High-throughput fuzzy clustering on heterogeneous architectures

Juan M. Cebrian[b], Baldomero Imbernón[a], Jesús Soto[a], José M. García[b], José M. Cecilia[a,c]

[a]*Department of Computer science, Universidad Caólica San Antonio de Murcia (UCAM), 30107, Murcia, Spain*
[b]*Department of Computing and Systems, University of Murcia, 30071, Murcia, Spain*
[c]*Department of Computer Engineering (DISCA), Universitat Politècnica de València (UPV), 46022, Valencia, Spain*

Abstract

The Internet of Things (IoT) is pushing the next economic revolution in which the main players are data and immediacy. IoT is increasingly producing large amounts of data that are now classified as "dark data" because most are created but never analyzed. The efficient analysis of this data deluge is becoming mandatory in order to transform it into meaningful information. Among the techniques available for this purpose, clustering techniques, which classify different patterns into groups, have proven to be very useful for obtaining knowledge from the data. However, clustering algorithms are computationally hard, especially when it comes to large data sets and, therefore, they require the most powerful computing platforms on the market. In this paper, we investigate coarse and fine grain parallelization strategies in Intel and Nvidia architectures of fuzzy minimals (FM) algorithm; a fuzzy clustering technique that has shown very good results in the literature. We provide an in-depth performance analysis of the FM's main bottlenecks, reporting a speed-up factor of up to 40× compared to the sequential counterpart version.

Keywords: Parallel fuzzy clustering, fuzzy clustering, fuzzy minimals

---

Email addresses: `jcebrian@ditec.um.es` (Juan M. Cebrian), `bimbernon@ucam.edu` (Baldomero Imbernón), `jsoto@ucam.edu` (Jesús Soto), `jmgarcia@um.es` (José M. García), `jmcecilia@ucam.edu` (José M. Cecilia)

1. Introduction

We are witnessing the steady transition from small to big data era. Data sets generated in different sectors such as industry 4.0, social media or precision agriculture are continuously increasing (just to mention a few). The overwhelming amount of data generated and the desire to process this information to generate valuable knowledge, pose compelling computational challenges to data-intensive kernels such as clustering. Despite the end of the Moore's law is looming in the horizon [1], computing systems are still increasing performance with each generation. The advent of heterogeneous systems, where multi-and-many core processors are plugged into the same motherboard (or even the same chip), also benefits from this continued growth in performance. Hardware platforms such as Graphics Processing Units (GPUs), Intel Xeon Phi or, to a lesser extent, the field-programmable gate arrays (FPGAs) are nowadays being extensively used by the scientific community as the only solution to overcome the challenges of processing such data deluge [2, 3].

However, heterogeneous architectures introduce serious difficulties in terms of programmability, being these even more accentuated when performance is mandatory (e.g., real-time), as they introduce several levels of parallelism, such as instruction-level parallelism (ILP), thread-level parallelism (TLP) and data-level parallelism (DLP), altogether. The first two parallelism levels have been traditionally exposed in latency-oriented and multicore architectures, but the best way to expose the latter is through vector computations, where a single instruction deals with several input elements (e.g., SIMD extensions [4, 5, 6, 7, 8], dedicated vector computing [9] or GPUs [10, 11]). Therefore, the effective and efficient use of heterogeneous hardware to serve an application remains a challenge.

Of particular interest to the scientific community are clustering algorithms, which aim to group a collection of individuals (measures, points, patterns, etc.) into clusters, mainly based on a given similarity or distance metric, such as Euclidean or Mahalanobis distance [12]. The individuals within the same cluster are closer than those that belong to a different cluster [13]. Clustering algorithms have been successfully applied to the analysis of data sets in various fields, such as image processing [14], smart cities [2, 15] or bioinformatics [16], to provide valuable knowledge in these fields.

Most of today's clustering algorithms rely on iterative procedures to find optimal solutions in high-dimensional datasets. The ability to find these solutions often requires many experiments with different algorithms and the study of the influence of different parameters and characteristics of the datasets. Therefore, clustering algorithms are computationally hard, for instance, the classical method *k-means* is hard as NP-complete even when the number of clusters to find is $k = 2$ [17]. Therefore, parallelization of clustering algorithms becomes mandatory in the big data era. Actually, some parallelization approaches have been published in the bibliography. The first clustering algorithm that was proposed for parallelization was the *k-means* [18, 19]. However, the *k-means* algorithms provides a hard clustering; i.e. each data point belongs exactly to a single cluster. Fuzzy clustering algorithms provide multiple and non-dichotomous cluster memberships. Among fuzzy clustering algorithms, it is noteworthy to highlight the fuzzy c-means (FCM) algorithm [20], which has been parallelized in the literature in datasets [21, 22, 23, 24, 25]. The FCM requires prior knowledge of the number of clusters to be generated (such as k-means), and then several runs must be made to calculate the optimal number of clusters, which increases computational time requirements. The FM clustering algorithm [26, 27] does not need prior knowledge about the number of clusters and presents the advantage that the clusters do not need to be compact well-separated, being this feature a requirement for parallelization [28].

In this paper, we present the first fully developed FM algorithm for the fuzzy classification of large datasets on Intel and Nvidia-based heterogeneous architectures. A data parallelism approach, which is better suited to the parallelism model of these architectures, is used to enhance performance. Our major contributions include the following:

1. To the best of our knowledge, this is the first data-parallelism scheme on GPUs and Intel CPUs/Xeon Phi for the FM clustering algorithm that exploits both coarse-grain (TLP) and fine-grain (DLP) parallelism.
2. Scalability analysis with vector register size on latency-oriented (Skylake-X) and throughput-oriented (Knights Landing) platforms.
3. Detailed evaluation via performance counters of the main performance bottlenecks of the FM clustering algorithm on Intel CPUs.
4. In-depth analysis of several GPU parameters are tuned to reach a speed-up factor of up to 12x, compared to Intel multithreaded and vectorized CPU counterpart version.
5. Propose an heterogeneous FM clustering implementation (GPU + CPU) approach that outperforms all individual platforms by up to $40\times$.

The rest of the paper is organized as follows. We briefly introduce FM clustering algorithm and programming heterogeneous Nvidia and Intel architectures in Section 2. In Section 3 we present vector and GPU designs for the main stages of the FM algorithm. Our experimental methodology is outlined in Section 4 before we describe the performance evaluation of our algorithm. Finally, Section 6 summarizes our findings and conclude with suggestions for future work.

2. Background

2.1. Fuzzy Clustering

Clustering analysis consists of assigning data points to clusters (or groups) so that point belonging the same cluster are as similar as possible, while points belonging to another group are as different as possible. This similarity process is based on the evaluation (i.e. minimization) of an objective function, which includes measures such as distance, connectivity and/or intensity. Different objective function may be chosen, depending on the dataset features or the application. Fuzzy clustering is a set of classification algorithms in which each data point can belong to more than one cluster. Typical examples of these algorithms include the FCM and the FM algorithms [20, 28]. The latter was initially proposed by Flores-Sintas *et al.* where authors demonstrate that it meets the expected characteristics of a classification algorithm; i.e. scalability, adaptability, self-driven, stability and data-independent. In addition, the FM algorithm does not require any prior knowledge (via input parameter) of the number of prototypes to be identified in the data set, as is the case with the FCM algorithm. In the following, we review the description of FM and refer to the reader to [26, 29, 27, 28] for insights in the definition and demonstration of FM algorithm.

Let X be a set of n data points, such that

$$X = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^F,$$

where F is the dimension of the vector space.

Algorithm 1 shows the sequential baselines of the FM algorithm. This algorithm has two main procedures (1) the calculation of the r factor (line 4) and (2) the calculation of prototypes or centroids (lines 5 to complete the set V). The r factor is a parameter that measures the isotropy in the data set. The use of Euclidean distance implies that the homogeneity and isotropy

Algorithm 1 The FM algorithm, where X is the input dataset to be classified, V is the algorithm output that contains the prototypes found by the clustering process. F is the dimension of the vector space.

```
1: Choose $\varepsilon_1$ and $\varepsilon_2$ standard parameters.
2: Initialize $V = \{ \} \subset \mathbb{R}^F$.
3: $Load\_Dataset(X)$
4: $r = Calculate\_r\_Factor(X)$
5: $Calculate\_Prototypes(X, r, \varepsilon_1, \varepsilon_2, V)$
```

of the features space are assumed. Whenever homogeneity and isotropy are broken, clusters are created in the features space.

$$\frac{\sqrt{|C^{-1}|}}{nr^F} \sum_{x \in X} \frac{1}{1 + r^2 d_{xm}^2} = 1. \tag{1}$$

The calculation of r factor is shown in Equation 1. It is a non-linear expression, where $|C^{-1}|$ is the determinant of the inverse of the covariance matrix, m is the mean of the sample X, d_{xm} is the Euclidean distance between x and m, and n is the number of elements of the sample. It is noteworthy to highlight that the calculation of the r factor is the most time consuming part of the algorithm for big datasets, as it requires to computer for each point the convolution and determinant of the input matrix.

Once the r factor is calculated, the calculation of prototypes is executed to obtain the clustering result in V (see Algorithm 2). The objective function used by FM is given by Equation 2

$$J_{(v)} = \sum_{x \in X} \mu_{xv} \cdot d_{xv}^2, \tag{2}$$

where

$$\mu_{xv} = \frac{1}{1 + r^2 \cdot d_{xv}^2}, \tag{3}$$

Equation 3 measures the degree of membership for a given element x to the cluster where v is the prototype. The FM algorithm is an iterative procedure that aims to minimize the objective function through the Equation 4, giving the prototypes represented by each cluster. Finally, ε_1 and ε_2 are

Algorithm 2 *Calculate_Prototypes*() of FM algorithm.

```
1: for $k = 1; k < n; k = k + 1$ do
2:     $v_{(0)} = x_k$, $t = 0$, $E_{(0)} = 1$
3:     while $E_{(t)} \geq \varepsilon_1$ do
4:         $t = t + 1$
5:         $\mu_{xv} = \frac{1}{1+r^2 \cdot d_{xv}^2}$, using $v_{(t-1)}$
6:         $v_{(t)} = \frac{\sum_{x \in X} \left(\mu_{xv}^{(t)}\right)^2 \cdot x}{\left(\mu_{xv}^{(t)}\right)^2}$
7:         $E_{(t)} = \sum_{\alpha=1}^{F} \left(v_{(t)}^{\alpha} - v_{(t-1)}^{\alpha}\right)$
8:     end while
9:     if $\sum_{\alpha}^{F} (v^{\alpha} - w^{\alpha}) > \varepsilon_2$, $\forall w \in V$ then
10:        $V \equiv V + \{v\}$.
11:    end if
12: end for
```

input parameters which establish the error degree committed in the minimum estimation and show the difference between potential minimums respectively.

$$v = \frac{\sum_{x \in X} \mu_{xv}^2 \cdot x}{\sum_{x \in X} \mu_{xv}^2} \tag{4}$$

2.2. Programming heterogeneous clusters

High Performance Computing (HPC) programmers have relied on the scalability of technology to deliver greater performance. The design of heterogeneous architectures, combining CPUs and accelerators, such as GPUs and TPUs [30], is now seen as the only solution to continue scaling Moore's law through specialization [31, 32]. Parallel multithreaded implementations are not always efficient in such heterogeneous systems, as they are designed for homogeneous systems in which all cores have the same computing capabilities. In addition, exploiting fine-grained parallelism via vectorization adds an additional complexity layer to the problem. For example, vector units inside most modern processors have a fixed size (e.g., SSE/NEON has 128-bits, AVX/AVX2 has 256-bits or AVX512 has 512-bits), although they are usually backwards compatible. ARM's SVE tries to solve this issue using a vector-length agnostic (VLA) programming model (the ISA does not assume a fixed vector register size) [6]. Moreover, new SIMD extensions usually add additional functionality via new instructions, further increasing the complexity when dealing with heterogeneous systems that have different ALU capabilities.

Vectorization offers a potential speedup that scales linearly with the vector register size. Vector register size has steadily doubled every 4 years [33] since the late 90s. Interestingly, doubling the register size can be significantly more energy efficient than doubling the number of cores. However, the generation of efficient vector code with increasing register size has several obstacles to overcome. Key performance limiting factors include: (a) at a design level, horizontal operations, data structure conversion and divergence control, and (b) at a hardware level, register bank size, cache and memory bandwidth and resource under utilization. Ultimately, the effectiveness the architecture depends on its ability to vectorize large quantities of code [34].

Automatic vectorization has been widely studied over the years to exploit vector capabilities. However, compilers show important limitations when vectorizing code with pointers and function calls. This is usually related to traditional compiler analysis limitations (e.g., pointer aliasing analysis) [34]. On the other hand, user-directed vectorization offers a portable solution to vectorization. The programmer can assist the compiler highlighting what regions of the code are candidates to be vectorized (in the form of #pragma directives, similar to OpenMP). While this improves code portability, there are certain low level code optimizations that are nearly impossible to produce by the compiler [35]. More specifically, SIMD codes require both function call vectorization and superblock optimizations to improve performance over scalar implementations, making manual vectorization the most suitable choice in our evaluation. Manual vectorization usually consists on the translation of high level source codes (usually the most time-consuming and inner-most loops) to equivalent low level intrinsic or assembly code.

Therefore, programmers play a fundamental role in this heterogeneous paradigm since programs have to be developed using different programming models through APIs such as OpenMP [36], MPI [37], OpenCL [38] or CUDA [39] to fully leverage heterogeneous systems. Moreover, they must also exploit resources of different characteristics within the chip, such as vector units.

3. Targeting heterogeneous processors

This section introduces the parallelization strategies applied to the FM algorithm on heterogeneous architectures. Particularly, our designs target two different architectures: Intel and Nvidia. FM has a fundamental advantage over other fuzzy clustering techniques, i.e. it does not require comparing groupings with each other to minimize an objective function. Therefore, our approach can take advantage of this feature to divide the original data set into subsets and then perform FM independently. Each independent subset is classified using an objective function that includes the r factor, previously calculated on the original dataset. The r factor provides general information about the dataset, ensuring that subsets do not lose information about global properties for classification.

3.1. Intel-based parallelization

As discussed in Section 2, the FM algorithm can be divided in two clearly independent steps, a) calculation of the r factor (Algorithm 1, line 4 and b) calculation of prototypes and add them if a distance condition is met (Algorithm 2). The parallelization of both stages in Intel-based architectures is briefly explained below.

Calculate r factor parallelization: A detailed profiling analysis of the factor r computation reveals that the most time consuming functions performed a convolution and a determinant for each row of the input matrix. This section can be parallelized at a coarse-grain using OpenMP. There are no critical sections in this code, except a final reduction to combine all partial factor r computations. The scalability of this code section is almost linear with thread count. We do not perform manual vectorization of this code section since we rely on Eigen to perform matrix computations (convolution and determinant) [40].

Calculate prototypes parallelization: The factor r is a global parameter that provides information about the whole data-set. This enables the CPU implementation to exploit coarse-level parallelism via OpenMP (Algorithm 2). However, this parallelization requires a critical section (lines 9 and 10) to add candidate prototypes to the final output. This critical section has O(N), meaning that the more prototypes in the output, the longer it will take to insert new ones.

The next step is to perform fine-grain parallelization via vectorization. A detailed profiling revealed two possible hot-spots where vectorization could be applied, algorithm 2 lines 6-7 and the euclidean distance required in line 5, being the later the most time consuming (around 90%

Algorithm 3 SIMD version of Algorithm 2 line 5.

```
1: for $l = 0; l < columns; l += SIMD\_WIDTH$ do
2:     $aux = simd\_load(PROTOTYPE[l]) - simd\_load(ROW[l])$
3:     $distance += aux \times aux$
4: end for
5: $scalar\_distance = reduce\_add(distance)$
6: $output = \frac{1}{1 + rFactorSquare \times scalar\_distance}$
```

of total execution time). We performed text-book manual vectorization of the euclidean distance function (Algorithm 3). However, there are several limitations to the potential speedup of the vectorized code: a) horizontal operations at the end of the computations (reduction) b) increased L2 cache misses with high number of dimensions (columns) c) low arithmetic intensity[1] and d) the division in algorithm 2 line 5. On the other hand, vectorization of lines 6-7 also has low arithmetic intensity and a division, but still it is better balanced. Intel's Top-Down model[2] information confirmed these hypothesis [41]. More detailed information will be provided in the evaluation section.

Finally, we decided to test superblocking in order to make use of the SIMD divisor ports available in the system. In order to do so, we unrolled 16 iterations (rows) of the main loop (see Algorithm 2), computing their distances, and then building two vector registers containing all distances, that we could then use in the rest of the code (Algorithm 4. This was implemented only for AVX512, and it was the best performing version we found. We also tried unrolling 8 iterations, but results were slightly worse. Superblocking further decreased cache locality, but the use of the SIMD division ports paid off in performance terms.

3.2. CUDA-based parallelization

This section shows CUDA parallelization strategies applied to FM systems. Likewise the Intel's approach, the explanation is given for each of the main functions in the algorithm.

[1]A measure of operations performed relative to the amount of memory accesses (Bytes).

[2]Performance counter information is used to define the processor behaviour for a given workload, and to detect hardware bottlenecks.

Algorithm 4 Superblocked SIMD version of Algorithm 2 line 5 (16 iterations) for a vector register of 8 doubles (512-bits).

```
1: for $block_count = 0; block_count < 16; block_count++$ do
2:     $aux = 0; distance[block_count] = 0$
3:     for $l = 0; l < columns; l+= SIMD\_WIDTH$ do
4:         $aux = simd\_load(PROTOTYPE[l]) - simd\_load(ROW[l + block_count * columns])$
5:         $distance[block_count] += aux \times aux$
6:     end for
7:     $scalar\_distance[block_count] = reduce\_add(distance[block_count])$
8: end for
9: $simd\_output1 = \frac{[1..1]}{[1..1]+[rFactorSquare..rFactorSquare]\times scalar\_distance[0..7]}$
10: $simd\_output2 = \frac{[1..1]}{[1..1]+[rFactorSquare..rFactorSquare]\times scalar\_distance[8..15]}$
```

Algorithm 5 *R Factor calculation algorithm in GPU*

```
1: for $i = 1; i < rows; i = i + 1$ do
2:     $covariance <<< bl, th, sh >>> (dataset, determ, row_i, rows, cols)$
3:     $detvalue = cusolver\_thrust(determ)$
4:     $rfactor+ = \frac{1}{\sqrt{detvalue}}$
5: end for
```

Calculate r factor parallelization (see Algorithm 5): As previously explained, the r factor procedure has two different tasks to be performed for each row in the dataset. They are the calculation of the fuzzy covariance matrix, which is a square matrix ($columns \times columns$) and the calculation of its determinant. It is noteworthy to highlight that there as many iterations as rows (points in the datasets) but the performance is also affected by the number of columns (i.e. the dimension of points F). Actuality, performance will be drastically affected by this last parameter, since the execution time of the determinant calculation will increase exponentially with the size of the matrix.

The determinant calculation can be performed by several methods [42]. One of the best known is to use factorization. With factorization, the determinant can be broken down into the product of two matrices, one triangular and one stepwise. The value of the determinant would be the product of the elements of the diagonal of the stepwise matrix. Partic-

Algorithm 6 *covariance (dataset, determ, p, rows, cols)*

```
1: for $i = 1; i < cols; i = i + 1$ do
2:     for $j = 1; j < cols; j = j + 1$ do
3:         $sum = 0$
4:         for $k = 1; k < rows; k = k + 1$ do
5:             $sum += (dataset[k][i] - p[i]) * (dataset[k][j] - p[j]);$
6:         end for
7:         $determ[j][i] = sum/rows;$
8:     end for
9: end for
```

ularly, the determinant is calculated by using the LU factorization as it is the technique used by the `Eigen` library. CUDA offers the `cuSOLVER` library for 64-bit platforms available from CUDA 7 and beyond. This library is based on `cuBLAS` and `cuSparse` libraries. It is also composed of three libraries that can be used independently `cuSolverDN`, `cuSolverSP` and `cuSolverRF`. In this work we use `cuSolverDN`, which it works with dense matrix factoring, and it incorporates several factoring routines such as LU, QR or Cholesky. Finally, the `thurst` library is used to reduce the diagonal in the GPU to obtain the determinant result.

Next, the fuzzy covariance matrix is calculated to obtain r factor. This calculation uses the whole dataset to construct the determinant. Algorithm 6 shows the sequential implementation to obtain the fuzzy covariance matrix, where the dataset is received as an input, *rows* and *cols* are rows and columns of data input, and p is a vector. This algorithm is memory bounded but there is some data locality in the most-inner loop. Thus, the GPU shared memory is used to improve the overall application bandwidth.

Calculate prototypes parallelization: The second step of the FM algorithm is the calculation of prototypes. This task is implemented on the GPU using each thread to calculate the distance between each row dataset and the prototype. Next, each thread calculates the equation 3 to obtain the probability of belonging to each prototype. Finally, in the GPU, a reduction per block must be done to obtain the total amount of this pertinence for each row dataset.

3.3. Performance Guidelines

This section provides a summary of our experiences during the optimization process of the FM algorithm on heterogeneous architectures. In this paper we have evaluated three different platforms: GPUs, server CPUs and the KNL architecture. GPUs are massively parallel architectures, where maximizing performance requires a huge amount of TLP. In many cases, thread count must be several orders of magnitude higher than the actual number of cores of the GPU. Thread independence, minimum synchronization, maximizing local memory usage and overlapping computation and data transfers to GPU memory are also required to extract performance of such architectures. As a drawback, GPU cores are simple. Therefore, complex operations such as divisions or square roots have a long latency.

On the other hand, we have CPUs (and KNL). In these architectures the user does not need to transfer data manually, and everything is handled transparently by the memory hierarchy. The amount of TLP available is smaller (i.e. one to four threads per core). Synchronization among threads is "easier" thanks to coherence protocols, which ensure the memory consistency model. In addition, vector ALUs are far more complex than the GPU's ALUs. These ALUs are usually pipelined (not in KNL), and can handle complex operations much faster. Clock frequencies are also 3 to 4 times faster in CPUs than in GPUs/KNL. This benefits serial code sections and functions with low levels of TLP.

The first step in any optimization process is to profile the application looking for performance bottlenecks. Tools such as Intel's Vtune, AMD's CodeXL/uProf or Nvidia's Nsight are openly available to developers. Once the developers have an insight of the code and its bottlenecks, they need to break down the code in functions that can be handled by standard or math libraries. The use of libraries is always recommended for several reasons.

- Correctness and development times.
- Code portability. Every time a new hardware is released, developers of these libraries update their codes to support this new hardware.
- Performance portability. Libraries are usually optimized for each architecture that is supported.

For the particular case of the FM algorithm, we realized that most of the total execution time of the algorithm referred to the r factor and the calculation of the prototypes. The r-factor calculation involves convolutions and determinant computations, which can be carried out by using mathematical libraries (e.g., Eigen or cuSOLVER). However, the prototype computation could not be easily adapted to any known standard or mathematical library.

The next step is to perform coarse-grain parallelization on the different code sections. We recommend the use of standard programming interfaces (API), such as OpenMP or OpenACC, to benefit code portability. Nevertheless, please note that these APIs do not ensure performance portability. Efforts should focus on outer loops that require minimum synchronization among threads. The r factor computation can be parallelized at a row level requiring minimum synchronization. On the other hand, the calculate prototype function requires a critical section to protect the insertion of new prototypes in the final output. This limits the scalability as the number of prototypes increases.

At this point developers should realize which code sections are more suitable to run on a GPU (if any). As discussed before, we need massive parallelism with minimal synchronization. We will benefit from not having complex instructions in the code (e.g., division). In addition, computations should take longer than the time it takes to transfer the data in and out of the GPU memory. The r factor computation matches all these requirements. It is massively parallel, with no complex operations. Moreover, the output of this function only produces a single double precision element, not requiring large data transfers out of the GPU memory. This means that GPU computations can be overlapped with data transfers, since bandwidth requirements for moving data out are minimal. On the other hand, prototype computation seems more suitable for CPU computations, since it uses complex operations (divisions). This code also has a critical section that will benefit from the faster core frequencies when dealing with big datasets that produce many candidate prototypes. Developers should handle computation and communications phases between these heterogeneous architectures in a pipelined fashion, overlapping GPU data transfers, GPU computation and CPU computations.

Finally, we should try to extract fine-grained parallelization (vectorization) on CPUs (KNL). First, developers should try compiler auto-vectorization (e.g., Intel ICC -O2 or GCC -O3 perform auto-vectorization). If there is no effect on performance, try to get info from the compiler. For example, GCCs *-fopt-info-vec-missed* provides information about missed optimization opportunities from vectorization passes. Developers can then try to modify or add "pragmas" to the code to resolve compiler issues with auto-vectorization.

Nevertheless, in most cases, developers will have to rely on manual vectorization.

Vectorization efforts should focus at the inner-most loop level. If performance remains unaffected, profile the application again, looking for reasons that limit vector scalability. If profiling reveals that the code section is heavily memory bound, there is nothing else the developer can do but to try compression techniques (e.g., [43]). On the other hand, if profiling reveals other bottlenecks, as it happened in our FM code with high divisor ALU contention, developers should try outer-loop vectorization or macro-blocks to make better use of the hardware resources.

4. Evaluation

This section shows an experimental evaluation of the FM parallelization strategies running on a heterogeneous systems based on Intel CPUs and Nvidia GPUs. First of all, we briefly introduce the hardware and software environment where the experiments are carried out.

4.1. Hardware environment and Benchmarking

Intel-based platform The Skylake-X i7-7820X CPU processor used in our evaluation has eight physical cores with sixteen threads running at 3.60GHz. It has 32+32KB of L1 and 1MB of L2 cache per core plus 11MB of shared L3 cache. It offers support for SSE4.2 (128-bit registers), AVX2 (256-bit registers) and AVX-512 (512-bit registers) instructions. On the other hand, the Xeon Phi 7250 processor has 68 cores with 272 threads running at 1.40GHz (but we only use 64 cores / 256 threads for computation as recommended by Intel). The L1 cache is identical to the i7-7820X but the L2 shares its 1MB between 2 cores. Our setup uses the 16GB HBM-MCDRAM as L3 shared cache.

Nvidia-based platform Platform with 2 hexa-core Intel Xeon E5-2650 at 2.20 GHz, 128 GB of RAM, private L1 and L2 caches of 32 KB and 256 KB per node, and a L3 cache of 32 MB shared by all the cores of a socket. It includes a GPU Nvidia GTX 1080 Ti(Pascal), with 3584 cores (28 SM and 128 SP per SM), 12 Global Memory DDR5 and 48 KB of shared memory per block.

The dataset to test our implementation is an input database with 100K points, belonging to five hyper-ellipsoids S_k, with $S_k \subset \mathbb{R}^{80}$, $\forall k \in \{1, 2, 3, 4, 5\}$, and $S_i \cap S_j = \emptyset \forall i \neq j$. The cardinal of each subset is: $|S_1| = 20868$, $|S_2| = 20104$, $|S_3| = 19874$, $|S_4| = 22380$, $|S_5| = 16774$, respectively.

4.2. Runtime Evaluation

This section shows the evaluation results for different input sizes (varying both the number of rows and columns). Columns represent the number of variables for each element that should be clustered (e.g., if elements are cars, each column represents a characteristic of the car, like brand, color, emissions, etc). On the other hand, rows represent different instances of the elements to classify (e.g., different cars). Our goal is to show the scalability on each dimension, since each field of application has different requirements. First we perform a detailed evaluation of different SIMD optimization strategies. Second, we compare all architectures: the GTX 1080 Ti (GPU), the Xeon Phi 7250 (KNL) and the Skylake 7820X (SKL-X). We have also tested different thread counts, but for the sake of clarity we focus on the best configuration for each architecture (maximum thread count).

4.2.1. SIMD Evaluation

Figure 1 shows the per-thread performance improvement (speedup) of the SIMD implementations over the scalar code for different input sizes. We show results for 1 and 16 threads to analyze the effects of sharing the 11MB of L3 cache. Note that the AVX512 results shown in the figures correspond to the superblocked version. The straightforward AVX512 version behaved similarly to SSE and AVX and is omitted for the sake of clarity.

The first thing to notice is that SSE shows the expected speedup of $2\times$ when the number of columns is greater than 8. Note that our implementations use double precision floating point, so only two doubles can be fitted into the 128-bit SSE register. However, for 8 columns, the speedup is slightly lower (around $1.5\times$). This can be explained by checking the instruction count for SSE (Figure 2-top). It can be seen that it is close to $1.73\times$ lower than the scalar code for 8 columns, while for 16 columns is $1.86\times$. In addition, when running over 16 columns or more, the backend is blocked slightly less time (Figure 2-bottom), and the application is less memory bound (Figure 3-top). This can be explained by the additional horizontal operations in the code (to perform the reduction in the Euclidean distance). For 8 columns, we perform a reduction every 4 loop iterations, while for 16 columns, we perform a reduction every 8 loop iterations. The increase in core-boundness also comes

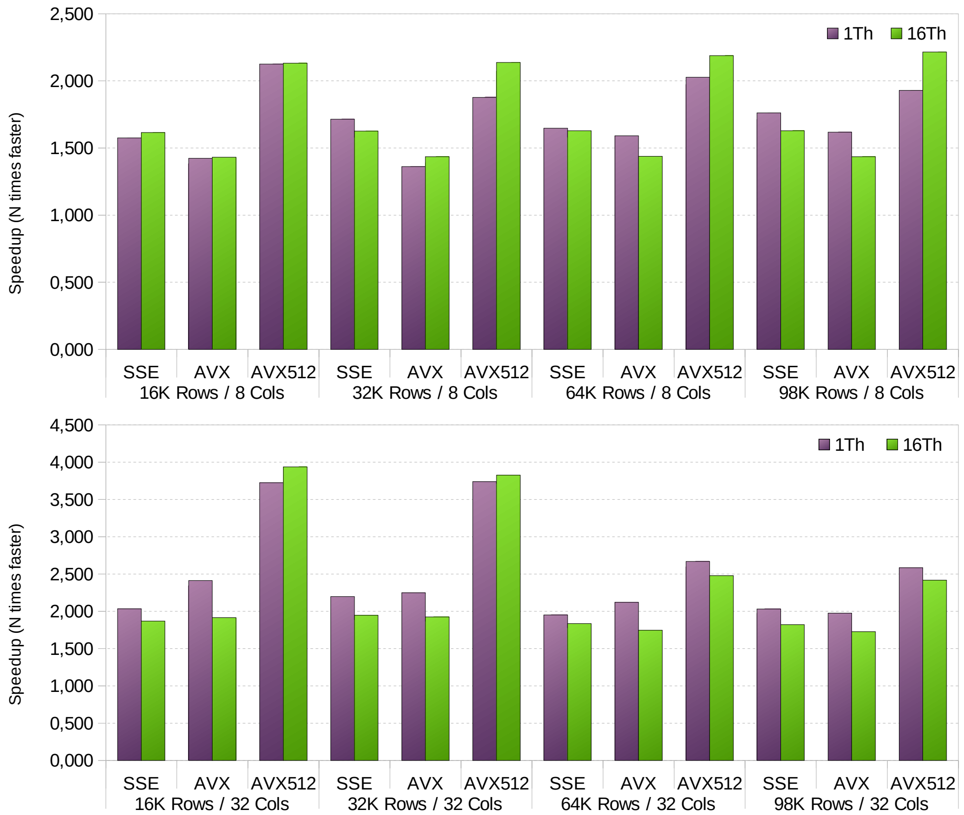


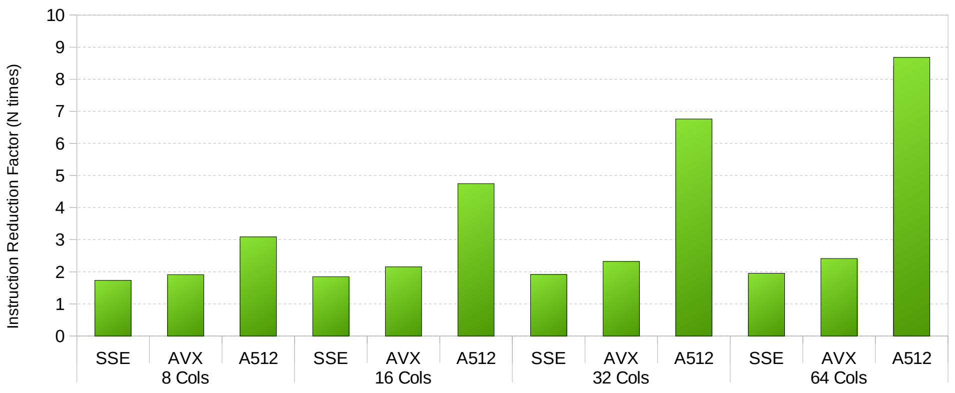
Figure 1: Per-thread speedup (over scalar runtime) for 1 and 16 threads and different SIMD implementations (SSE, AVX and AVX512). Input has 8 (top) and 32 (bottom) columns and different number of rows (X Axis).

from the cycle penalty of the non-vectorized divisions (that cost 14 cycles in Skylake-X and 32 for KNL [44]). Scalar divisions become a bottleneck when the rest of the code is computed faster via SIMD.

On the other hand, AVX performance is very similar to SSE. Figure 2-top shows that the instruction reduction factor of AVX is only slightly higher than SSE (not 4× as expected). This is mainly because AVX requires of four instructions to perform the horizontal reduction in the Euclidean distance, whereas SSE requires only one. As column count increases, the ratio of horizontal instructions decreases, and the instruction reduction factor goes up, and so does the speedup of AVX over SSE. Moreover, AVX is slightly slower than SSE for 8 columns, mainly due to additional backend stalls according to Figure 2-bottom. As it happened with SSE, the additional backend stalls come from horizontal operations and non-vectorized divisions.

Finally, the superblocked AVX512 outperforms both SSE and AVX in all cases, reaching 2× for some input sizes. Figure 2-top shows that the instruction reduction factor of the superblocked AVX512 scales much better than SSE and AVX. As it happened with AVX, the horizontal reduction in AVX512 is broken down into a set of micro-ops (around 15). The more columns in the input, the less ratio of horizontal operations over regular instructions we encounter, and thus the higher instruction reduction factor. However, the increase in the instruction reduction ratio is not always translated into a performance improvement. The answer to this behavior can be found in Figure 3. The top chart shows how the application becomes increasingly memory bound as the number of columns goes up. This is related to the arithmetic intensity going down as we reduce the instruction count via SIMD. In addition, the bottom chart shows what levels of the memory hierarchy are stalling the processor. As we can see, L2 is a critical factor for 8, 16 and 32 columns, while L3 and main memory are the main bottleneck for 64 columns. As the number of rows in the input increases, we reduce the cache locality in L3, and that has a significant performance impact on the superblocked AVX512 implementation.

Finally, it is worth mentioning that the performance difference of having exclusive access to the 11MB L3 (1th vs 16th in Figure 1) is minimal, and usually benefits the single core scenario.



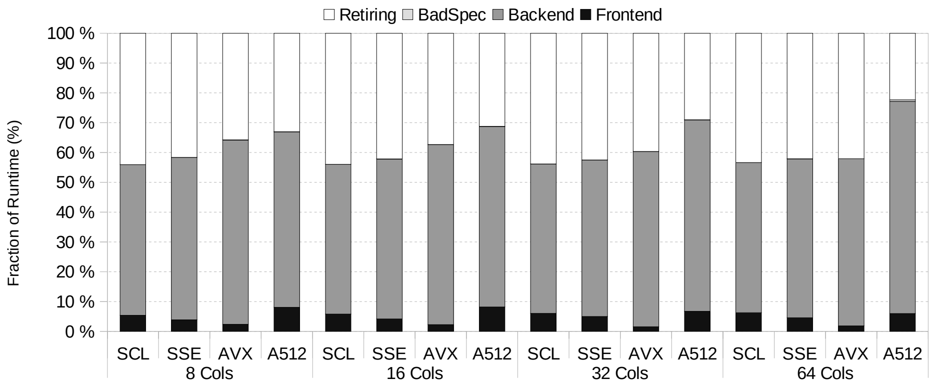


Figure 2: Instruction reduction factor (top) and level 0 of Top-Down model (bottom). Input has 16K rows and varying column count (X Axis).

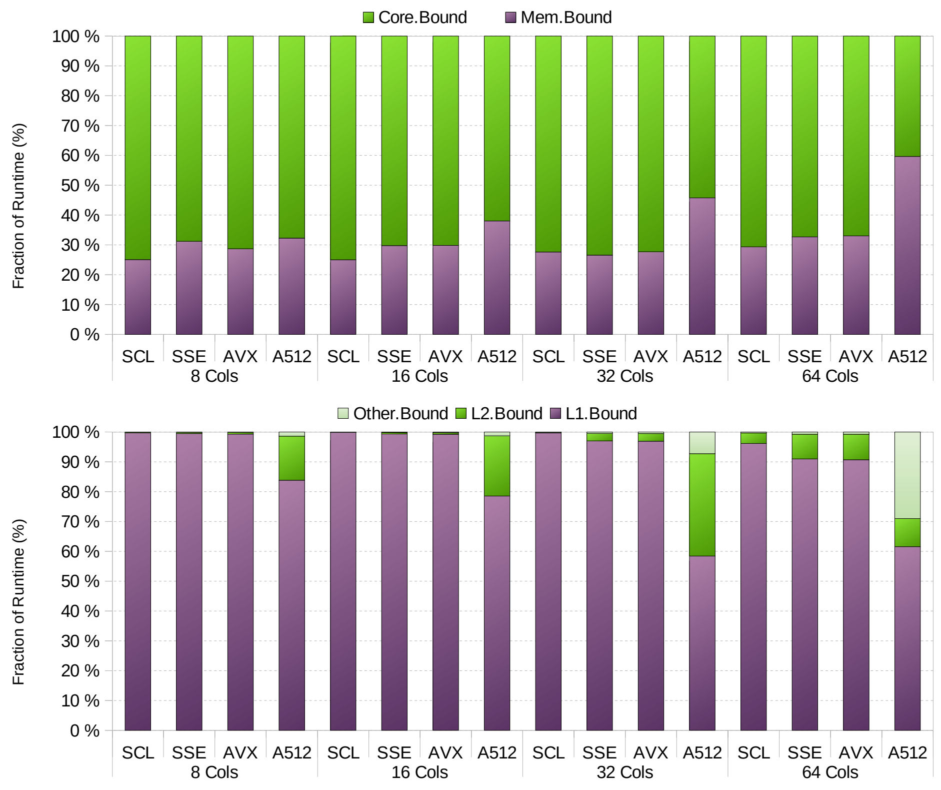


Figure 3: Level 1 (top) and Level 2 of Top-Down model (bottom). Input has 16K rows and varying column count (X Axis).

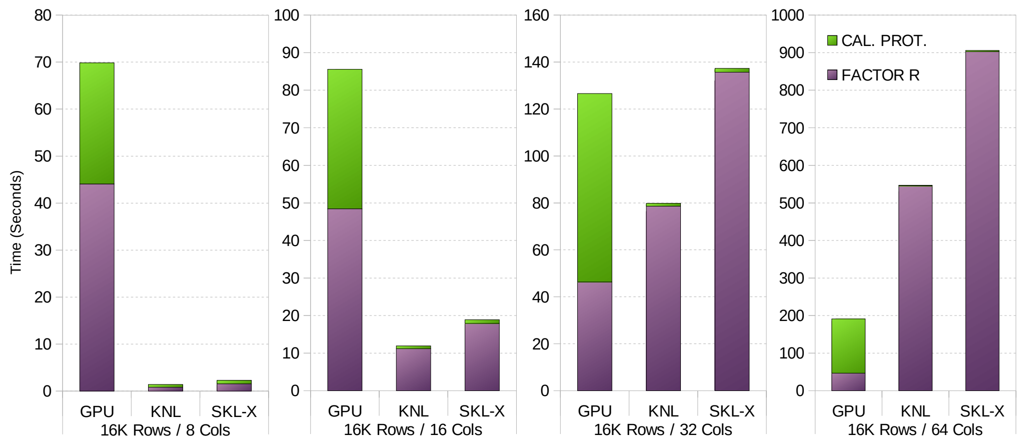


Figure 4: Runtime (Seconds) breakdown (Factor R + Prototypes) for the three evaluated platforms. Input has 16K rows and different number of columns (X Axis).

4.2.2. Platform Evaluation

The next step is to compare the different heterogeneous platforms to decide which one is the best candidate to perform the FM clustering technique. Figures 4 and 5 show the execution time breakdown of the FM kernel in two parts, factor r computation and prototype calculation for different input sizes. Each figure contains a plot for different number of columns (8, 16, 32 and 64), while the figures themselves vary the number of rows (16K and 98K). Intermediate row sizes can be extrapolated from these values.

The first thing to notice is that the GPU has a huge performance penalty for small number of columns when compared to other platforms, but it scales better along with the number of rows. This slow performance comes from two sources, first there is initialization time, and second it is the low floating point operations per core of the GPU when running double precision floating point. Secondly, it is clear that the GPU spends considerably more time calculating the prototypes than KNL and SKL-X. This is probably due to the divisions and the low arithmetic intensity in the prototype computation (Algorithm 1). This behavior is consistent across the different number of rows. The GPU clearly outperforms other platforms for factor r computations, running around 10 (18) times faster than the KNL (SKL-X).

However, KNL provides the best overall performance for less than 32 columns regardless of the number of rows. SKL-X is only 60 to 75% slower than KNL for 16K rows (Figure 4). In this scenario, the additional energy required and price cost of KNL does not seem justified, neither is the use of a GPU. However, as the row count increases (Figure 5), the massive thread count of KNL (256 compared to 16 on SKL-X) starts to pay off. Remember that we perform coarse-grained OpenMP parallelization over the rows.

Nevertheless, we must conclude that the best device in terms of money and energy efficiency to execute FM is neither of those platforms as a stand-alone. Indeed, the best performance requires going a step further in heterogeneous computation, and combining the execution on GPU (for factor r computation) and CPU (to estimate the prototypes). In addition, when processing several inputs, GPU and CPU computations can be overlapped, further reducing the total run-time of the application. Moreover, since the GPU seems under-utilized for factor r computation, running multiple instances of factor r is also plausible via streaming, virtualization or similar.

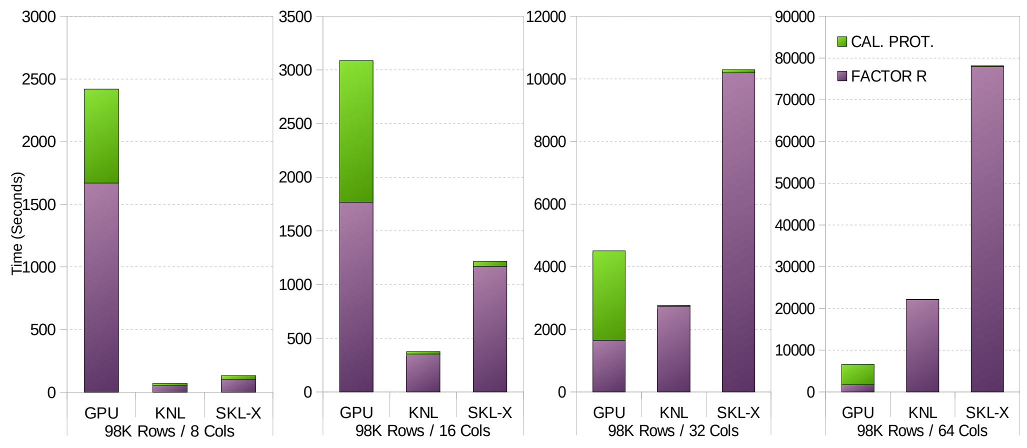


Figure 5: Runtime (Seconds) breakdown (Factor R + Prototypes) for the three evaluated platforms. Input has 98K rows and different number of columns (X Axis).

4.3. Quality Evaluation

To test the clustering efficiency of the FM algorithm, we use the database described in Section 4. More specifically, we generate 100K points that belong to five hyper-ellipsoids S_k, with $S_k \subset \mathbb{R}^{80}$, $\forall k \in \{1, 2, 3, 4, 5\}$, and $S_i \cap S_j = \emptyset \forall i \neq j$. For cardinals we use: $|S_1| = 20868$, $|S_2| = 20104$, $|S_3| = 19874$, $|S_4| = 22380$, $|S_5| = 16774$. The FM clustering output is a set of 37 prototypes, which looks very far from the actual result (i.e., 5 clusters). This variation comes from one of the parameters of the FM algorithm

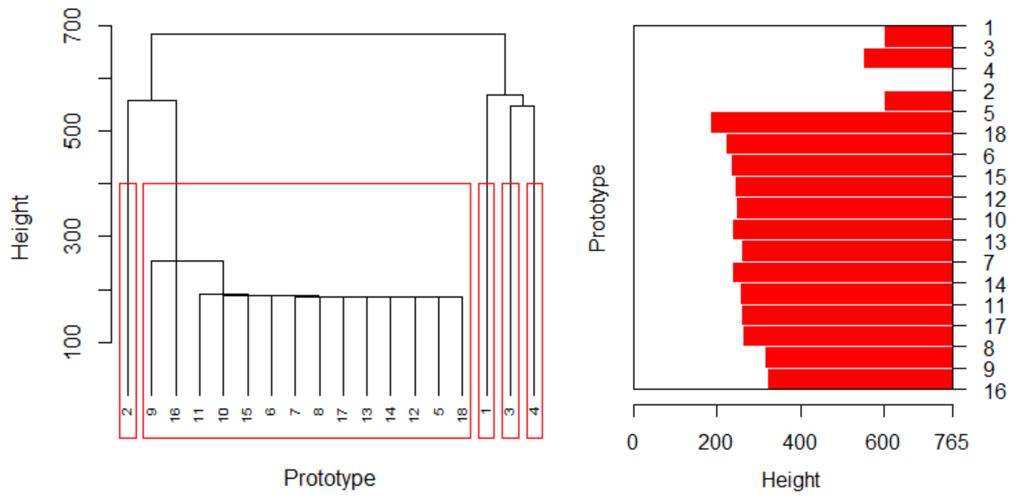


Figure 6: Dendrogram of the hierarchical agglomerative clustering of the FM prototypes (left). Bannerplot displays of prototype clusters (right).

that establishes the maximum distance of two prototype candidates. A high number of input points (100K in our case) will translate into a high number of prototypes if the maximum distance parameter is too small. However, in our evaluation we see that most close prototypes are representative of the same cluster. Thus, we perform hierarchical clustering to the output prototypes generated by FM to test which ones belong to the same cluster. More specifically, we use the hierarchical clustering analysis proposed by [45].

This algorithm is considered as one of the best techniques for partitioning objects into optimally homogeneous groups on the basis of empirical measures of similarity. The resulting clusters can be visualized as a dendrogram, which shows the sequence of cluster fusion and the distance at which each fusion took place. The agglomeration method used in our hierarchical clustering is the single-linkage clustering. As a measure of dissimilarity between sets we used the euclidean distance. Figure 6-left shows a dendrogram of our hierarchical clustering. If we divide the tree into five groups (or sets), as shown in red in Figure 6-left, these sets correspond unequivocally with the five hyper-ellipsoids the inputs are based on. This confirms the output quality of the FM algorithm.

This is the best case scenario, where we know in advance how the output should look like (around five clusters). However, there are other approaches that can help us to decide how many groups are required as the best solution given the output prototypes. Figure 6-right shows a "banner"[3] visualizing the (agglomerative or divisive) hierarchical clustering [46, 47]. Figure 6-right shows a big qualitative leap between the 4th and 2nd prototypes, while the 5th is representative of the rest. Therefore, a five cluster representation is clearly accurate, endorsing the results shown in Figure 6-left.

5. Related work

There are many methods in the literature for clustering [48], but only a few efforts have been made to parallelize them on different computing platforms. For instance, Jin *et al.* proposed a distributed single-linkage hierarchical clustering algorithm (DiSC) based on MapReduce [49]. Hou [50], Zhao *et al.* [51] and Xiong [52] also developed map reduce solutions using platforms, such as Hadoop, to improve performance of the k-means algorithm as applied to different context. In [53], authors introduced a hierarchical data structure and clustering algorithm (called parallel k-tree). This algorithm was designed to leverage clusters of computers to deal with extremely large datasets. They applied parallel k-tree to a large (8 terabyte) collection of Landsat 5 satellite images. In [54], authors optimize a particular implementation of k-means, triangle inequality, using a hybrid implementation based on MPI and OpenMP on homogeneous computing clusters.

GPUs have been also applied for the parallelization of clustering algorithms. Li *et al.* provide a GPU parallelization road-map of k-Means algorithm on a GPU, pointing out data dimensionality as an important factor in terms of performance on these platforms [55]. Therefore, they design different GPU implementations for low-and-high dimensional datasets. Different variations of k-means have been also ported to GPUs. For instance, the optimal tabu k-means clustering to increase efficacy by combining tabu search for calculation of centroids with clustering features of k-means [56]. Djenouri *et al.* use different HPC platforms for discovering parallel frequent item-sets in transactional databases and addresses the time complexity problem. Three different single scan (SS) algorithms are provided using GPUs. They exploit heterogeneous hardware by scheduling independent jobs to workers in a cluster of nodes equipped with CPUs and GPUs [57]. In [58], authors propose a

[3]A horizontal barplot.

classifier for efficiently mining data streams based on extreme learning machine, which is improved by an efficient drift detector and method to alleviate the bias towards the majority class.

Many works have also proposed the parallelization of k-means or, variations of it, as applied to particular problems, such as image segmentation [59], Circuit Transient Analysis [60], monitoring fault tolerance in Wireless Sensor Networks (WSN) [61], air pollution detection [2], health care data classification [62]. Finally, there are only few a articles that parallelize non-supervised clustering algorithms. Scully-Alison *et al.* propose a sparse fuzzy k-means algorithm that operates on multiple GPUs [63] and perform clustering for real environmental sensor data.

6. Conclusions and Future work

The analysis of large datasets has the potential to transform development and accelerate social progress worldwide. However, an optimal use of the available resources is required to process the information generated in near real time to ensure the success of this new paradigm. Current processors are heterogeneous and massively parallel. This makes it mandatory to redefine our algorithms to take full advantage of these architectures. This article shows the parallelization roadmap on Intel and Nvidia-based architectures for the FM algorithm to improve runtime classification of large data sets. Our results reveal that the Intel's KNL architecture is the best platform for small/medium datasets. Nvidia GPUs are very good to calculate the Factor r and for larger datasets. Moreover, KNL and SKL only show a 50% difference for prototypes (256th vs 16th), which means that KNL doesn't pay off in cost or energy, thus we may establish as the best solution the couple GPU+SKL by overlapping the Factor r calculation with the prototype calculation to completely hide the later.

Future work should include the redefinition of all data science algorithms that are applied for real-time decision making in the context of Internet of Things. Indeed, this redefinition must be carried out in the context of the specialization in which computational architectures are immersed.

Acknowledgments

This work was partially supported by the Fundación Séneca del Centro de Coordinación de la Investigación de la Región de Murcia under Project

20813/PI/18, and by Spanish Ministry of Science, Innovation and Universities under grants TIN2016-78799-P (AEI/FEDER, UE), RTI2018-096384-B-I00, RTI2018-098156-B-C53 and RTC-2017-6389-5.

References

[1] M. M. Waldrop, The chips are down for moores law, Nature News 530 (2016) 144.

[2] J. M. Cecilia, I. Timón, J. Soto, J. Santa, F. Pereñíguez, A. Muñoz, High-throughput infrastructure for advanced its services: A case study on air pollution monitoring, IEEE Transactions on Intelligent Transportation Systems 19 (2018) 2246–2257.

[3] D. Singh, C. K. Reddy, A survey on platforms for big data analytics, Journal of big data 2 (2015) 8.

[4] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference., 2015.

[5] ARMLimited, ARM NEON Technology, 2012.

[6] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, P. Walker, The ARM Scalable Vector Extension, IEEE Micro 37 (2017) 26–39.

[7] A. Sodani, Knights landing (KNL): 2nd Generation Intel Xeon Phi processor, in: Hot Chips.

[8] T. Yoshida, Introduction of Fujitsu's HPC processor for the Post-K computer, in: Hot Chips.

[9] NEC, Vector Supercomputer SX Series: SX-Aurora TSUBASA, 2017.

[10] S. A. Wright, Performance modeling, benchmarking and simulation of high performance computing systems, Future Generation Computer Systems 92 (2019) 900 – 902.

[11] I. Gelado, M. Garland, Throughput-oriented gpu memory allocation, in: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, ACM, pp. 27–37.

[12] Tan, P.,Steinbach, M., Kumar, V., Introduction to Data Mining, Addison Wesley, 2006.

[13] Jain, A., Murty, M. y Flynn, P., Data clustering: a review, ACM computing surveys (CSUR) 31 (1999) 264–323.

[14] J. Lee, B. Hong, S. Jung, V. Chang, Clustering learning model of cctv image pattern for producing road hazard meteorological information, Future Generation Computer Systems 86 (2018) 1338–1350.

[15] A. Bueno-Crespo, J. Soto, A. Muñoz, J. M. Cecilia, Air-pollution prediction in smart cities through machine learning methods: A case of study in murcia, spain., Journal of Universal Computer Science 24 (2018) 261–276.

[16] A. Pérez-Garrido, F. Girón-Rodríguez, A. Bueno-Crespo, J. Soto, H. Pérez-Sánchez, A. M. Helguera, Fuzzy clustering as rational partition method for qsar, Chemometrics and Intelligent Laboratory Systems 166 (2017) 1–6.

[17] C. K. R. E. Charu C. Aggarwal, Data Clustering: Algorithms and Applications, Chapman and Hall/CRC Data Mining and Knowledge Discovery Series, CRC, 2013.

[18] I. S. Dhillon, D. S. Modha, A data-clustering algorithm on distributed memory multiprocessors, in: M. J. Zaki, C.-T. Ho (Eds.), Large-Scale Parallel Data Mining, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 245–260.

[19] H. S. Nagesh, S. Goil, A. Choudhary, A scalable parallel subspace clustering algorithm for massive data sets, in: Proceedings 2000 International Conference on Parallel Processing, pp. 477–484.

[20] Bezdek, J., Ehrlich, R. y Full, W., FCM: The Fuzzy C-Means clustering algorithm, Computers & Geosciences 10 (1984) 191–203.

[21] T. Kwok, K. Smith, S. Lozano, D. Taniar, Parallel fuzzy c- means clustering for large data sets, in: B. Monien, R. Feldmann (Eds.), Euro-Par 2002 Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 365–374.

[22] A. Ravi, A. Suvarna, A. D'Souza, G. Ram Mohana Reddy, Megha, A parallel fuzzy c means algorithm for brain tumor segmentation on multiple mri images, in: A. Kumar M., S. R., T. V. S. Kumar (Eds.), Proceedings of International Conference on Advances in Computing, Springer India, New Delhi, 2012, pp. 787–794.

[23] S. Rahimi, M. Zargham, A. Thakre, D. Chhillar, A parallel fuzzy c-mean algorithm for image segmentation, in: IEEE Annual Meeting of the Fuzzy Information, 2004. Processing NAFIPS '04., volume 1, pp. 234–237 Vol.1.

[24] M. V. Modenesi, M. C. A. Costa, A. G. Evsukoff, N. F. F. Ebecken, Parallel fuzzy c-means cluster analysis, in: M. Daydé, J. M. L. M. Palma, Á. L. G. A. Coutinho, E. Pacitti, J. C. Lopes (Eds.), High Performance Computing for Computational Science - VECPAR 2006, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 52–65.

[25] T. C. Havens, J. C. Bezdek, C. Leckie, L. O. Hall, M. Palaniswami, Fuzzy c-means algorithms for very large data, IEEE Transactions on Fuzzy Systems 20 (2012) 1130–1146.

[26] Flores-Sintas, A., Cadenas, J.M. y Martn, F., A local geometrical properties application to fuzzy clustering, Fuzzy Sets and Systems 100 (1998) 245–256.

[27] Soto, J., Flores-Sintas, A. y Palarea-Albaladejo, J., Improving probabilities in a fuzzy clustering partition, Fuzzy Sets and Systems 159 (2008) 406–421.

[28] I. Timn, J. Soto, H. Prez-Snchez, J. M. Cecilia, Parallel implementation of fuzzy minimals clustering algorithm, Expert Systems with Applications 48 (2016) 35 – 41.

[29] Flores-Sintas, A., Cadenas, J.M. y Martn, F., Detecting homogeneous groups in clustering using the euclidean distance, Fuzzy Sets and Systems 120 (2001) 213–225.

[30] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al., In-datacenter performance analysis of a tensor processing unit, in: 2017 ACM/IEEE 44th

Annual International Symposium on Computer Architecture (ISCA), IEEE, pp. 1–12.

[31] Top500, Top500 supercomputer site, `http://www.top500.org/`, 2017. (accessed, April, 3th, 2017).

[32] T. Austin, Bridging the Moore's Law Performance Gap with Innovation Scaling, in: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, ACM, 2015, p. 1.

[33] Hennessy, Computer architecture: a quantitative approach, 3rd ed., Morgan Kauffman, 2003.

[34] 39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA, IEEE Computer Society, 2012.

[35] D. L. Caballero de Gea, PhD Thesis: SIMD@OpenMP : a programming model approach to leverage SIMD features, 2015.

[36] OpenMP Architecture Review Board, The OpenMP Specification, `http://www.openmp.org`, 2017. (accessed, April, 2th, 2017).

[37] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, D. K. Panda, MVAPICH2-GPU: optimized GPU to GPU communication for infiniband clusters, Computer Science-Research and Development 26 (2011) 257.

[38] D. R. Kaeli, P. Mistry, D. Schaa, D. P. Zhang, Heterogeneous Computing with OpenCL 2.0, Morgan Kaufmann, 2015.

[39] D. B. Kirk, W. H. Wen-mei, Programming massively parallel processors: a hands-on approach, Elsevier, 2013.

[40] G. Guennebaud, B. Jacob, et al., Eigen v3, http://eigen.tuxfamily.org, 2010.

[41] A. Yasin, A top-down method for performance analysis and counters architecture, in: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 35–44.

[42] E. Kaltofen, G. Villard, On the complexity of computing determinants, computational complexity 13 (2005) 91–130.

[43] A. A. Hasib, J. M. Cebrian, L. Natvig, V-pfordelta: Data compression for energy efficient computation of time series, in: 22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015.

[44] A. Fog, Instruction Tables. Instruction latencies, throughputs and micro-operation breakdowns., 2018.

[45] S. C. Johnson, Hierarchical clustering schemes, Psychometrika 32 (1967) 241–254.

[46] L. Kaufman, P. Rousseeuw, Finding Groups in Data: An Introduction to Cluster Analysis, John Wiley & Sons, 1990.

[47] A. Struyf, M. Hubert, P. Rousseeuw, Clustering in an object-oriented environment, Journal of Statistical Software, Articles 1 (1997) 1–30.

[48] A. Saxena, M. Prasad, A. Gupta, N. Bharill, O. P. Patel, A. Tiwari, M. J. Er, W. Ding, C.-T. Lin, A review of clustering techniques and developments, Neurocomputing 267 (2017) 664–681.

[49] C. Jin, M. M. A. Patwary, A. Agrawal, W. Hendrix, W.-k. Liao, A. Choudhary, Disc: A distributed single-linkage hierarchical clustering algorithm using mapreduce, work 23 (2013) 27.

[50] X. Hou, An improved k-means clustering algorithm based on hadoop platform, in: The International Conference on Cyber Security Intelligence and Analytics, Springer, pp. 1101–1109.

[51] Q. Zhao, Y. Shi, Z. Qing, Research on hadoop-based massive short text clustering algorithm, in: Fourth International Workshop on Pattern Recognition, volume 11198, International Society for Optics and Photonics, p. 111980A.

[52] H. Xiong, K-means image classification algorithm based on hadoop, in: Recent Developments in Intelligent Computing, Communication and Devices, Springer, 2019, pp. 1087–1092.

[53] A. Woodley, L.-X. Tang, S. Geva, R. Nayak, T. Chappell, Parallel k-tree: A multicore, multinode solution to extreme clustering, Future Generation Computer Systems 99 (2019) 333–345.

[54] W. Kwedlo, P. J. Czochanski, A hybrid mpi/openmp parallelization of k-means algorithms accelerated using the triangle inequality, IEEE Access 7 (2019) 42280–42297.

[55] Y. Li, K. Zhao, X. Chu, J. Liu, Speeding up k-means algorithm by gpus, Journal of Computer and System Sciences 79 (2013) 216–229.

[56] V. Saveetha, S. Sophia, Optimal tabu k-means clustering using massively parallel architecture, Journal of Circuits, Systems and Computers 27 (2018) 1850199.

[57] Y. Djenouri, D. Djenouri, A. Belhadi, A. Cano, Exploiting gpu and cluster parallelism in single scan frequent itemset mining, Information Sciences 496 (2019) 363–377.

[58] B. Krawczyk, Gpu-accelerated extreme learning machines for imbalanced data streams with concept drift, Procedia Computer Science 80 (2016) 1692–1701.

[59] S. Karbhari, S. Alawneh, Gpu-based parallel implementation of k-means clustering algorithm for image segmentation, in: 2018 IEEE International Conference on Electro/Information Technology (EIT), IEEE, pp. 0052–0057.

[60] S. V. Jagtap, Y. Rao, Clustering and parallel processing on gpu to accelerate circuit transient analysis, in: International Conference on Advanced Computing Networking and Informatics, Springer, pp. 339–347.

[61] Y. Fang, Q. Chen, N. Xiong, A multi-factor monitoring fault tolerance model based on a gpu cluster for big data processing, Information Sciences 496 (2019) 300–316.

[62] S. Tanweer, N. Rao, Novel algorithm of cpu-gpu hybrid system for health care data classification, Journal of Drug Delivery and Therapeutics 9 (2019) 355–357.

[63] C. Scully-Allison, R. Wu, S. M. Dascalu, L. Barford, F. C. Harris, Data imputation with an improved robust and sparse fuzzy k-means algorithm, in: 16th International Conference on Information Technology-New Generations (ITNG 2019), Springer, pp. 299–306.