PC-2019/20 Final Course Project: K Means Algorithm: comparison between CPU and GPU implementation using OMP and CUDA

Lorenzo Pisaneschi
E-mail address
lorenzo.pisaneschi1@stud.unifi.it

## Abstract

*Clustering is one of the most important and used technique for data analysis and k-Means clustering is surely the most used algorithm exploiting this task, due to its simplicity. Clustering goal is to find sub groups of data which has common characteristics inside of an entire dataset that can be formed by a lot of data; k-Means can perform very well also with a large amount of data. Given k centroids, the idea is to assign each point in the dataset to the nearest centroid: since each point can be processed independently of the others, k-Means algorithm can be performed with higher performance using parallel architectures instead of classic CPU approach. In this paper, the k-Means sequential implementation is compared with the OMP and CUDA ones; the aim of this work is to show the improvements which can be reached using parallel programming, mostly GPU programming with CUDA, when the problem has a parallelizable structure and the number of data increases. Finally, the comparison between k-Means and MeanShift clustering techniques is described.*

### 1. Introduction: the k-Means algorithm

K-Means[1] [2] is one of the most used *unsupervised* machine learning algorithms in order to classify a set of data by some patterns or characteristics. Given a dataset, which should be very large depending on the treated problem, the goal is to assign each point to one of the ***k cluster***, where *k* is an input parameter. A cluster is a collection of data points aggregated together because of certain similarities. To assign a point to a cluster during the computation, a ***metric*** must be chosen; the metric measures the distance (i.e. the similarity) among dataset points and the centroids which represent clusters centers of mass. It's possible to choose between a lot of different metrics, depending on which centroids are updated in the last phase of the iteration until one of these stopping criteria is reached: k-Means procedure converges (centroids are no longer modified) or a max number of iterations is exceeded. In other words, after an ***initializing phase***, where *k* clusters are identified with respect to theirs centroids, each point is assigned to the best cluster in terms of similarity; then, there is an ***updating phase*** where new centroids are found calculating means among data points of each centroids averaging points values; the procedure goes until convergence or the maximum number of iterations achievement.

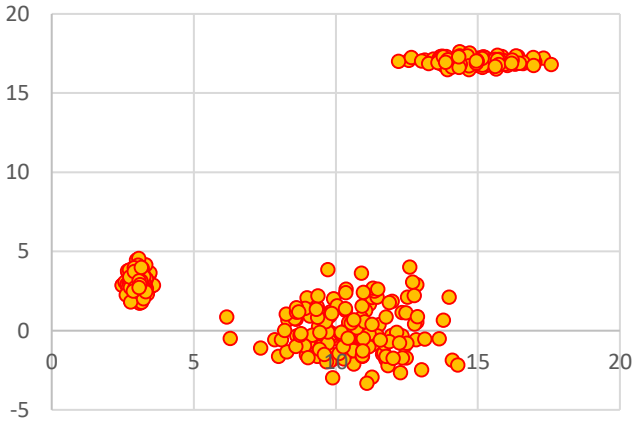


*Figure 1 k-Means clustering result with k = 3*

#### 1.1. Pseudocode

Here a very simple pseudocode is presented. In the code attached to this paper, the metric used is the ***Euclidean distance***, and the parameter ***k*** is set to 3.

```
Algorithm 1: k-Means
procedure   k-Means (inputPoints, clusters,
            maxIterations)
    it ← 0
    isNotConvergence ← false
    while t < maxIterations & isNotConvergence do
        CalculateAllDistances(inputPoints,
        clusters)
        isNotConvergence =
        FindNewCentroids(clusters)
        it ← it + 1
```

Where *inputPoints* are the input dataset, *clusters* represent the data structures with the centroids and the accumulators for the means calculation. The *k* parameter is the clusters size.

***Algorithm 2**: CalculateAllDistances*
***procedure*** *CalculateAllDistances(inputPoints, clusters)*
*minDistance* ← ∞
*clusterIndex* ← *0*
***for all*** $point_i, i = 1,..,|inputPoints|$ ***do***
***for all*** $cluster_j, j = 1,..,|clusters|$ ***do***
*distance =*
*EuclideanDistance*($point_i, cluster_j$)
***if*** *(distance < minDistance)*
*minDistance* ← *distance*
*clusterIndex = j*
***set*** *clusterIndex as* $point_i$ *clusterId*
***accumulate*** $point_i$ ***with***
*accumulateClusterPoints() function*

This procedure is the k-Means first step, where each point is assigned to the nearest cluster with respect to the centroids coordinates, accumulating point values in relative cluster accumulator for the next means calculation using *accumulateClusterPoints()* function. After this, is possible to update the centroids and verify if the algorithm converges.

***Algorithm 3**: FindNewCentroids*
***procedure*** *FindNewCentroids (clusters)*
*isConvergence* ← *false*
***for all*** $cluster_j, j = 1,..,|clusters|$ ***do***
$cluster_j$*.updateCentroid()*
$cluster_j$*.resetAccumulator()*
***return*** *isConvergence*

*updateCentroid()* updates each cluster centroid coordinates, calculating means using the respective accumulator; it returns a boolean value which indicates if there was an updating: if there was not updating, it implies that the algorithm has come to convergence.

### 1.2. Algorithm considerations

Given a set of observations $(x_1, x_2, \ldots, x_n)$ where each observation is *a d-dimensional* real vector (in this case *d = 2*) k-Means clustering aims to partition the *n observations* into *k <<< n* sets $S = \{ S_1, S, \ldots, S_k \}$ so as to **minimize** the whitin-cluster sum of squares (***WCSS***, i.e. variance [3]). The objective function is:

$$\boldsymbol{argmin_S} \sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||^2 \qquad (1)$$

K-Means algorithm is widely used. However, finding the optimal solution is NP-hard in general Euclidean-space. Convergence to local minimum may produce wrong results themselves may vary depending on the initialization. The k-Means algorithm is known also to have a time complexity of $O(knd)$, where n is the dataset dimension: it's computationally expensive. Furthermore, the *k* parameter is crucial: an inappropriate choice may yield to poor results.
As described above, it's clear that during an iteration, for each point three distances are calculated using the chosen metric (here the Euclidean one). Naturally it is really expensive computationally, mostly if the procedure is executed on a CPU; on the other hand, since each point is independently processed of the others, we can image that implementing k-Means in a parallel manner even more so using GPU architectures we can probably boost operations since each point will be processed parallel with the other by a specific thread. In the follow, k-Means is presented in his sequential version and it is compared with its parallel implementations: one is obtained using OMP, an API which supports multi-platform shared memory multiprocessing programming in C++ (the language here used) and CUDA, (Compute Unified Device Architecture), a technology used for GPU programming able to widely improve performance solving parallelizable problems.

## 2. K-Means: sequential and parallel implementations

***Algorithm 4**: k-MeansParallel*
***procedure*** *k-MeansParallel (inputPoints, clusters, maxIterations)*
*it* ← *0*
*isNotConvergence* ← *false*
***while*** *t < maxIterations & isNotConvergence* ***do***
*CalculateAllDistancesParallel(inputPoints, clusters)*
*threadsSynchronize()*
*isNotConvergence =*
*FindNewCentroidsParallel*($clusters$)
*threadsSynchronize()*
*it* ← *it + 1*

In k-Means parallel version is fundamental pay attention in ***sharing*** correctly variables and in ***synchronizing*** threads when they have successfully finished their tasks. The functions that will be parallelized are obviously *CalculateAllDistancesParallel* and *FindnewCentroidsParallel.* In the first, each thread has to know the **dataset dimension** and the **number of clusters** (k): these values will not change during the execution, so once read, they will be unchanged; then, points and cluster size should be shared between threads. On the other hand, since each thread will process one point, it must know in private its ***clusterIndex*** and its ***minDistance*** variables,

which change during computation differently from the others. Carrying on, once the *CalculateAllDistancesParallel* will finish, all threads must synchronize their works and pass to *FindNewCentroidsParallel* procedure where threads have to share the convergence condition *isConvergence*. With these thoughts in mind, it's possible step into implementations details.

### 2.1. OMP k-Means

OMP [4] does not require to reconstruct the code: it's enough to add some *directives* in the sequential code which indicate *parallel regions* and how to share variables during the program execution. When the program finds a parallel region, it will be executed as *main thread*, then other threads will be forked following the fork-join model; indeed, when a thread finishes its work, it waits the others (join). The *main thread* inserts an implicit barrier. The parallel execution mode is **SPMD** (Single Program Multiple Data): tasks are split up and run simultaneously on multiple processors with different input in order to obtain result faster. **Sequential consistency** is fundamental: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. On the other hand, also the **Bernstein's condition** must be respected: processes can execute concurrently only if their data are not used by other threads.

#### 2.1.1. OMP k-Means: directives

`#pragma omp parallel` defines a parallel region: it is the main directive, where it's possible to indicates also variables sharing levels. In this case, as mentioned above, we need something like this:

```
#pragma omp parallel default(shared)
private(minDistance, clusterIndex)
firstprivate(pointsSize, clustersSize)
#pragma omp for schedule(static)
```

Before the external for statement: these directives indicate that:

- `minDistance` and `clusterIndex` are private and allocated for each thread treating one point
- `pointsSize` and `clustersSize` are private for each thread but read and shared once starting computation
- `for schedule(static)` each thread deal with the same amount of work

Furthermore, the minDistance updating is defined as a critical section:

```
#pragma omp critical
if (distance < minDistance) {
  minDistance= distance;
  clusterIndex = j;
  }
```

This construct enforces mutual exclusion evaluating distances and writing the correct *clusterIndex;* it protects access to shared, modifiable data, allowing only one thread to enter it at a given time.

### 2.2. CUDA k-Means

OMP execution depends strictly on CPU cores available. Obviously, OMP k-Means presents a better performance than the sequential one, but not the best. K-Means algorithm implemented using CUDA [5] obtains even better performances. This is because CUDA works with GPU architectures, which have many more cores than the number of CPUs available on a computer. GPU have been designed to do simple operations in parallel using caches to boost RAM throughput. Furthermore, GPU threads are extremely lightweight, without the expensive and slow context switch that effects threading using CPU. In CUDA the **SIMT** model (Single Instruction Multiple Threads) is used. SIMT differs by the SIMD model presented above for the fact that each thread has its own instruction address counter, its register set (so a register counter also) and each thread can have an independent execution path. In other words, the SIMD paradigm is combined with threads using GPU architectures. The SIMT execution model is extremely important for GPGPU programming [6], where GPUs are widely used in boosting computational tasks normally handled by CPUs. Carrying on, is also important highlight the fact that parallelization in CUDA is obtained at compile time. *Threads* are organized into *blocks*, which in turn are organized in *grids*: when a block threads finishes, the control is passed to another block and so on. Threads are executed in group of thirty-two called *warps*. It's clear that hardware is important in CUDA: blocks and grid sizes are decided by the programmer, in relation to the problem. Finally, memory management is crucial: before invoking kernel functions, data must be correctly passed from CPU (host) to GPU (device). The goal is to allocate the largest number of threads so that they can execute in parallel and increase performances.
So, given the k-Means algorithm decide how to set block and grid size is fundamental: in other words, it determines the number of threads which will be allocated. Manage memory and data structures is equally important in order to get the most out of our GPU.

### 2.3. 1 CUDA k-Means: implementation details

In k-Means implementation here presented, the ***block dimension*** `BLOCK_DIM` has been varied from 8 to 64 for visualizing the how the program works depending on the blocks size. The ***grid dimension*** is [n / `BLOCK_DIM`].
The dataset is stored using an *Array of Structure*, an array where each element is a structure of the same type.
Remember that the points dimension is d = 2.

$$X = [\,(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\,] \qquad (2)$$

A general point can be accessed according to this formula:

$$index = (BlockDim * BlockIdx + tx) * k \qquad (3)$$

The threads will usually be arranged in the same shape as the data.
As in the sequential implementation and the OMP one, in CUDA two function have been defined, one for calculation and one for centroids updating: `cuda_kMeans_CalculateDistances` and `cuda_kMeans_updateCentroids;` between these kernel functions, `cudaDeviceSynchronize()` is called in order to synchronize all the operating threads before any other kernel function is called. In the k-Means code there is no problem of divergences.

## 3 Experiments and Results

In this section a comparison between different k-Means algorithm implementations is presented. The aim is to highlight how parallel computing increases performances using OMP and CUDA despite sequential implementations.
Using the OMP version of k-Means, the aim is to show how is possible obtain different performance in time varying the number of threads with `OMP_NUM_THREADS` setting. On the other hand, with the CUDA implementation the goal is to show the power of GPU computing.
K-Means has been executed with different datasets of different sizes; the averaged time obtained from 5 different algorithm execution has been chosen as reference time for each algorithm execution for each dataset.
All the experiments have been conducted on a machine equipped with:

- Intel(R) Core i7-8550U CPU @ 1.80GHz 4 Cores, 8 Threads
- RAM 16 GB DDR4 2400Mhz
- GPU "GeForce GTX 1050" 4GB with CUDA 10.2

Points have a 2D dimension (x, y). Datasets and initial clusters have been generated randomly.

The SpeedUp is the ratio:

$$Speedup = \frac{t_s}{t_p} \qquad (5)$$

where $t_s$ and $t_p$ are sequential time and parallel times.

| Dataset size | CPU Time | OMP Time | Speedup |
|---|---|---|---|
| 1000 | 0,0087537s | 0,0309955s | - |
| 10000 | 0,0714324s | 0,0618166s | **1,898x** |
| 100000 | 0,7779713s | 0,2858521s | **1,155x** |
| 500000 | 3,5961935s | 1,3595931s | **2,645x** |
| 1MLN | 6,7640652s | 2,1600539s | **3,131x** |
| 10MLN | 13,387533s | 3,5475999s | **3,773x** |

*Table 1 Sequential and OMP verision times for k-Means algorithm*

We immediately note that with a dataset composed by 1000 points the sequential version is faster than the one implemented using OMP: this is because threading introduces an overhead due to context switching, which doesn't improve the performance; on the other hand, we can see how the speedup grows increasing the dataset dimension.

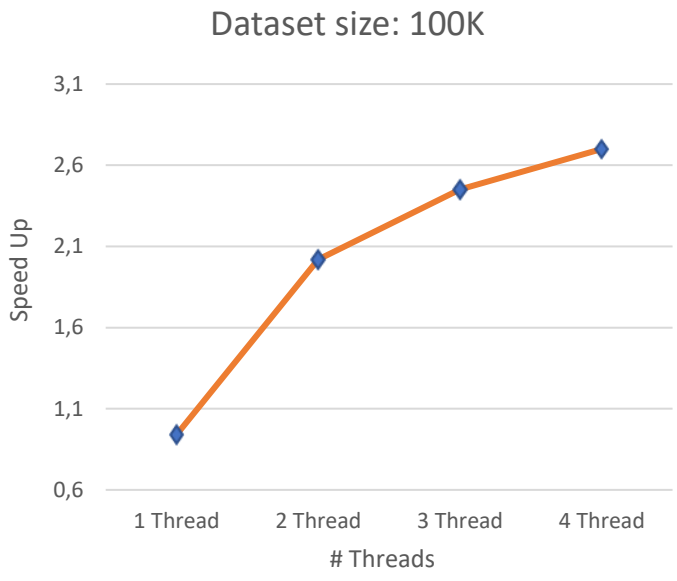


*Chart 1 SpeedUp variation varying threads number in OMP*

In the chart below is shown how speed up increases enhancing threads number. Increasing from 4 threads onwards, performances can get worse due to context switching expected in hyperthreading technology [7].

| Dataset size | CPU Time | CUDA Time | Speedup |
|---|---|---|---|
| 1000 | 0,0087537s | 0,0013126s | **66,721x** |
| 10000 | 0,0714324s | 0,0018166s | **39,464** |
| 100000 | 0,7779713s | 0,0102352s | **76,274x** |
| 500000 | 3,5961935s | 0,0440193s | **81,727x** |
| 1MLN | 6,7640652s | 0,0871926s | **77,657x** |
| 10MLN | 13,387533s | 0,1686117s | **81,408x** |

*Table 2 Sequential and CUDA verision times for k-Means algorithm (block size = 64)*

In *Table 2* results using CUDA are presented. The SpeedUp is considerably better than the OMP one: with CUDA it is possible to work using thousands of single process units called CUDA cores; for the experiments presented above a GTX 1050 has been used: this GPU can rely on:

- 5 CUDA Stream Multiprocessors (SM)
- 128 CUDA cores per SM, so 640 CUDA cores which can work in parallel

In the code presented,

```
dim3 gridDim = dim3(ceil((float)(datasetDim)
/ BLOCK_DIM));
dim3 blockDim = dim3(BLOCK_DIM);
```

means that, for a 100k points dataset, and `BLOCK_DIM = 64` there will be 1562 blocks with 64 threads for each block, so 99968 threads which want execute: since the maximum number per SM is 2048 threads, 10240 will run, the others must wait their turn.
As mentioned above, **occupancy** is very important: threads must work such as possible, keeping busy the GPU at the most.

| Dataset size | BLOCK SIZE = 8 | BLOCK SIZE = 16 | BLOCK SIZE = 32 | BLOCK SIZE = 64 |
|---|---|---|---|---|
| 1000 | 0.0018945s | 0.001713s | 0.0014125s | 0,0013126s |
| 10000 | 0.0048215s | 0.0035225s | 0.0020222s | 0,0018166s |
| 100000 | 0.0390164s | 0.0201506s | 0.0110184s | 0,0102352s |
| 500000 | 0.182285s | 0.0960688s | 0.0494925s | 0,0440193s |
| 1MLN | 0.359822s | 0.182848s | 0.0973374s | 0,0871926s |
| 10MLN | 3.018571s | 1.430911s | 0.9817622s | 0,1686117s |

*Table 3 Different times using different block sizes: lower the occupancy is lower will be performances*

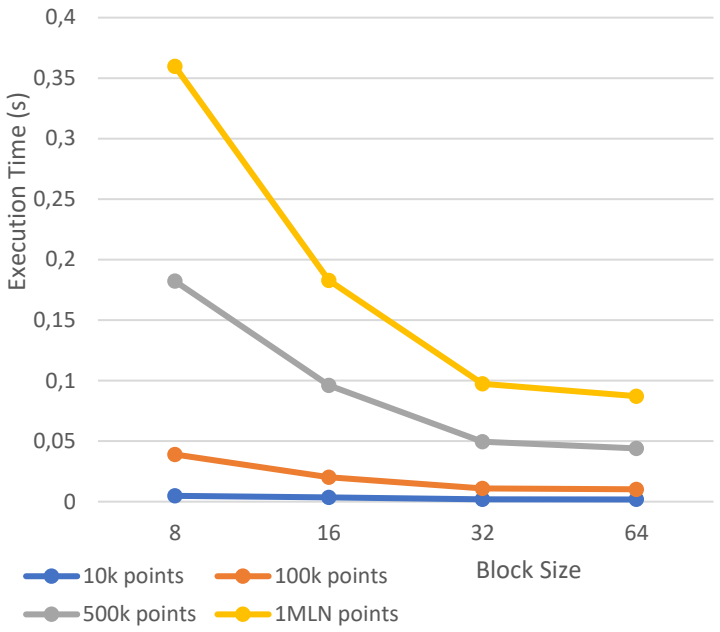


*Chart 2 Times vary varying block size because of occupancy using*

In chart 2 is possible to visualize how execution times decrease when block size increases up to 64 because of a better GPU exploitation.

**4. Discussion**

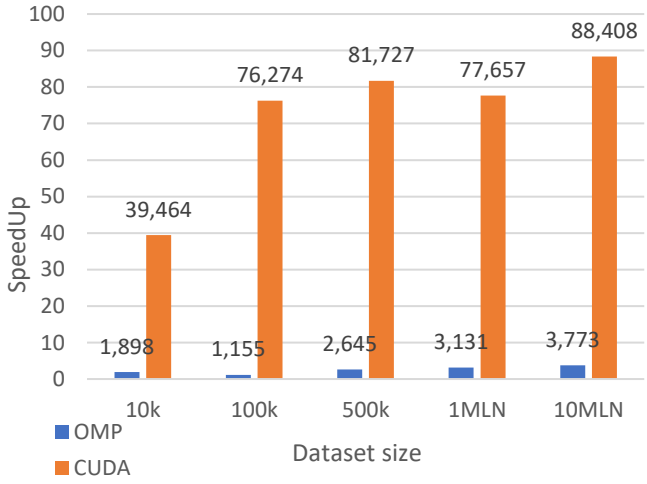


*Chart 3 Comparison between CUDA and OMP speedups obtained implementing the k-Means algorithm*

K-Means is an algorithm which fits perfectly parallel programming due to the fact that each point can easily processed independently by the others; parallel programming can exploited with different paradigm such as OMP and CUDA, two different technologies that use different resources: OMP uses multicore and

hyperthreading made available by modern multicore processors, simply introducing specific directives in sequential code; on the other hand, CUDA uses the power of GPU architectures, that are perfect for problems like k-Means, where iterations process independent element which then will be synchronized together in some computation. Chart 3 shows perfectly how CUDA powerful can boost performance for problems like the k-Means algorithm.

### 4.1. K-Means applications

K-Means is a clustering algorithm and as it is, it's a widely used technique in a lot of different scenarios. In general, clustering algorithm can retrieve information from dataset getting meaningful intuition of the structure of the data we're dealing with: for this reason, k-Means can be used in medical field, e-commerce, advertising since is possible obtain customers clusters and build recommendation networks. On the other hand, cluster algorithm can be used in image processing tasks, like image segmentation.

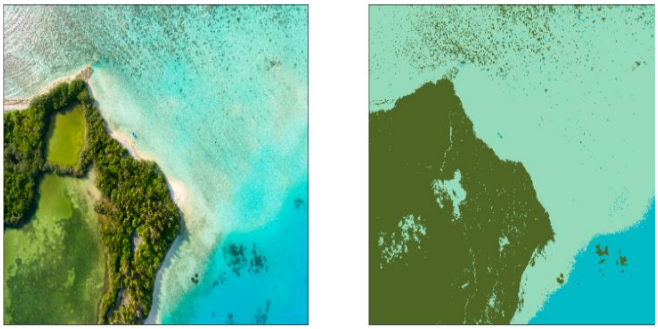

*Fig 1 Image segmentation with k-Means, k = 3*

### 4.2. Clustering algorithm: k-Means and MeanShift

In a related work [] another clustering algorithm has been implemented, the MeanShift algorithm. As described in the attached paper, the MeanShift algorithm is a computationally expensive clustering technique, heavier than k-Means algorithm: it is $O(n^2)$ while k-Means is $O(knd)$. On the other hand, the MeanShift algorithm has only one parameter in input, the bandwidth size: it is able to detect autonomously the k clusters number, unlike the k-Means which wants k as an input information; however, in real applications, it's difficult that we knows how many clusters we have to find, while the bandwidth value can be deducted from some data problem information.
In this scenario, it's clear that k-Means and MeanShift uses different clustering approaches (beyond means calculation) which may take advantages related to different kind of problems. From the implementation point of view, and obviously the parallel programming one, in both cases parallel approaches benefits are evident over the algorithms sequential version, even if k-Means max speedup value (76,274x) is greater than the one registered in MeanShift shared memory CUDA version (5,333x) with the same dataset (100k points size). It could be a good idea use a hybrid clustering algorithm, combining k-Means and Meanshift in order to take advantage of benefits provided by both the algorithms.

## 5. Conclusion

In this paper has been presented the k-Means clustering, a widely used unsupervised machine learning algorithm which can take information from great dimension dataset: its peculiar benefit is its low computational cost to be such an algorithm and for this k-Means is still one of the most used procedure. Then, it has been noticed that k-Means is naturally parallelizable due to the fact that is process a dataset element at a time, calculating distances between points and centroids; so, other two different k-Means clustering implementations have been presented: one written with OMP directives and another one implemented on GPU using CUDA. It has been presented an accurate analysis between these two implementations, finding that the CUDA one is the fastest. Finally, after a brief discussion about possible real-world applications, a confrontation between k-Means and MeanShift clustering algorithms was proposed, highlighting both benefits and defects.

## Resources

Code is available on Github:

- Sequential and OMP version: https://github.com/pisalore/kMeans_OMP
- Parallel version with CUDA: https://github.com/pisalore/kMeans_CUDA

## References


[1] Lloyd, Stuart P. "Least squares quantization in PCM." Information Theory, IEEE Transactions on 28.2 (1982): 129-137.

[2] MacQueen, James. "Some methods for classification and analysis of multivariate observations." Proceedings of the fifth Berkeley symposium on mathematical statistics and probability. Vol. 1. No. 14. 1967.

[3] Wikipedia k-means clustering. https://en.wikipedia.org/wiki/K-means_clustering. [Online; accessed 6-01-p2020].

[4] Thomas Dauber and Gudula Rünger, Springer. " Parallel Programming for Multicore and Cluster Systems", - Chapt. 6

[5] NVIDIA. Cuda documentation. https://docs. nvidia.com/cuda/, 2019. [Online; accessed 16-012019].

[6] D. B. Kirk and W-M. W. Hwu, Morgan Kaufman. "Programming Massively Parallel Processors: A Hands-on Approach"- 2nd edition - Chapt. 1-2-5

[7] D.T. Marr, F. Binus, D.L. Hill, G. Hinton, D.A. Konfaty, J.A. Miller, and M. Upton. Hyperthreading technology architecture and microarchitecture. Intel Technology Journal, 6(1): 4–15, 2002.