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 converged* or 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
```

* convergence is reached when the new centroids are *close enough* to the previous ones.

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 Array of Structures 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

OpenMP makes this really easy. Indeed, 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

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 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.
The parallel program was run with varying number of threads with the purpose to find the optimum value. Each time the parallel and sequential version were given the same seed to make the initial random choice of centroids consistent between executions.
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 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 abruptly drops by half, to less than the speed-up with only 2 threads! It keeps falling until below sequential performance somewhere above 10 threads.

N. of Threads	1	2	3	4	5	6	7	8	9	10
Time Sequential (μs)	3189									
Time Parallel (μs)	3060	1757	1239	1056	2117	2104	2303	2327	2438	2498
Speed-up	1.04	1.82	2.57	3.02	1.51	1.52	1.38	1.37	1.31	1.28

N. of Threads	11	12	13	14	15	16	17	18	19	20
Time Sequential (μs)										
Time Parallel (μs)	2794	3014	3924	3380	3050	3234	3224	3274	3459	3996
Speed-up	1.14	1.06	0.81	0.94	1.05	0.99	0.99	0.97	0.92	0.80

Table 1. Execution time (microseconds) and resulting speed-up of sequential vs parallel implementation, with number of threads from 1 to 20.

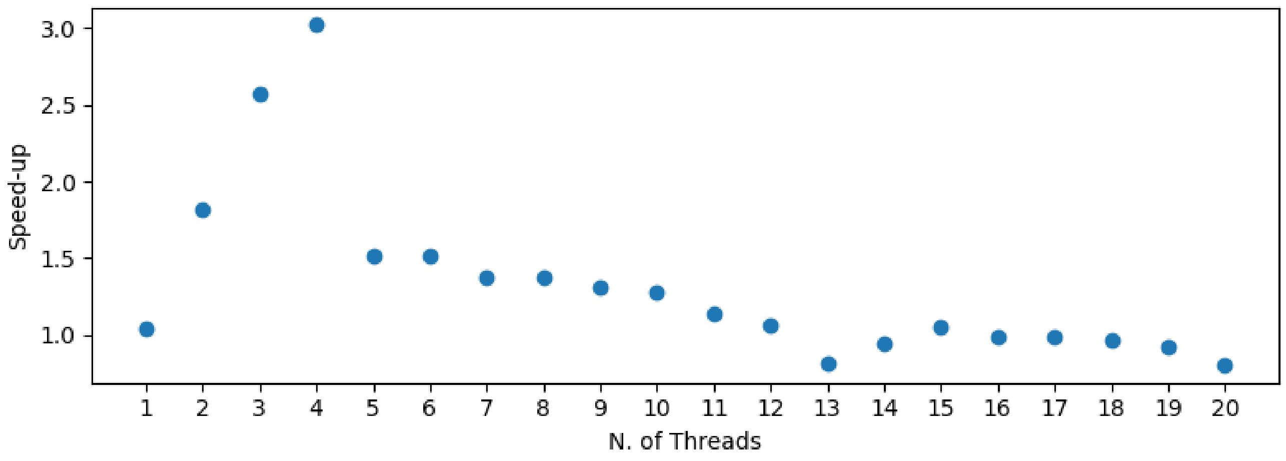

Figure 1. Plot of speed-up values against number of threads displayed in Table 1.