Project 2: K-Means Clustering

Marco Di Rienzo
marco.dirienzo@stud.unifi.it

Abstract

The goal of this paper is to parallelize the K-Means Clustering algorithm and analyze the performance gain with respect to the sequential version.
One parallel versions has been created using the OpenMP framework and another using CUDA to run on the GPU.
The performance gain is measured in terms of speed-up $= \frac{\textit{sequential time}}{\textit{parallel time}}$.

1. Introduction

Both the sequential and the parallel version of the algorithm have been implemented in C++, the parallel ones using the OpenMP framework and CUDA C.
The dataset used to perform the tests is composed of 100,000 points of 2 dimensions and has been generated with the `make_blobs` function of *sklearn*.
Since the algorithm involves an initial random choice between the dataset points, a fixed seed has been imposed to both versions to yield comparable results.

2. Pseudocode

The implemented algorithm has the following pseudocode:

```
Algorithm 1: K-Means Clustering
Data: $k$, $max\_iterations$, $observations$
Result: $centroids$, $labels$
1. Select $k$ random $observations$ as starting
   $centroids$;
while not $max\_iterations$ do
    foreach $observation$ do
        2. Compute the distance between each
           $centroid$ and $observation$;
        3. Assign $observation$ to closest $centroid$;
    end
    4. Compute new $centroids$ (mean of
       $observations$ in a cluster);
end
```

3. Algorithm nature

The algorithm lends itself very well to parallelization because the main loop (steps 2. and 3. of Algorithm 1) is embarrassingly parallel, i.e. every iteration is independent from one another. What this means is that observations (points of the dataset) can simply be distributed among any number of threads while not having to worry about communication whatsoever.

4. Languages and frameworks

The codebase of the project has been written in C++. Python has been used to analyze the results.
One parallel version has been created using the framework OpenMP, which allows the code to remain pretty much the same while distributing the workload among threads, instructed by the use of directives. This framework is ideal in this case because the only section to parallelize is the *foreach* of Algorithm 1 which as said in Section 3 is embarrassingly parallel, and it can be done with a single directive (and some ad-hoc reduction functions).
The other one was made with CUDA C, which allows even more parallelization since GPUs are capable of managing thousands of threads.

5. Sequential implementation

The sequential implementation is basically a C++ translation of the pseudocode in Algorithm 1, with some nuances:

1. Observations and centroids are organized as Structure of Arrays to improve the alignment with cache lines and benefit of the memory bursts when accessing the various coordinates.

2. Step 4. is partially executed in the for loop, in particular the sum required to compute the mean is updated as soon as a point is assigned to a centroid. This avoids having to loop again on every observation, and is also perfectly compatible with the later parallelization.

6. Parallel implementation

Parallelizing step 1. makes no sense since k is usually small and there's practically no computation required to choose random points from the given ones.
The best we can do with step 4. is making each thread update a partial sum as soon as a point is assigned to a centroid, which is what we already explained in Section 5; the actual mean computation is not a parallelizable problem.
The embarrassingly parallel section of the algorithm is the for loop.

6.1. OpenMP

The directive **#pragma omp parallel for** allows splitting observation indices among threads, so that each one will be assigned a different subset of the dataset.
Since the assignments of observations to clusters are independent from one another, no synchronization is required.
However, since a partial sum of points in a cluster is maintained in each thread, a final reduction is needed to extract the total sum and be able to compute the mean; this is accomplished with the directive **#pragma omp declare reduction(...)** which allows to specify a reduction function to conglomerate the data of each thread.
Custom reduction functions have been created to deal with the structures used in the program.

6.2. CUDA

Since GPUs' threads context switch is negligible, each thread is assigned a single observation, this way, the for loop can be removed and so the maximum parallelization possible is achieved.
Observations and initial centroids are copied from the host to the device **global memory**, and also arrays to store new centroids, the number of points per cluster and point labels are created; centroids are then pulled to **shared memory** to improve access time since they have to be read by every thread.
Inside a thread, its point is compared with every centroid and the resulting label is written in the global memory array. The points per cluster and new centroids arrays are atomically increased, so no manual synchronization is required.
After the kernel completes, every thread of the grid is synchronized so final values of sum of points in a cluster and the number of points per cluster are available. Another kernel is launched to compute the new centroids directly on the device, so to avoid copying arrays back to host at every iteration.
Only when final centroids are computed, the centroids coordinates and point labels are retrieved from the device and saved.

7. Tests

Every test has been carried out on a CPU Intel Core i5-4670 (4-core) and a GPU Nvidia GeForce GTX 770 (compute capability 3.0).
The dataset used to perform the tests was generated with the `make_blobs` function of *sklearn* and is composed of 100,000 elements of 2 dimensions. The number of centroids is fixed to 10, if not stated otherwise.
The purpose of these tests was the measurement of the speed-up of the parallel implementations of the algorithm against the sequential one.
Wall-clock time was measured by C++ standard functions of the `<chrono>` library.
Each time the parallel and sequential version were given the same seed to make the initial random choice of centroids consistent between executions.

7.1. OpenMP

The parallel program was run with varying number of threads with the purpose to find the optimum value.
Time results and resulting speed-up of the trials for OpenMP are shown in Table 1, and also graphically in Figure 1, it can be seen that the speed-up grows sub-linearly (i.e. when the number of threads is quadrupled, the speed-up is not exactly multiplied by 4) to the best execution time, achieved with **4 threads**, that is the exact number of CPU cores. This is because with less threads obviously the amount of work per thread is greater and thus more time is required to complete.
With more threads, it's true that each one has less points to process, but since only 4 cores are available, only 4 threads can be executed simultaneously, and there's also the overhead of switching threads in and out of the processors.
This is especially true in our case since each thread doesn't need to wait for any resources and thus there's no advantage in having more threads in the queue.
In fact, with 5 threads and beyond the speed-up remains constant at about 2.4.

7.2. CUDA

According to the **CUDA Occupancy Calculator** the number of threads per block was set to 128; the corresponding number of blocks is $\lceil \frac{N_{points}}{128} \rceil$.
No variations in the speed-up were noticed by using different numbers of threads per block.
The first test was done increasing the dataset size to appreciate the scaling of the various parallelization methods.
Results are shown in Table 2, and also graphically in Figure 2, it can be seen that the sequential and OpenMP implementations scale linearly to dataset size, while the CUDA one scales sub-linearly, with CUDA gaining more and more advantage as the size increases due to the maximum paral-

lelization of a single point per thread.
Another test was done by increasing the number of centroids, since the loop on the centroids is executed inside the thread, the execution time grows linearly for every implementation, but the greater the parallelizzation, the greater the speed-up, which with CUDA rises up to about 30 from 100 centroids on. Results are shown in Table 3, and graphically in Figure 3.

N. of Threads	1	2	3	4	5	6	7	8	9	10
Time Sequential (ms)	275									
Time Parallel (ms)	272	136	94	78	130	114	102	111	119	110
Speed-up	1.01	2.02	2.93	3.53	2.12	2.41	2.70	2.48	2.31	2.50

Table 1. Execution time (milliseconds) and resulting speed-up of sequential vs OpenMP parallel implementation, with number of threads from 1 to 10.

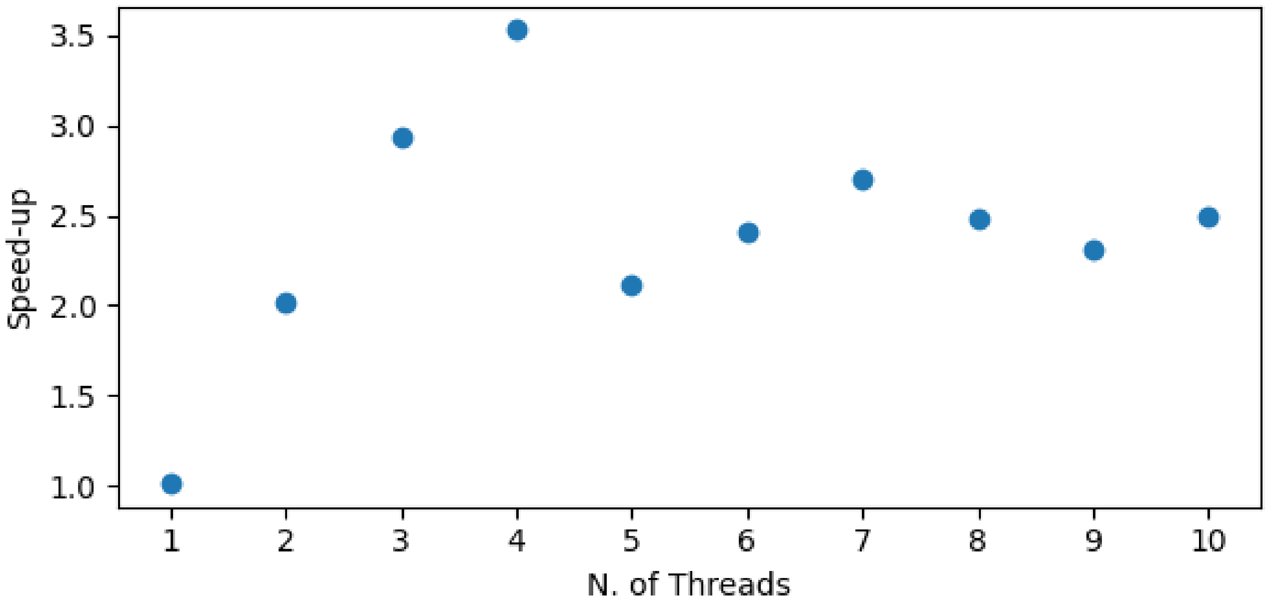

Figure 1. Plot of OpenMP speed-up values against number of threads displayed in Table 1.

Dataset size (1e4)	1	2	3	4	5	6	7	8	9	10
Time Sequential (ms)	27	55	76	106	132	152	186	207	246	275
Time OpenMP (ms)	7	17	28	32	41	49	61	67	77	78
Time CUDA (ms)	4	5	8	8	10	13	13	16	18	19
Speed-up OpenMP	3.86	3.24	2.71	3.31	3.22	3.1	3.05	3.09	3.19	3.53
Speed-up CUDA	6.75	11.0	9.5	13.25	13.2	11.69	14.31	12.94	13.67	14.47

Table 2. Execution time (milliseconds) and resulting speed-up of sequential vs OpenMP vs CUDA parallel implementation, with dataset size from 10^4 to 10^5.

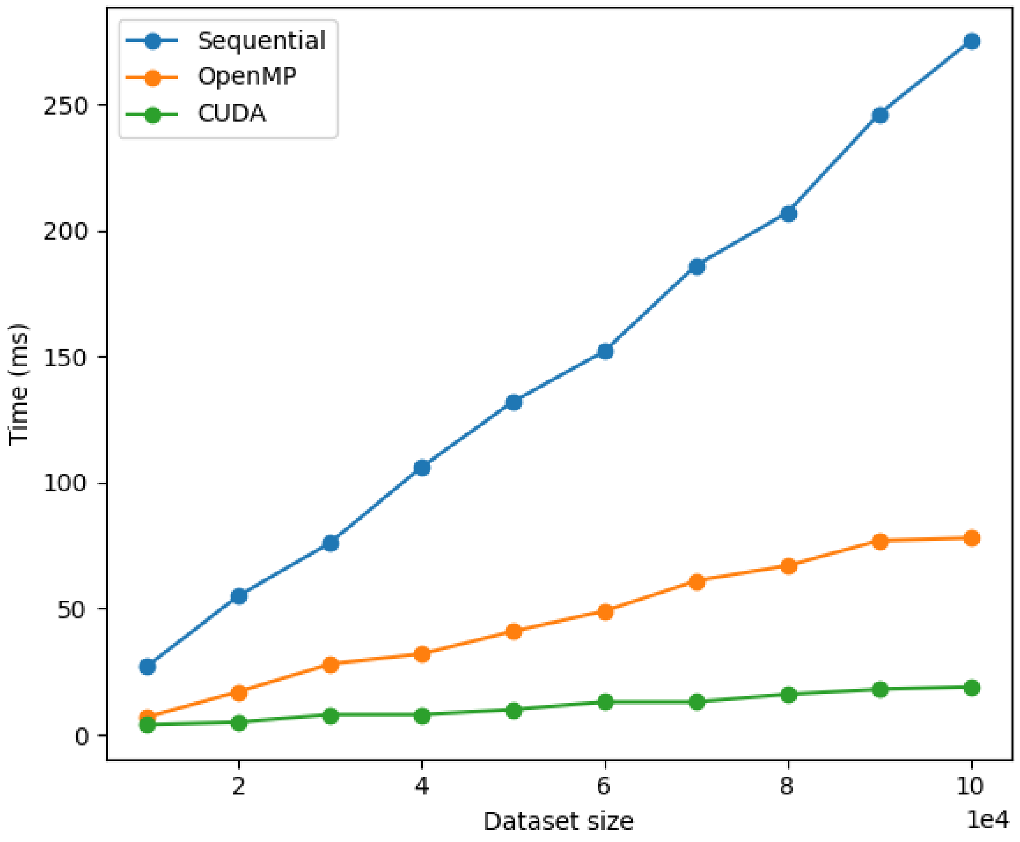

Figure 2. Plot of execution time against dataset sizes displayed in Table 2.

Centroids number	2	102	202	302	402	502
Time Sequential (ms)	117	3088	5514	7986	10351	12760
Time OpenMP (ms)	37	929	1717	2441	3272	4016
Time CUDA (ms)	44	106	206	323	404	501

Table 3. Execution time (milliseconds) of sequential vs OpenMP vs CUDA parallel implementation, with centroids number from 2 to 502.

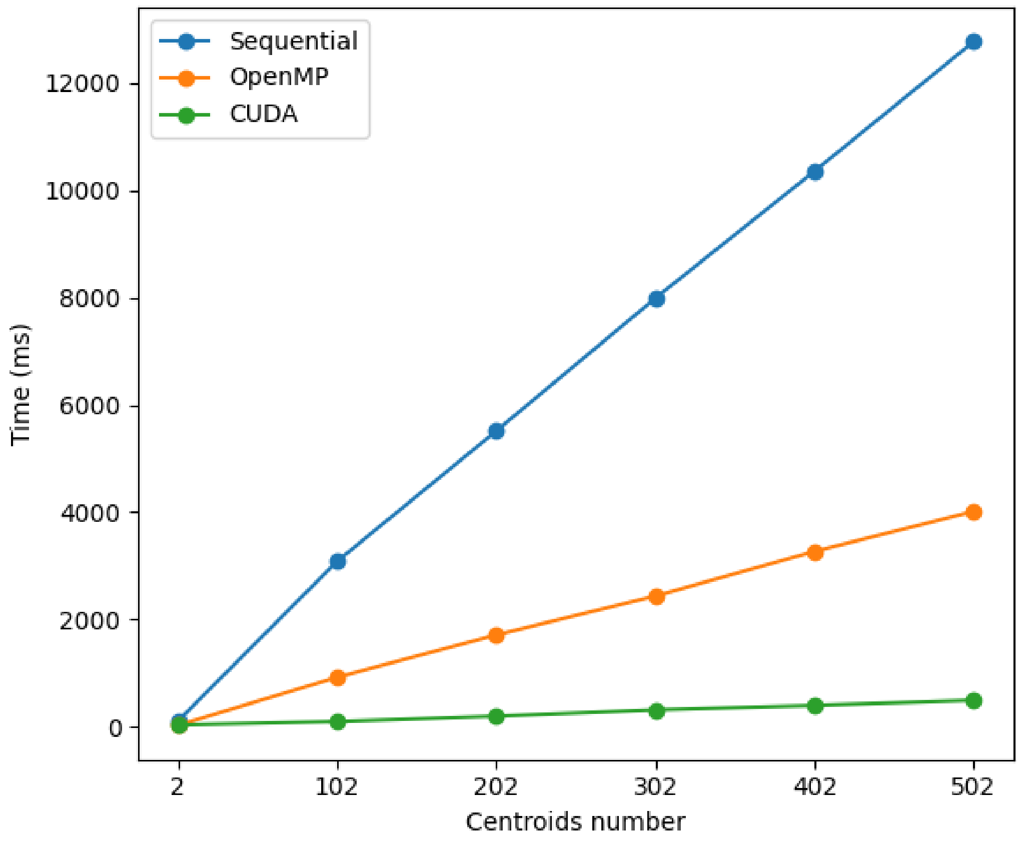

Figure 3. Plot of execution time against centroids number displayed in Table 3.