*Proceedings of the 6th World Congress on Electrical Engineering and Computer Systems and Sciences (EECSS'20)*
*~~Prague, Czech Republic~~ Virtual Conference – August, 2020*
*Paper No. CIST 103*
*DOI: 10.11159/cist20.103*

# Accelerating Clustering: An Architectural Approach

**Mihaela Malița[1], Gheorghe M. Ștefan[2]**
[1]Smith College
Northampton, MA
mmalita@smith.edu; gheorghe.stefan@upb.ro
[2]Politehnica University of Bucharest
Bucharest, Romania

***Abstract*** **–** The iterative nature of the clustering algorithms demands support for acceleration, not to mention that some of them have the time complexity of each iteration in O($n^2$) or even in O($n^3$). Six of the most frequently used clustering algorithms are investigated from the point of view as how they can be executed on a parallel accelerator. The resulting computational patterns are used to define the architecture of a *p*-cell accelerator as part of a heterogeneous computing system.  For all the investigated algorithms it has been proved that their computational part is accelerated a number of times in O(p).

***Keywords*****:** Clustering, Heterogeneous computing, Artificial intelligence, Accelerators, Parallel computation.

## 1.  Introduction

Clustering is an unsupervised learning technique. It deals with finding a structure in a collection of objects. Thus, a cluster is a sub-collection of objects which are *similar* between them and are *dissimilar* to the objects belonging to other clusters. The main types of clustering are:

- distance-based clustering: the objects belong to the same cluster if they are *close* according to a given (usually geometrical) distance; according to [1] the most used are:
    - k-Means
    - Mean Shift Clustering: is a sliding-window-based algorithm that attempts to find dense areas of data points
    - Density-Based Spatial Clustering
    - Expectation-Maximization Clustering using Gaussian Mixture Models
    - Agglomerative Hierarchical Clustering
- conceptual clustering: objects are grouped according to their match to descriptive concepts.

All these algorithms are iterative. The execution time depends on the actual data provided as input. What can be accelerated is the execution time for each iteration. This time varies from values in O(n) to values in O($n^3$).
In the second part of the paper the previous six algorithm are described in order to provide insights for defining the architecture of an accelerating engine. In the next section we propose a MapReduce accelerator and the algorithms are analyzed in the context offered by this cellular structure. The fourth section compares the performance of our solution with results already published.

## 2. Clustering Algorithms

For clustering algorithms, data is stored in the system memory as a sequence of vectors $X = \{x_1, x_2, ..., x_n\}$, where the vectors are of the form $x_i = [c_i^1, c_i^2, ..., c_i^d]$. Each vector contains the coordinates of a point in a *d*-dimension space and sometimes information used to identify each point.

In this section the algorithms are described independent by the way they are implemented, for a mono-core computing system or for a many-core accelerator. In the next section their parallel implementation will be investigated.

### 2.1. k-Means Clustering Algorithm (KMC)

k-Means clustering provides the partition of the set of points represented by *X* into *k* sets. The value of *k* is established by the user. The algorithm is presenter in Fig. 1.

```
1. Set randomly k d-dimension centers and allocate each point randomly to a center.
2. Compute for each d-dimension point the Euclidean distance to the k centers and
   assign each point to the "nearest" center.
3. Compare the new assignments to the previous one
       if (no difference) then stop the process else continue
4. Move the k centers to the means of the newly created groups and goto 2.
```

Fig. 1: k-Means distance-based algorithm.

The execution time of the main loop $\in O(n)$, but the actual time depends on how many iterations are performed until the centers' distribution stabilize.

## 2.2. Mean Shift Clustering Algorithm (MSC)

The efficiency of the k-Means algorithm depends on the value of k and the initial position of the centers. Mean Shift Clustering algorithm helps us to determine for the input data set the optimal number of centers and their position. We start with a number of centers uniformly distributed in the *d*-dimension space and iteratively these centers move toward the most probable number of centers and their position. The algorithm is presented in Fig. 2.

1. In the d-dimension space, where a set of points are distributed, we begin with sliding *d*-dimension window volumes with radius *r* centered uniformly over the space occupied by the set of points.
2. The sliding window is shifted at every iteration towards regions of higher density by shifting the center point to the mean of the points within the window. Thus, the windows will gradually move towards areas of higher point density until no longer the number of points in the window increases.
3. The sliding windows will continue shifting according to the mean until there is no direction at which a shift can accommodate more points inside a kernel.
4. Then multiple sliding windows overlap in few points representing the centers for applying the k-Means algorithm in one step.

Fig. 2: Mean-shift clustering algorithm.

The execution time of the main loop $\in O(n)$, but the actual time depends on how many iterations are performed until the centers' distribution stabilize.

## 2.3. Density-Based Spatial Clustering Algorithm (DBSC)

When the clusters are not distributed radially the algorithm must follow the higher density path. Then, we must define a radius *r* for the space in which high density is defined by *minPoints*. The algorithm is presented in Fig. 3.

```
1. Start the search for a new cluster with an arbitrary point which is unvisited.
2. Set the selected point as visited. Its neighborhood is identified using a radius r.
3. if within this neighborhood there are at least minPoints:
       then the point and its companions from the neighborhood are added to the current cluster.
4. while in the searched cluster there are unvisited points, select one of them and go to step 2.
5. while there are unvisited points, go to step 1..
```

Fig. 3: Density-based spatial clustering algorithm.

The execution time $\in O(n^2)$. The price of the possibility to identify clusters having any kind of shape, not only radially distributed, is the square time complexity instead of a linear time complexity.

## 2.4. Expectation-Maximization Clustering Using Gaussian Mixture Models Algorithm (EMC)

The idea of Gaussian Mixture Models is to find the parameters of *k* Gaussians that best explain our data [2]. We assume that data are Gaussians and we have to find parameters that maximize the probability of observing these data: we regard each point as being generated by a mixture of *k* Gaussians and must compute the probability to belong to each of them.

1. Select the number of clusters and randomly initialize the Gaussian distribution parameters for each of them.
2. Compute the probability that each data point belongs to a particular cluster.
3. Based on these probabilities, compute a new set of parameters for the Gaussian distributions such that we maximize the probabilities of data points within the clusters.
4. Steps 2 and 3 are repeated iteratively until convergence, i.e., the distributions don't change much from iteration to iteration.

Fig. 4: Expectation-maximization clustering using Gaussian mixture models algorithm.

The execution time of each iteration $\in O(n)$.

## 2.5. Agglomerative Hierarchical Clustering Algorithm (AHC)

The Agglomerative Hierarchical Clustering algorithm considers initially each point in the input data as a single cluster and then successively merges (or agglomerates) pairs of clusters until all clusters have been merged into a single cluster that contains all data points. The hierarchical clustering does not require the specification of the number of clusters because at the end we build a binary tree.

1. We start with each data point as a single cluster.
2. On each iteration a distance metric, that measures the distance between two clusters, is used to combine two clusters with the smallest linkage into one.
3. Step 2 is repeated until we only have one cluster which contains all data points.

Fig. 5: Agglomerative hierarchical clustering algorithm.

.

The execution time $\in O(n^3)$.

## 2.6. Conceptual Clustering Algorithm (CC)

Instead of numerical evaluation of the “distance”, now associative mechanisms must be used to evaluate the “distance” of each point from each center. A numerical evaluation is substituted with a fitting mechanism. Instead of the Euclidean distance the Hamming distance is considered. Each object has a set of *m* attributes coded as an *m*-bit number. One bit for each attribute. One solution is a hierarchical approach which uses a recursive bi-partitioning ***spectral clustering algorithm***. The bi-partitioning clustering algorithm is presented in Fig. 6.

1. Generate the similarity matrix S based on the Hamming distance between the objects.
2. Compute the associated Laplacian matrix using the degree matrix D of S.
3. Compute iteratively the eigenvector of the Laplacian matrix in order to emphasize the bi-partition.

Fig. 6: Conceptual clustering algorithm.

.

The execution time of each iteration for computing the eigenvector $\in O(n^2)$.

## 3. Parallel Approach for Clustering

The previously described algorithms are parallelized starting from the fact that they have the following main specific characteristics:

- a given function is ***applied-to-all*** elements of a sequence of data returning a sequence, i.e., a function is **map**ped to a sequence of data
- a given associative and commutative function uses as argument all the elements of a sequence returning a scalar, i.e., a **reduction** function is applied to the sequence

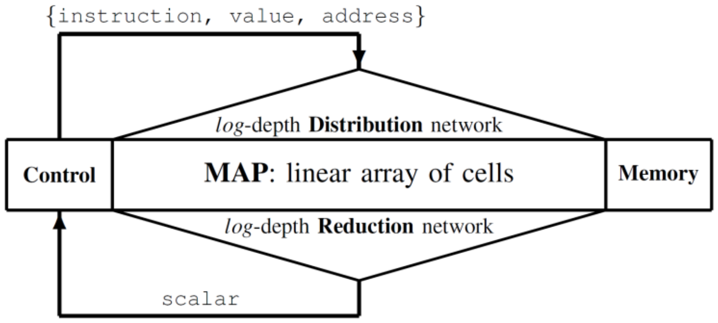
{instruction, value, address}
log-depth Distribution network
Control
MAP: linear array of cells
Memory
log-depth Reduction network
scalar

Fig. 7. Programmable MapReduce accelerator as a linear array of execution units (MAP) loop connected with a controller through a Reduction function.

These two types of functions are applied under the control of a mono-core controller on sequences of data represented as vectors mapped on a linear array of cells. Therefore, the hardware necessary for the previously emphasized functions consists of an accelerator with a linear array of *p* cells loop connected with a controller through a reduction network (see Fig. 7) [3] [4] [5]. The Control unit issues in each clock cycle `{instruction, value, address}`, through a *log*-depth distribution network, to be executed in each active cell of the linear array. Each cell, $cell_i$, consists of an execution unit and a local data memory, $mem_i$, of *m* scalars. Thus, the memory distributed along the cells is seen as the $m \times p$ matrix **M** whose element $s_{ij}$ is the *i*-th scalar stored in the memory of the *j*-th cell.

Along the cells is distributed a Boolean vector, $B=[b_1,b_2,\dots,b_p]$, used to specify the active cells. Each $b_j$ evolves according to the internal state of the *j*-the cell and the instruction received from Control. The operations, defined on, the so called, horizontal vectors $V_i = [s_{i1},s_{i2},\dots,s_{ip}] \in$ **M**, for $i = 1, 2, \dots, m$, are ***predicated map*** operations:

$$V_i \;<=\; OP(V_k,V_q) ::= s_{ij} \;<=\; b_j \;?\; OP(s_{kj},\, s_{qj}) \;:\; s_{ij}$$
$$V_i \;<=\; OP(V_k,s) ::= s_{ij} \;<=\; b_j \;?\; OP(s_{kj},\, s) \;:\; s_{ij}$$

***predicated reduction*** operations:

$$s \;<=\; redOP(Vi) ::= s \;<=\; OP^{p}_{p=1}\; b_j \;?\; s_{ij} \;:\; NAN$$

or ***spatial control*** operations (used to select the active cells according to a condition):

$$\textbf{WHERE}(cond) ::= b_j \;<=\; cond_i \;?\; 1 \;:\; 0$$

for $j = 1, 2, \dots, p$, where `cond` is a Boolean vector, $s$ is a scalar received by the controller Control.

To minimize the overhead introduced by “von Neumann bottleneck”, the data transfer between the array of cells and Memory is transparent to the computational process.

In the following is detailed the k-Means implementation in order to exemplify the use of our proposal. The other algorithms will be discussed related to this algorithm.

## 3.1. k-Means Clustering Implementation

In the external memory, Memory (see Fig. 7), there is stored initially a sequence of *n d*-component vectors containing the coordinates of the *n* points submitted to the clustering algorithm. The algorithm returns a vector *K* containing indexes associated to the *k* clusters, so each point *i* is associated to one of the clusters, and *d* vectors, one for each coordinate of the clusters identified by the algorithm. Let us take, for example, the pseudocode for $d = 3$, $n \leq p$ (see Fig. 8).

```
Initial data in Memory
   S = [[$x_1$,$y_1$,$z_1$],…,[ $x_n$,$y_n$,$z_n$]] // sequence of coordinate vectors in a 3D space
Final result in Memory
   K = [$k_1$,… $k_n$] // sequence of clusters' indexes associated to the points
  x = [$x^1$,…,$x^k$] // x coordinates of the centers of the k cluster
   y = [$y^1$,…,$y^k$] // y coordinates of the centers of the k cluster
   z = [$z^1$,…,$z^k$] // z coordinates of the centers of the k cluster

// Initialization
   K'<= [$k_1$,…,$k_n$] // K'is randomly initialized in MAP array with $k_i \in [1,d]$, for i = 1,…,n
   x <= [$x^1$,…,$x^k$] // x is randomly initialized in Controller with the $x^j$ coordinates of the k centers
   y <= [$y^1$,…,$y^k$] // y is randomly initialized in Controller with the $y^j$ coordinates of the k centers
   z <= [$z^1$,…,$z^k$] // z is randomly initialized in Controller with the $z^j$ coordinates of the k centers
   D <= [$d_1$,…,$d_n$] // is initialized with $d_i$ = "∞", for i = 1,…,n; "each point is far away from any center"
   K <= [0,…,0]
// Data transfer from Memory to MAP array (in time ∈ O(n))
   X <= [[$x_1$,$y_1$,$z_1$],…,[$x_{\lfloor n/3\rfloor}$,$y_{\lfloor n/3\rfloor}$,$z_{\lfloor n/3\rfloor}$]]
   Y <= [[$x_{\lfloor n/3\rfloor+1}$,$y_{\lfloor n/3\rfloor+1}$,$z_{\lfloor n/3\rfloor+1}$],…, [$x_{2\times\lfloor n/3\rfloor}$,$y_{2\times\lfloor n/3\rfloor}$,$z_{2\times\lfloor n/3\rfloor}$]]
   Z <= [[$x_{2\times\lfloor n/3\rfloor+1}$,$y_{2\times\lfloor n/3\rfloor+1}$,$z_{2\times\lfloor n/3\rfloor+1}$],…, [$x_{3\times\lfloor n/3\rfloor}$,$y_{3\times\lfloor n/3\rfloor}$,$z_{3\times\lfloor n/3\rfloor}$]]
// Computing the final values for K, x, y, z
   transpose(X,Y,Z;3×3)    // the $\lfloor n/3\rfloor$ matrices stored in the vectors X,Y,Z are transposed (in time ∈ O($d^2$))
                           // X = [$x_1$,…,$x_n$]: vector of the coordinates x of each point
      // Y = [$y_1$,…,$y_n$]: vector of the coordinates y of each point
      // Z = [$z_1$,…,$z_n$]: vector of the coordinates z of each point
      WHILE (K ≠ K') // test the end of process (in time ∈ O(log p))
         K <= K'
      FOR (i = 1; i ≤ k; i = i+1)      // update K'(in time ∈ O(k×d))
               D'<= $(X-x^i)^2$
               D'<= D'+$(Y-y^i)^2$
               D'<= D'+$(Z-z^i)^2$
               where (D'< D)
                     D <= D'
                     K' <= I
         FOR (i = 1; i ≤ k; i = i+1)   // update x, y, z (in time ∈ O(k×d + log p) ≈ O(k×d))
               WHERE (K' = i)
                     $x^i$ <= redAdd(X)/redAdd(B)
                     $y^i$ <= redAdd(Y)/redAdd(B)
                     $z^i$ <= redAdd(Z)/redAdd(B)
// Transfer the result from MAP to Memory (in time ∈ O(n))
   Memory <= K
   Memory <= [x,y,z]
```

Fig. 8: k-Means clustering implementation for $d = 3$, $n \leq p$

.

The execution time for this clustering algorithm depends on the actual distribution of the input data, S. Therefore, only the acceleration provided by the `while` loop can be evaluated, and compared with other implementations, because *how many times* the loop is executed cannot be taken into account. The mono-core execution time for this loop is $\approx 40\times k\times d\times n$ clock cycles, while the execution time for the parallel system we propose is $\approx 20\times k\times d$ clock cycles.

The acceleration of ***2n×*** provided by our approach is supra-linear because, besides the parallelism introduced at the level of MAP array, there is a parallelism introduced by the parallelism between MAP, Control and Reduce.

In a mono-core execution, for example, the control and the computation are performed by the same engine, while in our accelerator most of computation is performed in MAP and the control is performed in parallel by the Control module.

Unfortunately, the overall acceleration of the clustering computation must take into account also the transfer time of data between Memory and MAP which is in *O(n)*. If the number of executions of the **`while`** loop is big, then the transfer time has a small weight in the total time, and the value of the acceleration is dominated by computation. But, if the number of executions of the **`while`** loop is small, then the execution time could be dominated by the transfer time. Fortunately, we use this algorithm only when the computation is very intense, when the loop is executed many times and the overall execution time is dominated by computation.

## 3.2. Mean Shift Clustering Implementation

In contrast to k-Means clustering, there is no need to select the number $k$ of clusters. The algorithm is conceived to automatically discover the value of $k$. That's an important advantage. The drawback is that the selection of the window radius $r$ can be non-trivial. The mechanisms involved in running this algorithm on our proposed accelerator are similar with those used for k-Means. The main difference is given by the biggest number of initial clusters and by the way we associate a point to a cluster which is based on the radius $r$. Therefore, the acceleration and the limitation introduced by data transfer are similar with those of the implementation of the k-Means algorithm.

## 3.3. Density-Based Spatial Clustering Implementation

Density-Based algorithm is similar with k-Means and Mean Shift clustering, but has some advantages. Does not require a pre-determined number of clusters. The algorithm determines the outlier points and ignores them as noise, rather than including them in the most probable cluster. And, it can identify arbitrarily sized and shaped clusters quite well.

Because for each point the size of the neighborhood must be evaluated the latency of the Reduction network limits the acceleration to a value in $O(p/\log p)$. Future work on algorithm implementation must be done to "remove" the logarithm.

## 3.4. Expectation-Maximization Clustering Using Gaussian Mixture Models Implementation

Step 2 is computed in a pure SIMD mode in constant time. Because step 3 computes, using reduction the sum of probabilities associated to each Gaussian, the execution time is *(constant + log p)*. Each iteration (steps 2 and 3) is executed in *(constant + log p)* time. Because in the foreseeable future the silicon technology will offer us the possibility to implement MAP with *log p* no bigger than 20, we can approximate *(constant + log p)* with *constant*.

Therefore, the acceleration provided for the computational part of this algorithm $\in O(p)$.
The "von Neumann bottleneck" effect on the data transfer remains and affects the overall performance depending on the number of iterations requested by the actual data input.

## 3.5. Agglomerative Hierarchical Clustering Implementation

Each of the $n$ point comes, besides the coordinates, with a name, as follows:

```
[name, coordinate1, …, coordinateD]
```

The result is a number of $n-1$ clusters each characterized by

```
[name, leftName, rightName, coordinate1, …, coordinateD]
```

For the same reasons as for the previous algorithm, the acceleration of the computation by our approach $\in O(p)$.

## 3.6. Conceptual Clustering Implementation

For $n \leq p$, the $n \times n$ similarity matrix and the associated Laplacian are computed by a system with our accelerator in time belonging to $O(n)$. The eigenvector is computed iteratively, each iteration in time $\in O(n)$, because the hardware is featured with a mechanism to avoid, for this operation and similar ones, the *log* latency introduced by the reduction network. Therefore, the acceleration belongs to $O(p)$.

### 3.7. Conclusions for the Clustering Implementation on MapReduce Architecture

The performance improvement provided by our MapReduce architecture depends of the relation between the time complexity of the data transfer from/to Memory (which is not improved using our approach because the "von Neumann Bottleneck" does not forgive anyone) and the time complexity of the computation. The time associated for data transfer belongs, for all the algorithms and all the current solutions – our included –, to $O(n)$.

Therefore, we can emphasize two situations:

1. For the iterative algorithms with the mono-core execution time of each iteration in $O(n)$ – KMC, MSC, EMC – the overall acceleration depends on the number of iterations performed until the convergence is obtained. If the number is big (in the case when the parallel approach makes sense), then the transfer time has a small weight and the acceleration is in $O(p)$, else the efficiency of using the accelerator is diminished.
2. For DBSC and CC, with the mono-core execution time in $O(n^2)$, and the AHC, with the mono-core execution time in $O(n^3)$, the data transfer time does not affect the magnitude order of acceleration which $\in O(p)$.

## 4. State of the Art

To provide a meaningful comparison concerning the acceleration of the computation, let us take algorithms which are not I/O bounded: AHC and CC.

The AHC algorithm, in our accelerator, computes the hierarchical clustering in time $\in O(n^2)$ with a $n$-cell accelerator with an acceleration belonging to $O(p)$. In [6] hierarchical clustering is accelerated 11.5x to 13.2x, depending on the number of points, using as accelerator GPUs with thousands of cores. The reference is the CPU time provided by an engine with tenths of cores. We expect to have an acceleration in the range of 100x. Where is lost around 10x boost? The answer is provided in [7], where concerning the reduction function is claimed: "is much more trickier to parallelize since the centroid computations depend on all of the other data points in its cluster". Indeed, the expected performance improvement from 100x cells is only in the range of 10x because the overhead introduced by the reduction addition in a GPU is in the range of $log_2$ from thousands, i.e., around 10x.

The core of the CC algorithm performs matrix-vector multiplications. In [8] are published benchmarks showing, for linear algebra computations, a mean value of acceleration ~10× (maximum 36×) for a GPU compared with a CPU. Again, the acceleration is far from the ratio between the execution units involved in comparison. (More detailed criticisms about the current parallel approach see in [9].)

## 5. Our Implementation

The structure of the MapReduce accelerator was implemented in few versions. The last one was implemented in 65nm and it provided: *>120 GOPS/Watt* and *>6.25 GOPS/mm*$^2$ (GOPS stands for 16-bit Giga Integer Operations per Second).

The last evaluation for 28 nm technology provided, for *2048 32*-bit cells, running at $f_{clock} = 1GHz$, with *4096 KB* of memory each, implemented on $9.2 \times 9.2\ mm^2$, using standard cell *28nm* library, **2 TOPS** or **0.83 TFLOPS** powered at **~15 Watt**. These figures translate in more than 50 GFLOPS/Watt.

NVidia's Kepler architecture implemented in *28nm* provides less than 10 GFLOPS/Watt [10].

The area and energy used in our approach is improved:

(1) because the floating-point operations are implemented as a sequence of integer operations (on an average of 8 cycle per float operation), and

(2) because the predictability of the data flow in intense computation is very high, instead of a cache-based memory hierarchy we adopted a buffer-based memory hierarchy.

The MapReduce accelerator can be integrated in various ways in a heterogeneous computing system. One way is to use the FPGA approach in pseudo-reconfigurable computing systems. "Pseudo" because the accelerator is programmable and only the program must be reloaded, while the reconfigurability refers to a parametrized synthesizable core. This approach can be used in embedded computation or in FPGA services offered in cloud.

Another approach is to use the accelerator as an IP integrated on the same chip with a general-purpose CPU.

In any form of the actual implementation, the accelerator must be integrated in the heterogeneous system as a hardware kernel library of a currently used library (for example Eigen). This approach will allow the development of the library in a high- level language using the kernel library, while the kernel library remains to be optimized in assembly language.

## 6. Conclusion

The proposed MapReduce architecture (with *p* cells) provides in most of the cases a robust *O(p)* acceleration with a structure having the size in *O(p)*. The energy consumed is around *5x* less than of-the-shelf solutions provided by the use of GPUs as GPGPUs. The structural and architectural features of our proposal allow a performance O(*log p)*-times higher than the currently used accelerators due to the ***main novelty*** of our approach:

- we provide a fully covered **hardware support for reduction** functions.
- in most of the cases, using a special hardware feature and an appropriate algorithmic approach **we avoid the latency** introduced by the *log*-depth reduction network
- **data transfer** between the computational array MAP and the external memory Memory (see Figure 1) **is performed transparent** with the computation, allowing a partial reduction of the I/O bounding effect.

The integration in heterogeneous systems is efficiently done using the intermediary step of a kernel library optimized in assembly language.

The remaining weakness, only for KMC, MSC, EMC parallel algorithms, is related with the data transfer between Memory and MAP. Maybe future technological improvements will allow to accommodate on the silicon chip more distributed memory, thus avoiding partially the data movement between the MAP section and the external Memory.

## Acknowledgements

The authors got support from the technical contributors to the development of the *ConnexArray*$^{TM}$ technology: Emanuele Altieri, Frank Ho, Bogdan Mîțu, Marius Stoian, Dominique Thiébaut, Tom Thomson, Dan Tomescu.

## References

[1] G. Seif, *The 5 Clustering Algorithms Data Scientist Need to Know*, https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68
[2] M. Deshpande, *Clustering with Gaussian Mixture Models*, 2017, https://pythonmachinelearning.pro/clustering-with-gaussian-mixture-models
[3] G. Ștefan, A. Sheel, B. Mîțu, T. Thomson, D. Tomescu, “The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing”, in *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford University, August 20 to 22, 2006. https://youtu.be/HMLT4EpKBAw at time 35:00.
[4] M. Malița, G. Ștefan, D. Thiébaut, ”Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation”, in *ACM SIGARCH Computer Architecture News*, Vol. 35 , no. 5, Dec. 2007, pp. 32-38.
[5] G. Ștefan, M. Malița, “Can one-chip parallel computing be liberated from ad hoc solutions? A computation model-based approach and its implementation”, in *18th Inter. Conf. on Circuits, Systems, Communications and Computers*, 2014, pp. 582--597.
[6] G. Sissons, *GPU Accelerated R in the Cloud with Teraproc Cluster-as a Service*, 2015. https://devblogs.nvidia.com/gpu-accelerated-r-cloud-teraproc-cluster-service/
[7] A. Minnaar, *A CUDA Implementation of the K-Means Clustering Algorithm*, 2019. http://alexminnaar.com/2019/03/05/cuda-kmeans.html
[8] S. Tomov, R. Nath, H. Ltaief, J. Dongarra, “Dense linear algebra solvers for multicore with GPU accelerators”, in *24th IEEE International Symposium on Parallel and Distributed Processing*, Atlanta, Georgia, 2010.
[9] J. Vegh, “The Need for Modern Computing Paradigm: Science Applied to Computing”, in 6th *Annual Conference on Computational Science & Computational Intelligence*, Dec. 5-9 2019, Las Vegas, Nevada.
[10] H. Mujtaba, *Nvidia Pascal GP100 GPU Expected to Feature 12 TFLOPs of Single Precision Compute, 4 TFLOPs of Double Precision Compute Performance*, 2016.