K-means and Mean shift clustering

Fernando Chirici
E-mail address
fernando.chirici@stud.unifi.it

Abstract

In this paper will be presented and analyzed two clustering algorithm: K-means and Mean shift. Specifically, following a small introduction about the characteristics of each algorithm, will be reported the implementation using a specific programming language. K-means has been implemented using Python, that will be at first tested sequentially and then parallelizing using the multiprocessing library. The python implementation will enlight the limits of such programming language to operate in such a way and show, throughth the use of the multiprocessing library, how to relieve those limitations. Mean shift has been implemented using two different programming language to compare the performance of two of the most ideal implementation for such algorithm. At first will be shown the OpenMP implementation, then the CUDA one.
For every implementation will be reported a set of experiments built in order to find the ideal configuration parameters to use these clustering tools on a growing set of input data. In conclusion will be possible to understand limitation ad capacity of each and every algorithm presented.

1. Introduction to K-means

The K-means algorithm, also called *Lloyd algorithm*, is a **clustering algorithm** , i.e. is part of the family of the techniques that aim to partition an input set in one or more output sets named ***clusters***. Specifically, K-means is a non-parametric clustering algorithm so every cluster center (***centroid***) is placed over the most dense region of the featured space. This characteristic make this algorithm inefficient in the presence of elongated clusters or manifolds with irregular shapes. On the opposite, it's simplicity make it really easy to use and efficient with **convex and isotropic clusters.**

Starting with random centroids, the algorithm iteratively assign every point of the feature space to one and only one cluster using as a discriminant the minimum euclidean distance from each centroid. At the following steps the centroids are updated and the assignment is repeated until the algorithm doesn't converge or the prefixed maximum number of iteration is reached. The number of cluster obtained in output is a configuration parameter that need to be set before the algorithm is launched.

The K-means algorithm has a linear computational complexity:

$$C = O(nKt) \tag{1}$$

where **n** is the number of input points, **K** is the number of output clusters and **t** is the number of iterations that the algorithm needs to converge (this value, as already said can be limited above). It's easy to see from it's linearity that K-means lack of scalability. The bottle neck is the time spent for every euclidean distance calculated from and to each point and the following assignment to a specific cluster. This limitation can be partially overcome using parallelization. Searching for the minimum euclidean distance is in fact a completely independent task for each point, so it is possible to run multiple instance of this particular trivial task at the same time.
In the next section will be presented an imple-

mentation of K-means and a set of experiments using both a sequential execution, where every instruction is executed only after the last one has been computed, and a parallel one, characterized by a subdivision of the work on multiple cores of the CPU.

The experiments on the K-means clustering algorithm have been done working in a 2-dimensional space. This choice has been take for reasons of computational limitation of the device that would have forced to limit the number of points to be analyzed and overshadowed the performance of the parallelization.

1.1. Python implementation

To develop the Python implementation and testing the performance of the K-means algorithm has been used Pycharm IDE on a Macbook Pro 2017 with a 3,1 GHz Intel Core i7 quad-core hyperthreading to 8 logical cores.
Starting from the pseudo-codes below a high level python implementation has been implemented both for a sequential and a parallelized execution. In both cases has also been tested the effects of multi-threading on the performance.

For the serial algorithm (Algorithm 1), we only have in input the input-set P of points and the number of desired output clusters K that is the only configuration parameter needed. With that given, K centroid are randomly placed in the featured space. Each point in the feature space is then assigned to the cluster that results as the one with the centroid with the lowest euclidean distance to the point itself. In the last step, the centroids are updated following the new topology. This procedure is repeated until convergence or until the maximum number iteration has been reached.

In the parallel version of this algorithm (Algorithm 2), the difference lies in the subdivision between cores of the work of finding for each point the closest centroid on the featured space. The inputs for this algorithm are the input-set P, the number of output clusters K and the number of cores to work on C. As visible in the pseudo-code below, this operation is executed by firstly dividing the data points in *p* subgroups, and only after working on the euclidean calculations, the subgroups are re-aggregated in one single set in order to be able to update the centroids at the following step.

Algorithm 1: Serial K-means algorithm

```
Data: P, K
Place the centroids c_1^0, c_2^0, .....c_k^0 randomly;
while is not reached convergence or maximum
 iterations do
    for each data point x_i do
        find the nearest centroid(c_1^i, c_2^i..c_k^i);
        assign the point to that cluster;
    end
    for each cluster j = 1..k do
        Update centroids c_i^{i+1} = mean of all points
          assigned to the cluster c_i^i;
    end
end
End;
```

1.1.1 Experiments

The experiments on the K-means clustering algorithm have been constructed to gradually upscale the number of points to be clustered and notice the differences between the various performances. Every execution has been run using the *timeit* library, set up in order to run for every experiment 50 different execution and return the average time to complete it. Note that timeit library is set to use a *performance timer* usually used for benchmarks. The times that will be reported are to be interpreted in time relative to CPU counters and have no connection with the *"real world time"*.

Algorithm 2: Parallel K-means algorithm

Data: P, K, C
Place the centroids $c_1^0, c_2^0, ...c_k^0$ randomly;
Randomly shuffle the data points and partition them into p subgroups;
while *is not reached convergence or maximum iterations* **do**
  for *each subgroup* p_i **do**
    find the nearest centroid($c_1^i, c_2^i..c_k^i$);
    assign the point to that cluster;
  end
  Aggregate the results from each subgroup p to form K clusters;
  for *each cluster j = 1..k* **do**
    Update centroids c_i^{i+1} = mean of all points assigned to the cluster c_i^i;
  end
end
End;

In Figure 1 is shown an execution of the algorithm on different input sized going from 500 to 35000 2D points, in order to obtain 3 output cluster (K). It is very interesting to notice that

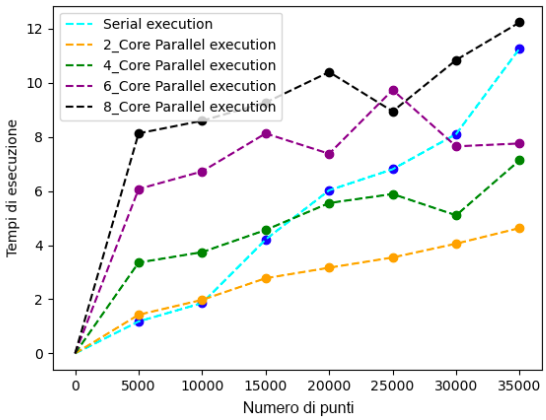

Figure 1. K-means run from 0 to 35k - K=3.

35k points is not yet enough to get from the parallelization over multiple cores the desired positive effect. In fact the cyan line, representing the serial execution of the algorithm is not even the worst among the other, instead that's the 8 core run. This is due to concurrency and core access issues, and the technical time needed in order to do so. Increasing the number of points, this delays no longer weigh so much on the execution. This is shown in Figure 2.

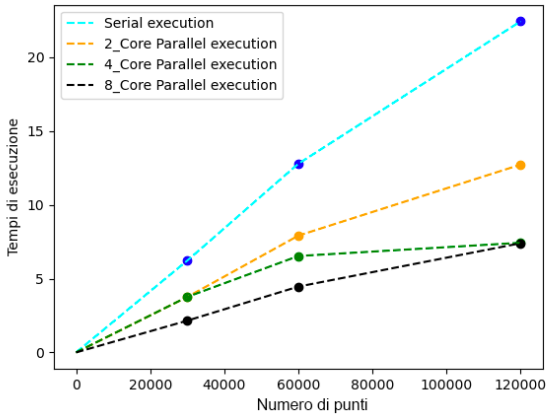

Figure 2. K-means run from 0 to 120k - K=3.

In Table 1 are also reported the speedup gained from the parallelization over multiple cores.

Table 1: K-Means parallel speedup - K=3

Mode	Points	Time	Speedup
Sequential	30000	6.192	—
2 Core	30000	4.298	1.441
4 Core	30000	4.266	1.451
8 Core	**30000**	**1.980**	**3.127**
Sequential	60000	13.434	—
2 Core	60000	7.223	1.860
4 Core	60000	5.991	2.242
8 Core	**60000**	**4.021**	**3.340**
Sequential	120000	23.919	—
2 Core	120000	12.942	1.848
4 Core	120000	6.373	3.753
8 Core	**120000**	**6.225**	**3.842**

Clustering over big amount of data clearly shows a gap between serial and parallel execution. The sequential execution takes an enormous amount of time, almost six times the 8 and 4 core executions, to run the clustering algorithm on 120k points. Still, the limits of K-means and Python multiprocessing performance needs to be noted. Even if the parallel executions

clearly outperforms the serial one, those still needed a lot of hours to terminate every run. The linear nature of the complexity of the algorithm means a linear strong increase on the number of operations needed in order to compute K-means, to which Python is not able to cope with. In fact, even though Python is an interpreted language, it first gets compiled into byte code. This byte code is then interpreted and executed by the Python Virtual Machine. The act of compilation and following execution drastically slows down Python code and doesn't make it a good candidate to develop this kind of algorithms.

Increasing the number of output cluster also increase the number of operations so it's natural to expect worst performance for every kind of run. In the experiment results shown in Figure 3, every data have been left the same but the number of output cluster set to K = 50. As expected the

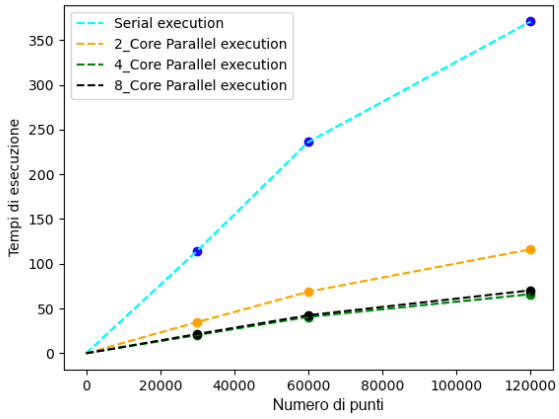

Figure 3. K-means run from 0 to 120k - K=50.

total time of execution for every run drastically increase and the sequential execution moves even further away from the other one. An interesting fact to be notice is the similarity between the 8 and 4 core executions. In fact, in the last example the 4 core actually slightly outperform the other. This issue can be addressed to a lack of capacity from Python multiprocessing to succeed on parallelize on every of the 8 logical cores and only really using the 4 physical ones. The same issue is observable in Figure 4 and seems to be more present when facing a big amount of work. For example, notice that in Figure 2, elaborating only 3 output cluster, the same issue only appeared on the elaboration of the 120k input set. In this last

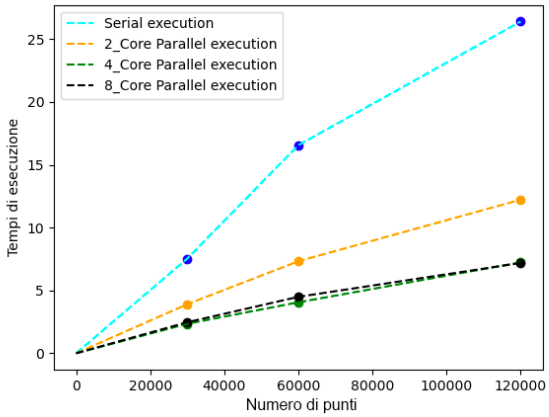

Figure 4. K-means threading run from 0 to 120k - K=3 - Max Workers=100.

experiment the algorithm is again set up in order to generate 3 output cluster on the same amounts of increasing data but instead of multiprocessing, a multithreading approach is performed using the library *ThreadPoolExecutor*. In this scenario almost no limit has been set on the number of workers (set to max 100).
Comparing with Figure 2, it's visible the speedup of the 2 cores execution and the slight improvement of the 4 and 8 cores runs.

1.2. Conclusions on K-means

In this experiments has been possible to observe clearly the speedup gained by running multiple process in parallel to compute due distances calculation of K-means clustering. Starting from around 35k points, the sequential run became the worst case run and from there on it stays like that. The biggest difference between the parallel execution stands between the 2 cores and the other runs. This is both because the first one simply uses less cores then the others in order to compute the algorithm, and also because of the problem discovered about Python's multiprocessing library not being able to compute efficiently using

the 4 remaining hyper-threaded logical cores of the CPU, but only working the 4 physical ones. At last, switching to a multithreading scenario it's been possible to notice a significant speedup for the 2 cores execution and a slight improvement for the 4 and 8 cores runs.

2. Introduction to Mean shift

Mean shift is an unsupervised learning algorithm that is mostly used for **clustering**. It is widely used in real-world data analysis (e.g., image segmentation) because it's non-parametric and doesn't require any predefined shape of the clusters in the feature space. In fact more generally Mean shift is a model for detecting modes and their regions of influence in the feature space based on the Kernel Density Estimation, a technique to estimate the underlying density function of a distribution of probability over a certain dataset. Unlike K-means, this algorithm also does not require to specify the number of cluster to be output, instead the value will depend on the distribution.
The algorithm is structured so that at each step a kernel function is applied to each point in order to shift it in the direction of the local maxima of the kernel function itself. This procedure repeats until every point haven't reached such maxima finally forming a cluster. The kernel function used in this implementation, as it usually happens, is a Gaussian kernel:

$$K(d) = e^{-}(\frac{d^2}{2\sigma^2}) \tag{2}$$

where d is the distance between the center point to any other point and the standard deviation σ is defined as bandwidth so the parameter to adjust how fast the weight decreases with the increasing of the distance. The algorithm is not highly scalable, as it requires multiple nearest neighbor searches during its execution. It will be showed that it's complexity of

$$O(n^2) \tag{3}$$

will highly impact on the sequential execution and that the parameter configuration is critical.

The experiments on the Mean shift clustering algorithm have been done working in a 3-dimensional space. This choice has been take in order to have fewer trivial parameters to confront between the OpenMP and the CUDA implementations.

2.1. C++ and OpenMP implementation

The OpenMP version of the Mean shift clustering algorithm has been naturally developed in C++ using, as already said, the Guassian kernel as kernel function. The Algorithm 3 shows how the implementation has been structured. Given the input set P of 3-dimensional points, the function iterates until maximum iterations isn't reached. Inside the outer loop, the OpenMP directive ***pragma*** is called in order to define parallel region in which work is done by threads in parallel. In this case, the directory is statically bind to the enclosing parallel region. The static binding has been preferred to the dynamic one because of the structure and memory access of the parallelized loop pragma works on. In fact, static schedule means that iterations blocks are mapped statically to the execution threads in a round-robin way. The static flag guarantees that if multiple loops characterized by the same amount of iterations are running using the same number of threads using static scheduling, then each thread will receive exactly the same iteration ranges in every in every parallel region. This is particularly important in NUMA[1] systems because from the first memory access at the first loop, that memory will then reside in the NUMA node where the executing thread is. From the second loop on, the same thread is be able to access the same memory location much faster since it will reside on the NUMA node. Another reason to choose the static scheduling instead of the dynamic one is that the workload on the parallel section in evenly divided by the deterministic evolution of the algorithm.

In the Algorith 4 is reported the implementation of the shifting point function. In order to re-

[1] Non-uniform memory access

Algorithm 3: OpenMP Mean shift implementation

```
Data: P
ShiftedP_set = P;
while is not reached maximum iterations do
    #pragma parallel for schedule(static);
    for each point p in ShiftedPset do
        p = shift_single_point(p, P);
    end
end
End;
```

Algorithm 4: shift_single_point implementation

```
Data: P, p, BW
shiftedP, weight = 0;
for each point x in P do
    d = Squared distance(p, x);
    w = GaussKernel(d,BW) ;
    shiftedP = shiftedP + w ∗ x ;
    weight = weight + w;
end
return shiftedP/weight;
```

turn the shifted value (new value) of a point p of the feature space, the function needs the current value of the point p, the input set P and the bandwidth, a parameter that controls the smoothing of the resulting probability density function (so increasing the bandwidth would generate fewer but bigger clusters). Inside the function, for every point of the featured space, the euclidean distance from the current point p is calculated in order to evaluate it in the kernel function. The updated point is returned weighed according to its new weight.

2.1.1 OpenMP experiments

The experiments were run on the same machine of the K-means ones and in order to achieve interesting results in acceptable times, every test runs the Mean Shift clustering algorithm on different 3-dimensional input sets of sizes [1000; 10000, 35000; 50000] at first sequentially, removing the pragma directive, then using 2, 4, 6 and 8 threads and moving the bandwidth in order to find the best performance.

Table 2: Speedup bandwidth 2

Mode	Time	Speedup	Points
Sequential	7.312	-	1000
2 cores	3.81295	1.918	1000
4 cores	2.27233	3.218	1000
6 cores	2.02205	3.616	1000
8 cores	**1.83168**	**3.992**	**1000**
Sequential	854.922	-	10000
2 cores	402.092	2.126	10000
4 cores	170.191	5.0233	10000
6 cores	119.799	7.136	10000
8 cores	**115.395**	**7.409**	**10000**
Sequential	7829.59	-	350000
2 cores	3276.77	2.389	350000
4 cores	1895.40	4.131	350000
6 cores	1559.57	5.020	350000
8 cores	**1321.15**	**5.926**	**350000**
Sequential	15221.1	-	500000
2 cores	6004.2	2.535	500000
4 cores	3399.5	4.477	500000
6 cores	2786.98	5.461	500000
8 cores	**2466.52**	**6.171**	**500000**

The first experiments shows how the algorithm works with a bandwidth equal to 2. As visible from Table 2 above and Figure 5, the speedup defined by parallelizing on an increasing number of threads is definitely visible. The 8 core version in fact is always the best one, outperform the 2 core one of 1.98s and the sequential one of 5.478s working on the 1000 point set, and keeps the same trend for the bigger sets. In Figure 6 and 7 is visible that the same trend also apply to the experiments using bigger bandwidth, respectively equal to 6 and 14. It's interesting to find the best performance achievable by Mean shift on those input sets. After what's already been showed, it's reasonable to look for the best configuration in 8 core mode of execution.

Table 3: Best bandwidth for input sets

Bandwidth	Time	Points
2	1.83168	1000
6	1.1326	1000
14	**1.05149**	**1000**
2	115.395	10000
6	**110.901**	**10000**
14	115.46	10000
2	**1321.15**	**35000**
6	1343.03	35000
14	1402.42	35000
2	**2766.52**	**50000**
6	3197.641	50000
14	3221.142	50000

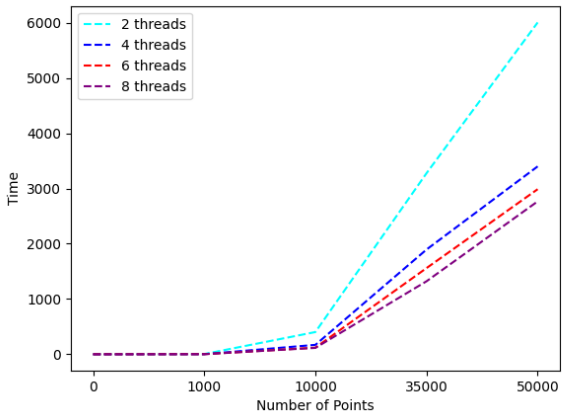

Figure 5. OpenMP Mean shift speedup - BDW=2

It's interest to notice that somewhat counterintuitively the best performance in small dataset is achieved by big bandwidth (14) where in big one is achieved by small bandwidth (2). the reason is probably attributable to the mere starting position of the points.

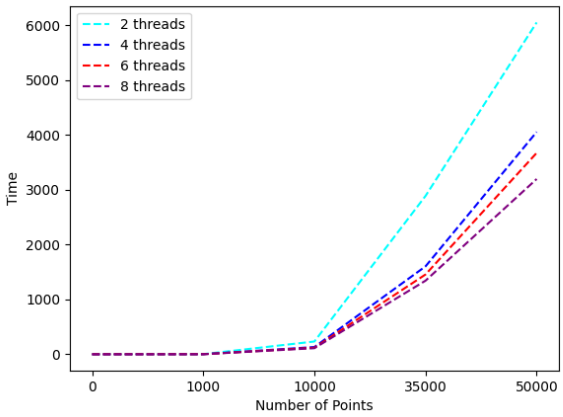

Figure 6. OpenMP Mean shift speedup - BDW=6

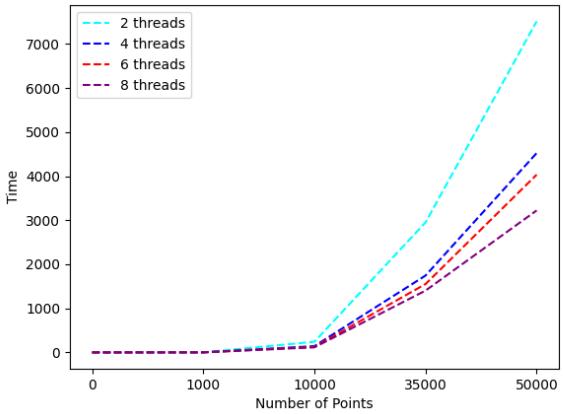

Figure 7. OpenMP Mean shift speedup - BDW=14

2.2. CUDA implementation

The CUDA implementation of Mean shift is naturally similar to the OpenMP one but built in order to exploit as much as possible from the GPU. For the working environment has been chosen *Google Colab* that, in free mode, allow you to run on an NVIDIA Tesla K80 GPU and 12GB of ram.

A CUDA implementation of Mean shift allow to take advantage of a great number of GPU cores. Following what's already been said for the OpenMP implementation, in this case is also possible to assign individual shifting to different cores. The implementation is build in order to

take advantage from the fact that modern DRAM systems are designed to always be accessed in burst mode. In fact the points are stored in memory as a *structure of arrays* in order to work with coalesced memory and to be more suitable to work on L2 cache memory.
In order to only need one coordinate to define the grid and the blocks, the array of points is shape as a 1-dimension array arranged by concatenating the sets x y and z coordinates in fixed blocks of BLOCK_DIM size. This means that the access to the array takes the form:

$$blockDim.x * blockIdx.x + threadIdx.x \quad (4)$$

where $blockDim.x = BLOCK\_DIM$ This shape allows threads to work through consecutive global memory locations and enable the hardware to merge and map every memory access into a consolidated access to consecutive DRAM locations, drastically increasing performances.

In the CUDA implementation, the Mean shift algorithm (Algorithm 6) is implemented as a CUDA kernel that is invoked from the main function (Algorithm 5) in order to accomplish the maximum amount of iterations set during the configuration.
The GRID is defined as a function of the block size as follow:

$$GridDim = [dim3](nPoints/BLOCK\_DIM); \quad (5)$$

Algorithm 5: CUDA Mean shift core implementation

```
Data: P
ShiftedP_set = P;
while is not reached maximum iterations do
    CUDA_mean_shift(ShiftedP_set,P);
end
End;
```

The CUDA mean_shift kernel take in input the set of shifted points, the input set and the bandwidth. At first *idx* is accessed following the instruction reported in (3). Then, before computing the shift of its corresponding point each thread has to check if its index is not greater that the number of points. In this case the thread simply does nothing. This may happen when the number of points is not multiple of BLOCK_DIM, therefore more threads have been instantiated than necessary. Instead, if the condition is match, the algorithm proceed to calculate the shifted point just like showed in the OpenMP implementation.

Algorithm 6: CUDA_mean_shift implementation

```
Data: ShiftedP_set, P, BDW
idx = blockIdx.x * blockDim.x + threadIdx.x;
if idx < |P| then
    shiftedP = 0;
    weight = 0;
    p = shiftedP_set[idx];
    for each point x in originalP_set do
        d = Squared dist(p, x);
        w = GaussKernel(dist,BW);
        shiftedP = shiftedP + w * x weight = weight
          + w
    end
    shiftedP_set[idx] = shiftedP/weight
end
```

2.2.1 CUDA experiments

The CUDA implementation of Mean shift algorithm, as expected, gives much better results than the C++ and OpenMP ones therefore in order to test the performances of such algorithm the tests run over increasing input sets of sizes [100000, 250000, 525000] points, quite bigger then those used to test OpenMP. Apart from the CUDA implementation itself, another contributing factor is that Google Colab offer an ideal development environment for such applications.

In Figure 8 are visible the results of the test executed in order to find the best block size, hence the ideal grid, to run Mean shift clustering on the 100k input set. The ideal block size, marked in red, is 64 allowing the algorithm to run in an average time of 0.465359. Following (4), this block size corresponds to a grid of size 1562.

The same test has been run for the other input sets and the results are visible in Table 4.

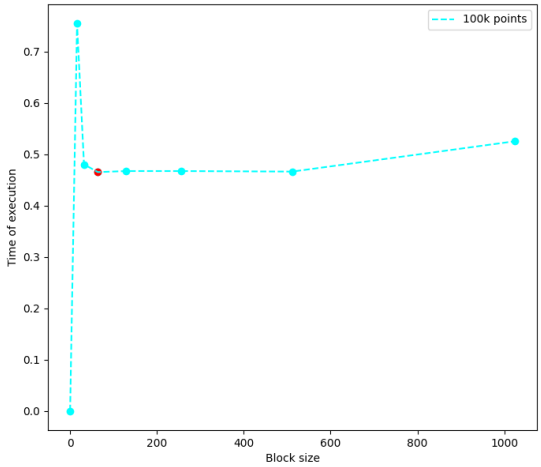

Figure 8. Best block size for 100k input set.

Table 4: Best block size for input sets

Points	Time	Block size	Grid size
100000	0.465359	64	1562
250000	3.064378	128	1953
525000	12.991542	256	2050

Is visible the proportional trend that the dimensions of block and grid keeps following the size of the input set.

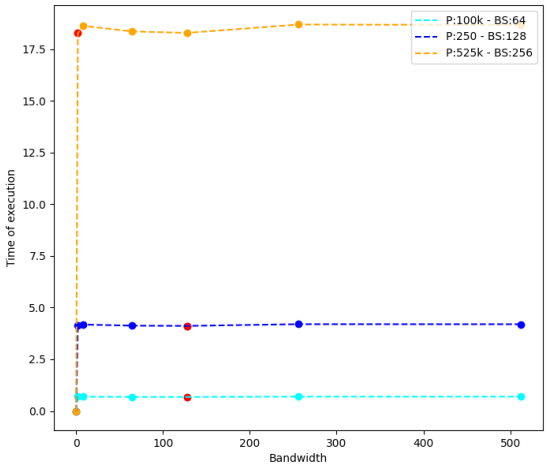

Figure 9. Best bandwidth for every optimal grid.

Established the best setup of the grid to run the algorithm on the chosen input sets, it's interesting to find out how the bandwidth influence the performances.

In Figure 9 is reported a test that shows for every input sets different executions set up with the best block size found as the band varies from 2 to 1024. As visible the changes are not quite relevant and always variate on a range of maximum 1%.

2.3. OpenMP and CUDA comparison

It can be seen in Figure 10 the impressive difference between running the algorithm in OpenMP and CUDA set on their best configuration to run over the 100k input set. Specifically the first one, as discovered in section 2.1.1, is set to run on 8 cores and a bandwidth at 2 while the second one, following the results of the experiments, is set to run with a block size equal to 32 and bandwidth 2. The speedup is obvious and reported in the summary on Table 5.

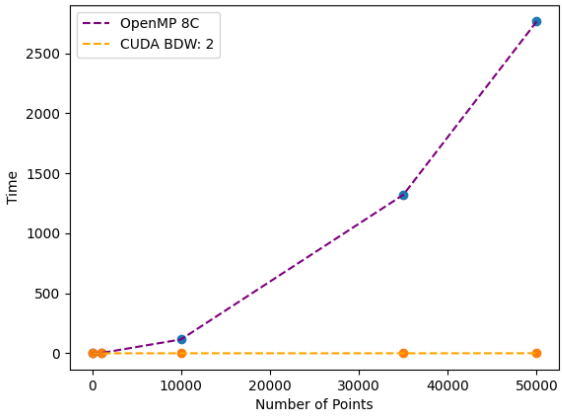

Figure 10. todo

Table 5: Speedup CUDA over OpenMP*

Points	OpenMP	CUDA	Speedup
1000	1.83168	0.002386	767.68
10000	115.395	0.015821	7293.78
35000	1321.15	0.103233	12797.75
50000	2766.52	0.200798	13777.63

Results obtain on different hardware.

3. Conclusions

Starting from the python implementation of K-means, until the CUDA implementation of Mean shift, through this experiments has been possible to see the incredibly positive effects of parallelization on both parametric and non parametric clustering algorithm. For the K-means algorithm, embedding the multiprocessing pool directory allowed the script to gain up to 6.225 speedup. The OpenMP implementation is not only very easy to obtain from a C++ program, just by adding a pragma directive, but also show speedup on the execution up to a factor 7.409. Finally the CUDA implementation, unfortunately forced to be tested on a different environment, show an incredible speedup over the OpenMP version up to a factor 13777.63. This last implementation in fact, running on the Colab GPU doesn't seems to feel big changes even in the configuration parameters, until reaching input sets over 500k 3-dimensional points.

References

[1] H.-H. Bock. Clustering methods: a history of k-means algorithms. *Selected contributions in data analysis and classification*, pages 161–172, 2007.

[2] D. Comaniciu and P. Meer. Mean shift analysis and applications. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1197–1203. IEEE, 1999.

[3] NVIDIA, P. Vingelmann, and F. H. Fitzek. Cuda, release: 10.2.89, 2020.