the time needed for the assignment functions is quite undetectable in the bars regarding the GPU, because of the parallel implementation. This is another proof of the success of the experiment.

However, as we can see in Fig. 3 and 4 the execution time for $N = 100$ is longer for GPU than CPU, as the GPU takes significant amount of time for copying data from the host to device and back from the device to host making it inefficient for small data sets. Moreover, the sequential code's segmentation failed for N =1000000.

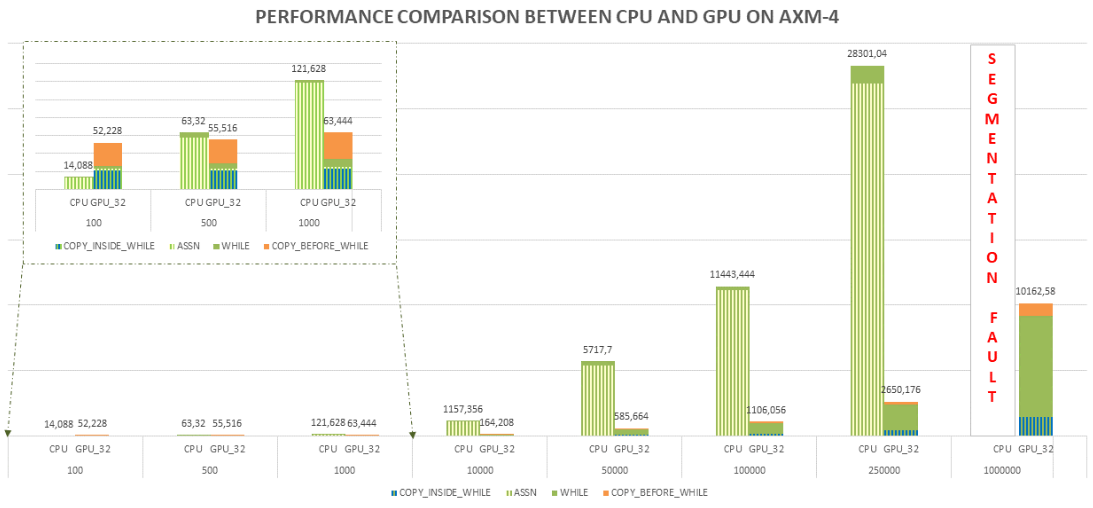

Figure 3: Execution time (of one iteration) comparison of CPU and GPU on machine axm-4 labeled with the times spent in each ROI.

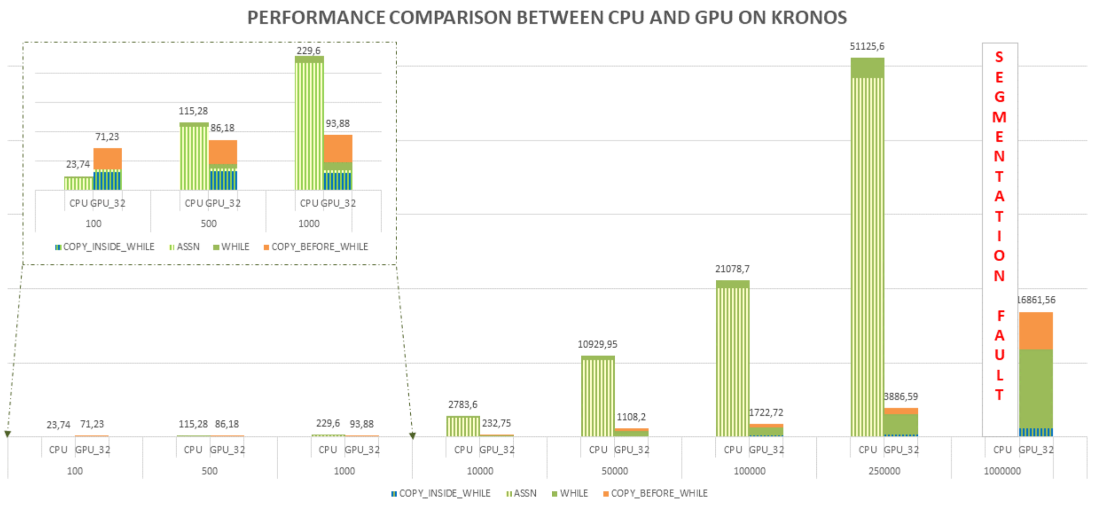

Figure 4: Execution time (of one iteration) comparison of CPU and GPU on machine kronos labeled with the times spent in each ROI.

The number portrayed on the top of each bar represents the total time of interest, seen as the sum of all the ROI, in particular it means:

- ROI$_{while}$ for the sequential code, because the assignment function is inside the while cycle;
- ROI$_{while}$+ROI$_{CP0}$ for the parallel one, because the assignment function, copy 1 and copy 2 happen inside the while cycle.

In each experiment we also computed the achieved speedup as:

$$S = \frac{T_{cpu}}{T_{gpu}}$$

obtaining significant values for both of the machines, as shown in Fig. 5 and 6, apart from the problem previously mentioned concerning kronos' results.
As expected, the larger number of data points N, the better values of speedup we achieved.

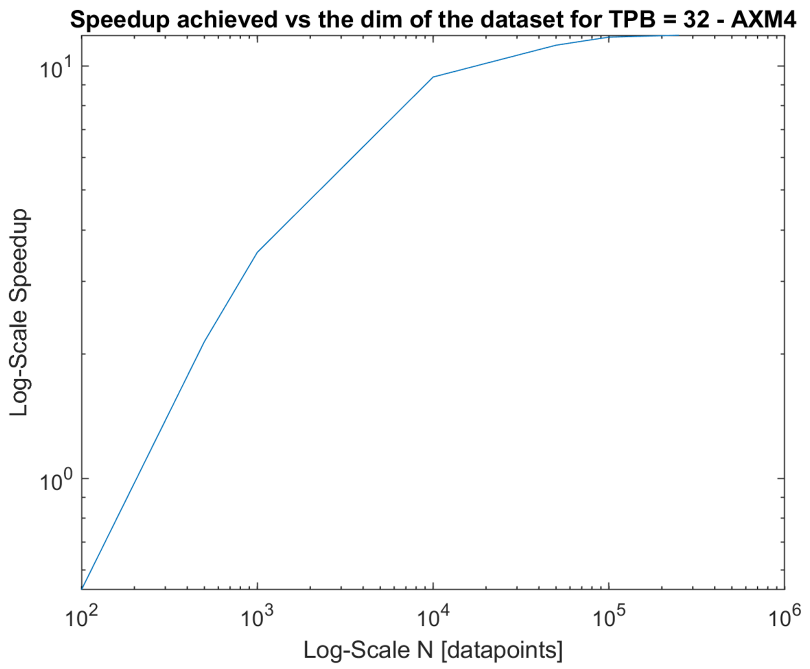

Figure 5: Speedup achieved by axm-4 machine according to the size of each data set.

The speedup for $N = 1000000$ could not be computed because of the incapability of both machines to solve the problem, in the sequential way.

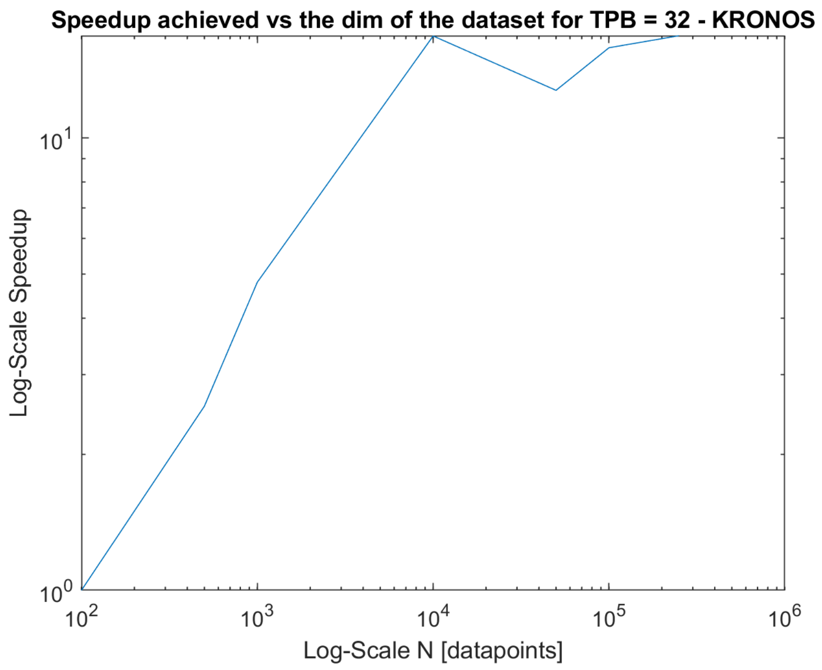

Figure 6: Speedup achieved by kronos machine according to the size of each data set.

Actually, we repeated all those experiments also by changing the number of threads per block, TPB; in particular choosing $TPB = [32, 128, 512]$ but the results were about the same. The only little difference came from the experiment with $TPB = 512$, as we had a very small increase of execution time because of the instantiation of bigger blocks, but in our analysis it was negligible.
For this reason, we have reported the results concerning experiments with 32 threads per block.

Appendix A - specification of axm-4 machine

Operating system:

Name:	Ubuntu
Version:	20.04.3 LTS (Focal Fossa)

CPU information:

Architecture:	x86_64
Model name:	AMD Ryzen Threadripper 1950X 16-Core Processor

GPU information:

Name:	TITAN Xp
Total global memory:	12787122176 bytes
Total shared memory per block:	49152 bytes
Total registers per block:	65536
Maximum threads per block:	1024
Maximum dimension 0 of block:	1024
Maximum dimension 1 of block:	1024
Maximum dimension 2 of block:	64
Number of multiprocessors:	30

Compiler version for C++: g++ (Ubuntu 9.3.0-17ubuntu1 20.04) 9.3.0 Copyright (C) 2019 Free Software Foundation, Inc.

Compiler version for CUDA: nvcc: NVIDIA (R) Cuda compiler driver Copyright (c) 2005-2021 NVIDIA Corporation Cuda compilation tools, release 11.2, V11.2.152

Appendix B - specification of kronos machine

Operating system:

Name:	Ubuntu
Version:	18.04.3 LTS (Bionic Beaver)

CPU information:

Architecture:	x86_64
Model name:	Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz

GPU information[1]:

Name:	Tesla V100-SXM2-32GB
Total global memory:	34089730048 bytes
Total shared memory per block:	49152 bytes
Total registers per block:	65536
Maximum threads per block:	1024
Maximum dimension 0 of block:	1024
Maximum dimension 1 of block:	1024
Maximum dimension 2 of block:	64
Number of multiprocessors:	80

Compiler version for C++: g++ (Ubuntu 7.5.0-3ubuntu1 18.04) 7.5.0 Copyright (C) 2017 Free Software Foundation, Inc.

Compiler version for CUDA: nvcc: NVIDIA (R) Cuda compiler driver Copyright (c) 2005-2018 NVIDIA Corporation Cuda compilation tools, release 10.0, V10.0.130

[1]On this machine there are two Cuda devices available, however the specification of the GPU of both is the same.

Attachments

- points_generator.py
- cluster_displayer.py
- datapoints.zip
- kmeans_sequential.cpp
- kmeans_parallel.cu
- README.txt

References

[1] A Clustering Method Based on K-Means Algorithm: Li, Y., Wu, H. (2012), Physics Procedia, 25, 1104–1109. doi:10.1016/j.phpro.2012.03.206

[2] CUDA C++ Programming Guide: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[3] K-Means algorithm for CUDA: https://github.com/alexminnaar/cuKMeans