PC-2022/23 Final-Term Project
Advantages of parallelism for the K-Means clustering algorithm: a GPU implementation

Mattia Bennati
7122582
mattia.bennati@stud.unifi.it

Abstract

The objective of this report is to analyze and evaluate the advantages of implementing parallelism through the GPUs utilization, in relation to the K-Means clustering algorithm. The report aims at highlighting how parallel programming can enhance the efficiency of the algorithm, and under which circumstances, with the eventually encountered limitations. More specifically, the benefits of parallelism have been outlined by implementing the algorithm in C++ and creating a parallel version of the program, in which part of the code is executed by a GPU.
The source code will be released publicly.

1. Introduction

The K-Means clustering is an unsupervised machine learning algorithm used to process a given dataset and partition the contained data into K unique clusters.
An algorithm is unsupervised if it works on unlabelled data. This means that it doesn't have any information on how the given records are related between themselves. The aim of these algorithms is to detect patterns and relationships within the input data, with no given prior direction.
A cluster is a subset of the initial data in which all the contained elements share common features or characteristics.
The K-Means algorithm relies on the position of the so-called "centroids". A centroid represents the average center position of all the elements in a cluster. It actually synthesizes the characteristics of all the enclosed data points.
The objective of the algorithm is to minimize the differences between the points inside each cluster while maximizing the ones in between all the clusters.
In order to achieve the best consistency during the comparison of the sequential and parallel execution of the program, the same algorithm has been implemented in both versions. The two versions also share the same centroid initialization that is performed randomly, for each program execution. This ensures that the same clustering steps are performed in both cases.
Since the objective of this report is to evaluate the differences in terms of performances, a dataset of 10 million records has been specifically generated for benchmarking purposes.

2. The K-Means Algorithm

2.1. Clustering logic

The algorithm consists of a sequence of steps in which the data points are iteratively assigned to their nearest cluster, and the related centroids positions are consecutively updated.
Since the given data could eventually be multi-dimensional, it is usually necessary to change its domain by creating a space in which the distance between the provided points becomes meaningful and can actually be computed. This means that non-numerical data must be encoded first, so that

it can be evaluated properly.
The given dataset allows to work directly with no encoding necessity, and the preprocessing logic of the data has not been implemented.
The implemented clustering logic comprises the following steps:

1. Initialization: K centroids are randomly placed into the features' space starting from the given dataset. In the case of a large dataset, points are partitioned into smaller batches, but the initialization considers all of them.
   This leads to the implementation of the "Mini-Batch K-Means" algorithm in which a single batch at a time is processed. The result is an approximation of the K-Means algorithm, but this allows processing input data of any size.

2. Assignment: This phase consists of the computation of the distances in between each point and all the available centroids. Each data point is then assigned to the nearest cluster of which the centroid represents the center point, based onto a specific distance metric. In this case, the Euclidean distance between two n-dimensional points:

$$\text{dist}(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \qquad (1)$$

   where x is defined as:

$$x = (x_1, x_2, \ldots, x_n)$$

   and y as:

$$y = (y_1, y_2, \ldots, y_n)$$

   Given a dataset X, and having defined the nearest cluster with k, each assignment is performed as follows:

$$S_k = \{x \in X : \arg\min_j \text{dist}(x, \text{centroid}_j) = k\} \qquad (2)$$

   S_k is the set of data points assigned to the k-th cluster along with its centroid_j.

3. Update: After assigning all the points in the batch to their cluster, each centroid's position is updated by calculating the average coordinates of the contained data points. During the next iteration, the new centroids' positions will be used to compute the distances again. The update of a single centroid makes use of the following formula:

$$\text{centroid}_k = \left(\frac{1}{|S_k|} \sum_{x \in S_k} x_1, \ldots, \frac{1}{|S_k|} \sum_{x \in S_k} x_n \right) \qquad (3)$$

   in which the per-coordinate average in between all the data points is computed.
   $|S_k|$ represents the set's cardinality (size).

4. Convergence evaluation: This step focuses onto the evaluation of all the centroids' stability. The goal here is to make all centroids accurately approximate the average characteristics of their clusters. If the position shifts of any centroid are greater than the maximum allowed tolerance, the previous steps are repeated multiple times. The algorithm stops once all centroids converge or the maximum number of iterations is reached. By setting the maximum tolerance to 0, we are seeking for the absolute convergence of the centroids, guaranteed by their random initialization within the feature space.

$$V_k = \text{dist}(\text{centroid}_{k,\text{new}}, \text{centroid}_{k,\text{prev}}) \qquad (4)$$

   V_k is the total per-cluster centroid's variation used to verify if it actually converges.

5. Results: After convergence, the algorithm provides the detected clusters with their relative centroids. The solutions of the algorithm might not be optimal all the time as the initial conditions can have a big impact on them. So it would be useful to perform multiple initializations and executions, and select the solutions in which the distances of the data points contained in each cluster are minimized.

The final clusters' representation of a 2-dimensional dataset looks as follows:

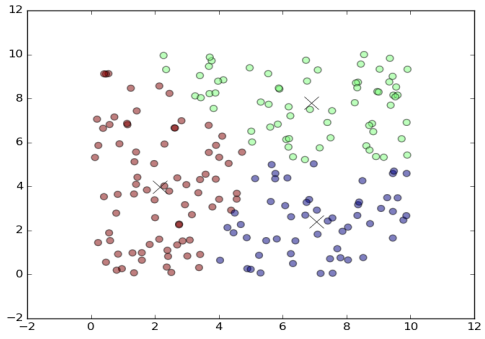

Figure 1. Clusters of data created with k-means [1].

2.2. GPU Implementation

By applying the above algorithm in C++ there is a limitation in parallelizing the code execution. There are two synchronization points that are strictly required in order to get the desired outcome, as some of the involved steps must be executed consecutively.
The algorithm consists of two main parts, for which two CUDA kernel functions have been accordingly defined. The parallel version of the project then relies on the kernels, that are executed on the GPU, to achieve better performances.

1. The first kernel manages the data points assignments
2. The second includes the centroids' updates along with their convergence evaluations

2.2.1 Resources allocation

In order to handle input data of any size and properly manage the memory requirements, the given dataset is divided into batches of the same size, and only one batch at a time gets processed.
This allows working on a good amount of data without encountering failed memory allocations caused by the limited available resources. More specifically, the batch size greatly depends on the number of available GPU threads.
A constraint to use no more than 85% of all the GPU resources has been set, and the memory allocations on both the host and the device are performed only once during initialization. Data points are copied to the GPU for each batch computation, and clusters along with their centroids are copied back to the host only after finishing processing a batch.

2.2.2 Assignments

As stated previously, the first kernel is responsible for computing all the data points assignments to their nearest clusters.
After copying the data points contained in the current batch to the device, the kernel is launched with the required number of blocks and threads, in order to handle all points simultaneously.
The kernel signature looks as follows:

```
__global__ void p_generate_and_optimize_clusters(int
↪  num_dimensions, int num_clusters, int
↪  data_points_batch_size, int
↪  actual_data_points_size, double* data_points,
↪  double* centroids, double* distances, double*
↪  clusters, bool* clusters_optimized)
```

The "__global__" keyword tells the compiler that the kernel is launched by the host, but executed on the device. All the pointers to the device-allocated memory are passed here as references in order to allow each thread to work with them.
The kernel is typically launched with a call like the following, in which the total number of blocks and threads per block is specified just before the invocation:

```
// GENERATING/OPTIMIZING THE CLUSTERS FOR THE CURRENT
↪  BATCH
p_generate_and_optimize_clusters<<<BLOCKS, THREADS>>>(
        num_dimensions,
        this->k,
        data_points_batch_size,
        actual_data_points_batch_size,
        dev_data_points,
        dev_centroids,
        dev_distances,
        dev_clusters,
        dev_clusters_optimized
);
```

The kernel consists of two sections: the first one in which each thread re-initializes the data point's clusters and distances sections to discard the previous values, and the second one in which the assignment is actually performed.
The cluster clearing procedure has been implemented in the following way:

```
// CLEARING THE CLUSTERS AND DISTANCES RELATED TO THE
↪  CURRENT DATA_POINT
if (unique_index == 0)
    // Reset the global convergence value
    *clusters_optimized = true;
for(int cluster_num = 0; cluster_num < num_clusters;
↪  cluster_num++) {
    int base_cluster_index = cluster_num *
↪  data_points_batch_size;
    for(int dimension = 0; dimension < num_dimensions;
↪  dimension++)
        clusters[base_cluster_index * num_dimensions +
↪  point_base_index + dimension] = nan("");
    distances[base_cluster_index + unique_index] = 0;
}
```

The assignment instead is performed by computing all the data point's distances from the current centroids and by minimizing them:

```
// UPDATING THE CLUSTERS

if (unique_index < actual_data_points_size) {
    // Executed by all threads up to the
↪  actual_data_point_size
    int min_dist_index = 0;
    for (int cluster_num = 0; cluster_num <
↪  num_clusters; cluster_num++) {
        double squared_differences_sum = 0;
        for (int j = 0; j < num_dimensions; j++) {
            double curr_difference =
↪  data_points[point_base_index + j] -

↪  centroids[cluster_num * num_dimensions + j];
            double squared_difference = curr_difference
↪  * curr_difference;
            squared_differences_sum +=
↪  squared_difference;
        }

        double dist = sqrt(squared_differences_sum);
        distances[cluster_num * data_points_batch_size
↪  + unique_index] = dist;

        if (dist < distances[min_dist_index *
↪  data_points_batch_size + unique_index])
            min_dist_index = cluster_num;
    }

    // Assigning the data point to the cluster with the
↪  nearest centroid
    for (int i = 0; i < num_dimensions; i++)
        clusters[min_dist_index *
↪  data_points_batch_size * num_dimensions +
                point_base_index + i] =
↪  data_points[point_base_index + i];
}
```

2.2.3 Updating the centroids

The kernel responsible for the update of each centroid and the relative convergence evaluation is launched with as many blocks as the number of clusters.
It is defined as a "__global__" function too.
Since the number of blocks matches the number of clusters: the greater the number of clusters will be, the smaller will the gain in terms of performances be.
Multiple threads cooperate on the same cluster by evenly sharing the work between themselves. This allows each thread to work simultaneously on a different subset of the cluster's data points.
The work is shared between threads within the same block by using shared memory in order to temporarily store each result:

```
// Block = Cluster
// Thread = Worker
__shared__ double cluster_sums[N_CLUSTERS_CO_WORKERS];
__shared__ double
↪  workers_cluster_elements[N_CLUSTERS_CO_WORKERS];
```

This is the code executed by all the threads in the same block:

```
int base_cluster_data_index = blockIdx.x *
↪  data_points_batch_size * num_dimensions;
int remaining_points = actual_data_points_size %
↪  N_CLUSTERS_CO_WORKERS;
int worker_inner_batch = actual_data_points_size /
↪  N_CLUSTERS_CO_WORKERS;
int curr_worker_inner_batch = worker_inner_batch;
int curr_cluster_start_index = threadIdx.x *
↪  worker_inner_batch;

if(remaining_points > 0) {
    worker_inner_batch += 1;
    curr_cluster_start_index = threadIdx.x *
↪  worker_inner_batch;
    if(threadIdx.x < remaining_points)
        curr_worker_inner_batch = worker_inner_batch;
    else if(threadIdx.x > remaining_points) {
        curr_cluster_start_index = remaining_points *
↪  worker_inner_batch + (threadIdx.x -
↪  remaining_points) * (worker_inner_batch - 1);
    }
}

// This loop is executed by N co-workers for faster
↪  executions
double centroids_shifts_sum = 0.0;
for (int dimension = 0; dimension < num_dimensions;
↪  dimension++) {
    // Resetting the shared memory variables
    cluster_sums[threadIdx.x] = 0;
    workers_cluster_elements[threadIdx.x] = 0;
    for (int point_index = curr_cluster_start_index;
↪  point_index < curr_cluster_start_index +
↪  curr_worker_inner_batch; point_index++) {
            int cluster_data_index =
↪  base_cluster_data_index + point_index *
↪  num_dimensions + dimension;
            if (!isnan(clusters[cluster_data_index])) {
                cluster_sums[threadIdx.x] +=
↪  clusters[cluster_data_index];
                workers_cluster_elements[threadIdx.x]
↪  += 1;
            }
    }

    // Syncs all the threads in the block (Each block
↪  is bound to a specific cluster)
    __syncthreads();
```

The "__syncthreads();" call is used to ensure that all the threads in the block have finished their computation. At this point, only one thread is allowed to calculate the centroid's coordinate shift in comparison to its previous position:

```
    if (threadIdx.x == 0) {
        double curr_sum = 0.0;
        double cluster_elements = 0.0;
        for (int i = 0; i < N_CLUSTERS_CO_WORKERS; i++)
↪   {
            curr_sum += cluster_sums[i];
            cluster_elements +=
↪   workers_cluster_elements[i];
        }

        int curr_centroid_dim_index = blockIdx.x *
↪   num_dimensions + dimension;
        // Gets a reference to the current centroid's
↪   dimension
        double &curr_centroid_dim_val =
↪   centroids[curr_centroid_dim_index];
        // The previous dev_centroids positions are
↪   saved in order to evaluate the convergence later
↪   and to check if
        // the maximum tolerance requirement has been
↪   met.
        prev_centroids[curr_centroid_dim_index] =
↪   curr_centroid_dim_val;
        curr_centroid_dim_val = curr_sum /
↪   cluster_elements;

        // Used to evaluate the convergence later
        centroids_shifts_sum +=
                (curr_centroid_dim_val -
↪   prev_centroids[curr_centroid_dim_index]) /
↪   curr_centroid_dim_val *
                100.0;
    }

    // Syncs all the threads in the block (Each block
↪   is bound to a specific cluster)
    __syncthreads();
}
```

After computing the arithmetic sum of all the dimensional shifts, that same thread is allowed to evaluate the centroid's convergence. Therefore, synchronization is required again:

```
if(threadIdx.x == 0) {
    // Evaluating the convergence for the current
↪   centroid
    if (fabs(centroids_shifts_sum) > max_tolerance) {
        *clusters_optimized = false;
    }
}
```

Even if only a single centroid doesn't converge, the global convergence flag is set to false. After waiting for all the threads to synchronize, the algorithm can be resumed with the next iteration.

3. Performance analysis

The testing phase has been carried out by executing both the sequential and the parallel algorithm implementations multiple times.
Both versions share the same centroids initializations and solution steps thanks to the generated random seed.
The performed tests consist of multiple executions with different combinations of clusters' and data points' numbers.

3.1. Speedup

In order to evaluate the benefits in terms of performances between the two versions of the program, the speedup metric has been calculated.
For simplicity, the following formula has been used for the computation:

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

in which t_s represents the elapsed time for the sequential version, required to execute the algorithm, and t_p the elapsed time of the parallel version.

3.1.1 First considerations

By fixing the number of data points and looking at the elapsed times of the two versions, it is possible to notice that the time required by the CPU grows much faster than the time required by the GPU, but in both cases the increment is strictly dependent on the number of clusters.
So the number of clusters has a big impact on the computation compared to the number of data points.
It's worth noticing that the bigger the input data size is, the more is beneficial to achieve parallelism thanks on the GPU.
By looking at the following graphs that outline the trend explained above, it becomes clear that even if it's possible to benefit from parallelism when it comes to a big number of data points, the maximum speedup is limited by the number of clusters.

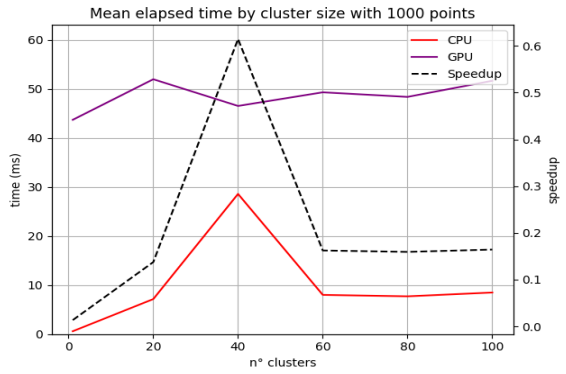

Figure 2. Interpolation of the mean elapsed time by number of clusters with 1,000 points.

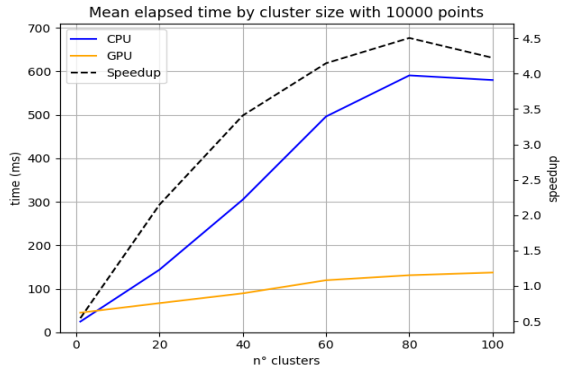

Figure 3. Interpolation of the mean elapsed time by number of clusters with 10,000 points.

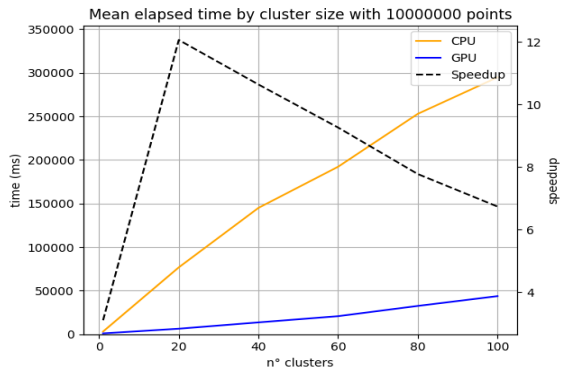

Figure 4. Interpolation of the mean elapsed time by number of clusters with 10,000,000 points.

With the current algorithm implementation, the overall speedup is optimal for 20 to 40 clusters. All of these considerations have been inferred and are outlined by the next graphs, in which the number of clusters has been fixed.

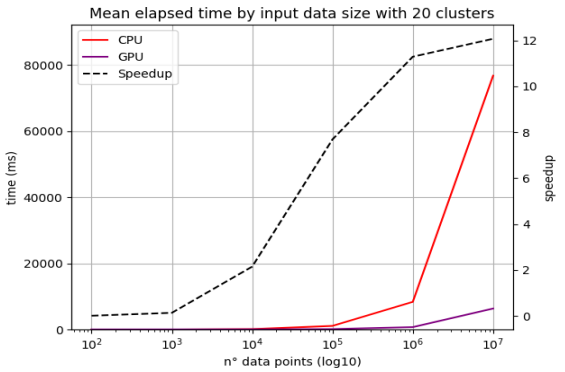

Figure 5. Interpolation of the mean elapsed time by input data size with 20 clusters.

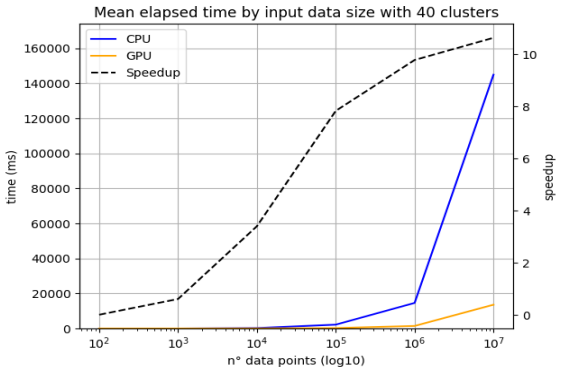

Figure 6. Interpolation of the mean elapsed time by input data size with 40 clusters.

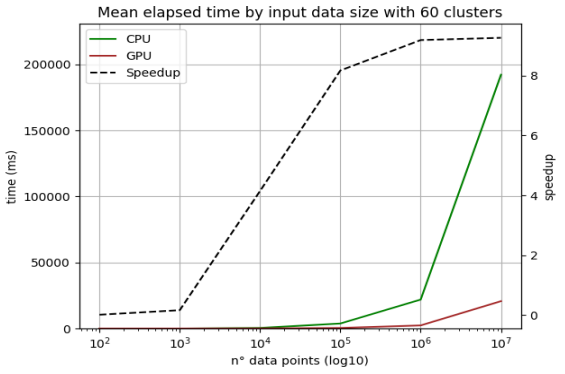

Figure 7. Interpolation of the mean elapsed time by input data size with 60 clusters.

The current GPU implementation relies on a dynamic resource reservation mechanism that is required for an optimal execution.
It is no wonder to have a better speedup with around 20 to 40 clusters as the host tries to launch as many blocks as clusters, and all the available threads are evenly partitioned. The aim here is to update the centroids' positions and evaluate their convergence by using multiple co-workers.
A smaller number of clusters results into a bigger number of available co-workers for each of them. This grants a gain in terms of speedup.

3.2. Achieved results

In order to evaluate the overall performances, each one of the following combinations of clusters and data points sizes has been tested 50 times. The tests have been structured with all the combinations in between the following sets:

$$\text{Clusters} = \{20, 40, 60, 80, 100\}$$
$$\text{Data points} = \{1000, 10000, 100000, 1000000, 10000000\}$$

The average speedup for each combination is showed in the following table in which "C" and "P" represent the number of clusters and data points respectively:

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.

3.3. Conclusion

By looking at the previous table, it is possible to notice that it is not beneficial to implement parallelism for a small number of data points, as the sequential version will be a lot faster in that case.
The highest speedup overall has been achieved with 20 clusters and 10^7 data points. This indicates that the optimal configuration for the current GPU implementation consists of using 20 different clusters with an increasing number of points.
The critical part here is reached when the number of clusters is very high and tends to be similar to the data batch's size. There wouldn't be enough per-cluster co-workers, and there will be no real benefit from implementing parallelism anymore. This criticality actually exists because of the limited GPU resources available.
The bigger the dataset is, the more the performance benefits are, and this can also be inferred by checking the data points averages over all numbers of clusters (latest row of the table). This is possible because many points can be processed at once by launching multiple threads per block. The program ensures that the batch size will never exceed the maximum number of threads available so that it is always possible to process all the points inside the batch simultaneously.
The overall speedup is not very high as the conducted tests consider also batch sizes with 10^3 points that are not leading to any tangible advantage if processed by the GPU. This is due to the existing overhead in launching and executing the kernel functions, in addition to managing: the related memory allocations and the data copies between the host and the device.
The source code of the project and its related reporting are publicly available on GitHub:
https://github.com/Scrayil/k-means

References

[1] S. Kovalev, S. Sintsov and A. Khizhniak. Implementing k-means clustering with tensorflow. Retrieved from *Altoros*: https://www.altoros.com/blog/using-k-means-clustering-in-tensorflow/, 2019.