# K-Means clustering

## A GPU implementation

**Mattia Bennati**

**Arezzo 05/07/23**

# Introduction

## Project summary

Objective: The aim of the project is to outline the differences between a sequential and a parallel implementation of the K-Means algorithm, by utilizing some of the GPU programming paradigms and comparing the timings required by the two versions to terminate their computations.

This project makes use of the CUDA compiler, tools and APIs in order to implement parallelism and this can potentially lead to a much faster execution compared to the serial execution of the instructions.

In order to make tests and performance evaluations consistent and reliable, both the sequential and the parallel version of the project share the same execution seed.
This ensures that both versions follow the same exact centroids initializations and clustering steps.

Note: the seed is randomly generated, if not proivided in the configuration file.

# Introduction

## Project summary

In order to simplify the benchmarking phase of both the implementations, a single executable file has been generated. The program is capable of running both the versions based on a configuration file.

The configuration allows also to specify the maximum number of records to process and some parameters for the K-Means algorithm like: the number of clusters to use, the maximum tolerance for the convergence evaluation and the maximum number of allowed iterations.
Note that the number of clusters must be greater than 1 and smaller than the number of processed records, since we are trying to meaningfully partition them based onto their characteristics.

At the end of each execution, the final centroids positions are saved to the disk along with the related reporting.

By specifying a particular seed it is possible to get predictable centroids' initializations and sequences of clustering steps.

The source code of the project can be retrieved at the following location:

https://github.com/Scrayil/k-means

# K-Means Clustering

## Algorithm overview

The K-Means clustering is an unsupervised machine learning algorithm that is used to partition a given dataset into multiple groups (clusters), based on the similarities and correlations between the contained records.

By being unsupervised, the algorithm is capable of detecting complex patterns and trends autonomously, with no prior guidance.

The objective is to minimize the intra-cluster differences between the points and maximize the inter-cluster ones instead.

The differences between the points are evaluated with a specific metric, in this case the Euclidean distance. Since not all data can be represented the same way, it is necessary to encode it so that the distances can actually be computed and properly evaluated.

This is done by encoding the given data so that each record corresponds to an element of the features' space. The features' space is a vector space in which all the fields of a record are treated as coordinates and the distance between them becomes meaningful.

The benchmarking dataset has been specifically generated to avoid the encoding necessity, so the data pre-processing strategy has not been implemented.

# K-Means Clustering

## Algorithm implementation

The algorithm‘s workflow relies over the positions of the centroids, that are the average centers of all the clusters in the features’ space.
A centroid represents an approxymation of its cluster’s points’ characteristics.

The steps required for the clusterization are excuted iteratively until all the centroids converge to their final positions or the maximum number of iterations is reached, if specified.

Implemented steps:

1. **Initialization**: K centroids are placed into the features’ space starting by randomly occupying some of the data points’ positions. In the case of a very big dataset, the input data is divided into multiple batches that are processed consecutively.
   a. This leads to the so-called Mini-Batch K-Means algorithm in which a single batch of data at a time is processed.
   b. It provides an approxymation of the K-Means clustering, but allows to process input data of any size.

2. **Assignment**: Here the distances between each data point and all the centroids are computed. Each data point is then assigned to the cluster of which the nearest centroid represents the average position.
   a. This step is required for associating points to clusters based on their features’ similarities.

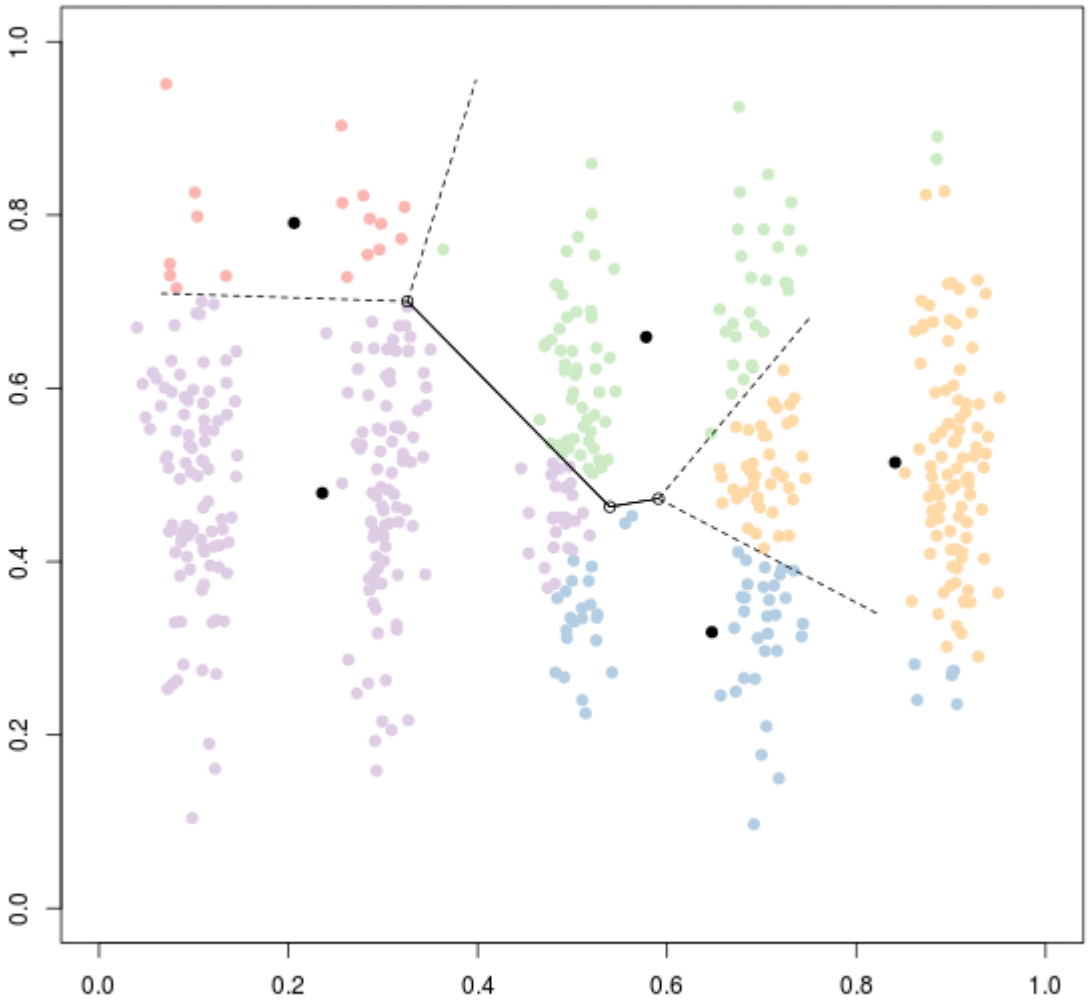
# K-Means Clustering

## Algorithm implementation

3. **Update**: This step consists into computing the new centroids positions by calculating the average coordinates of the points contained inside the related clusters. The centroids then occupy those positions to better approxymate the cluster's characteristics. The new positions will be used during the next iteration to compute the distances again.

4. **Convergence evaluation**: This phase aims at evaluating the centroids stability. This is done by comparing each centroid's position with the previous one and calculating the absolute percentage shift.
   a. If the variation is greater than the maximum allowed tolerance, the centroid does not converge and the previous two steps are performed again, until the maximum number of iterations is reached. The algorithm stops whenever the iterations exceeded their threshold or all the centroids converge.
   b. The maximum allowed tolerance can also be set to 0, in this case we are seeking for the absolute centroids convergence, guaranteed by their initializations inside the feature space.

Not all the solutions are optimal because the initial centroids positions can have an impact on the overall clustering accuracy.
It would be good to perform multiple executions and choose the results in which the intra-cluster differences have been minimized.




Visual representation of 5 different clusters from “K-Means clustering and Vonoi sets”, https://freakonometrics.hypotheses.org/19156. Accessed 05 July 2023.

# Testing machine

## Specifications

The computer used for the development and the testing phases of the project has the following characteristics:

CPU: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz with 8 physical cores

RAM: 2 x 16GiB SODIMM DDR4 Synchronous 3200 Mhz

GPU: GA107M [GeForce RTX 3050 Ti Mobile] with 2560 CUDA cores

# Mini-Batches

## Logic

Big datasets are partitioned into smaller batches of the same size. This allows to never exceed the maximum available memory, by limiting the data structures' sizes.

The size of the batch is computed in both cases at runtime, based on the available GPU resources by always leaving 15% of them unused.

After that, the first batch of data is processed, and the resulting centroids are then kept for the clusterization of the next one.

For what concerns the parallel version of the program, the total number of threads is then reduced and shared evenly among the launched blocks on the GPU.

The aim here is to initially launch as many threads as data points, in order to process all of them simultaneously, and then having as many blocks as clusters, in which the work is shared across all the threads (co-workers).

# Sequential version

## Limitations

Since the sequential version of the algorithm is executed entirely on the host, there is no need to manually allocate the memory. All the data has been handled by using vectors.

Even if it wouldn't be strictly required, in order to have a consistent comparison between the two algorithm implementations, the sequential version follows the same mini-batch clustering strategy as the parallel version.

The time required for clustering the data depends primarily on the number of records to process, since one point at a time is processed and the initial centroids positions might have an impact on the total number of iterations to perform.

The main concern here is that by having a very big dataset, there might be too many points to process and this can be very time consuming.
The limited available memory would have surely been a concern too, if the Mini-Batch logic wasn't applied.

# Parallel version

## Overcoming some limitation

The GPU implementation relies onto the manual allocation of the required memory resources and on the definition of two kernel functions.

The memory management is performed by creating pointers to arrays for both the host and the device, using the CUDA APIs functions, like “cudaMalloc” and “cudaFree”.

- Since GPUs primarily handle arrays of data, in order to not exceed the available resources and to have at least as many threads as data points, the Mini-Batch clustering approach introduced previously has been implemented.

Both the kernels are launched by the host and executed on the device, by instructing the compiler with the “**__global__**” keyword.
Since the two functions are executed multiple times, synchronization barriers have been set between them in order to wait for all threads to finish their computations.

The “**cudaDeviceSynchronize()**” call used, makes the CPU wait for the kernel computation to end before executing the next instructions on the host side.

# Parallel version

## Overcoming some limitation

The **first kernel** is responsible for generating and updating the clusters by assigning the data points to them.

- All the available threads are evenly partitioned among the blocks in order to: process each data point simultaneously, compute the distances from the centroids and assign it to the nearest cluster.

**The second** one instead updates and evaluates all the centroids' at the same time, by launching as many blocks as centroids and allocating all the available threads evenly to them.

This function contains two arrays shared across the block, to which each thread can write the result of it's computation.

- This allows to have multiple co-workers processing different groups of points fo each cluster.
- This function presents two points where synchronization is necessary.
  - The first one is required to ensure that all the threads in the block have completed their task so that the first thread is allowed to process the partial results.
  - The second one occurres just after computing the average shift of the evaluated centroid, as this denies other threads to modify the shared variables while they are being accessed.

# Parallel version

## Limitations

There is **criticality** in the second kernel, as if the number of clusters is very big or even similar to the number of points in the batch, there wouldn't be enough co-workers and implementing parallelism would not result into any real advantage, compared to the sequential version.

This leads to the conclusion that smaller numbers of clusters and big datasets are optimally executed by the GPU.

This problem exists because of the limited resources available on the GPU, as by using better or multiple GPUs this could actually be solved.

Note: it is also important to keep in mind that there are 4 synchronization barriers overall, and in some cases the overhead in the memory management and the kernel launches might lead to a slower execution of the program in the case of small datasets.

# Performance evaluation

## Results

The average speedup has been computed by comparing the elapsed times of the two versions, and it has been shwon in the following table by taking into consideration both the number of clusters and the number of points.
In order to calculate the speedup, the following formula has been used:

$$S_P = \frac{t_s}{t_p}$$

Where $t_s$ represents the execution time of the sequential version and $t_p$ the timing of the parallel one.

Table 1. Average speedup by number of clusters and data points

| C \ P | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ | X |
|---|---|---|---|---|---|---|
| 20 | 0.137 | 2.107 | 7.613 | 11.271 | 12.061 | 5.533 |
| 40 | 0.549 | 3.34 | 7.8 | 9.789 | 10.642 | 5.355 |
| 60 | 0.161 | 4.029 | 8.153 | 9.189 | 9.264 | 5.134 |
| 80 | 0.158 | 4.405 | 8.376 | 7.223 | 7.919 | 4.682 |
| 100 | 0.162 | 4.154 | 7.801 | 8.607 | 6.816 | 4.591 |
| X | 0.196 | 3.086 | 6.769 | 8.076 | 8.301 | 4.417 |

* The "X" placeholder indicates that the field has not been fixed to a specific value. All related values have been taken into consideration.

# Performance evaluation

## Conclusion

The highest speedup has been achieved with 20 clusters and $10^7$data points.

Using 20 to 40 clusters with increasing data points worked best for the GPU implementation.
The critical point occurs when the number of clusters is almost equal to the batch's size.
This happens because of the limited GPU resources that slow down the execution with high numbers of clusters, as there won't be enough co-workers threads.

Performance benefits increase with larger datasets as having many threads per block allows processing multiple data points simultaneously.

However, the overall speedup is not significant in this case as small batches grant no advantages due to the overhead in the kernel execution and the memory management.