Abstract: Clustering is a classification method that organizes objects into groups based on their similarity. Data clustering can extract valuable information, such as human behavior, trends, and so on, from large datasets by using either hard or fuzzy approaches. However, this is a time-consuming problem due to the increasing volumes of data collected. In this context, sequential executions are not feasible and their parallelization is mandatory to complete the process in an acceptable time. Parallelization requires redesigning algorithms to take advantage of massively parallel platforms. In this paper we propose a novel parallel implementation of the fuzzy minimals algorithm on graphics processing unit as a high-performance low-cost solution for common clustering issues. The performance of this implementation is compared with an equivalent algorithm based on the message passing interface. Numerical simulations show that the proposed solution on graphics processing unit can achieve high performances with regards to the cost-accuracy ratio.

Keywords: fuzzy clustering; parallel computing; fuzzy minimals algorithm; GPU; MPI

1. Introduction

Technological advances have allowed the collection of large amounts of data. Published forecasts indicate that a significant growth in the amount of data is expected for the next years—for instance, the Internet traffic in 2020 was circa 95 times the volume from 2005. A detailed analysis of that data can extract valuable information about them. In fact, large datasets have been widely used to identify trends, recommend products or services to provide a customized experience that matches the interests of users [1]. These applications are made feasible through classification algorithms.

Classification algorithms organize objects from datasets to provide a meaningful understanding, that is, learning, about key factors of the data. If the data is partially labeled, organizing it can be referred as supervised learning. On the other hand, unsupervised learning is defined when no prior information is available. Unsupervised learning, which is the most challenging approach, is commonly studied by clustering algorithms. Data clustering (or cluster analysis) is the task of grouping objects into clusters (groups) based on their similarity. Intuitively, objects within the same group share more common features among themselves than objects from other groups. Clustering has a wide range of applications, e.g. biology, physics, economics, chemistry, etc. [2–6]. Organizing objects from a dataset into clusters require identifying their fundamental features that make themselves similar or not to each other. For instance, cars can be grouped based on their power, fuel consumption, price, size, etc. Each feature receives a numerical value, which is stored in a vector that represents a particular object. Objects can be compared using distance metrics among their vectors, meaning that close objects are similar to each other.

Whenever classical mathematical logic cannot model a system with sought accuracy, we can use fuzzy sets. Fuzzy sets and fuzzy logic are a form of infinite-valued logic that
can be used to solve a large amount of computing problems, including clustering and image analysis. The birth of fuzzy logic is traced back to Zadeh in 1965 [7]. His main idea was to introduce uncertainty in logic states. In fact, physical systems cannot be described solely by states such as true or false. This directly implies the concept of membership, where categories (e.g., hot, warm and cold) are defined, and a system status can have different degrees of membership of these categories (e.g., warm, extremely cold, etc.). The membership is usually expressed by smooth or continuous functions, mostly of regular shapes (e.g., triangular or trapezoidal shapes). Application of fuzzy logic is based in two main concepts: fuzzification and granulation. The fuzzification is the act of replacing a set with sharply defined boundaries with one with fuzzy boundaries. On the other hand, the granulation means partitioning an object into a collection of granules of related points.

In science and engineering, several processes can be much better described with such vague definitions. Fuzzy models are widely used in traffic signs control [8], pattern recognition [9], decision planning [10], etc. The methods based on classic clustering assign each object to a single cluster. This approach can imply some difficulties when clusters are not well delimited (e.g., image processing, pattern recognition, etc.) and overlapping clusters are present. In fuzzy clustering algorithms, both of these two problems can be solved by the introduction of a membership function that allows assigning objects to multiple clusters.

Clustering has been known for decades as a computationally difficult (NP-hard) problem [11]. In general, clustering algorithms are based on the minimization of an objective function that measures similarities among vector attributes representing objects. Moreover, data volumes are growing fast and impacting significantly their performance. In this scenario, serial executions are not feasible, and their parallelization is mandatory. These issues are often solved using algorithms based on K-Means [12,13], the most popular hard partition algorithm. Many parallel versions were proposed, and the initial solutions were based on the message passing method. Quite recently, new solutions have been based on other techniques and platforms, e.g., OpenMP, GPUs, and Intel Xeon Phi [14]. Likewise, fuzzy clustering algorithms also demand parallel solutions to explore large datasets. The most adopted fuzzy clustering algorithm has an extensive literature about its parallelization [15,16]. However, these proposals do not change the requirements of this algorithm, i.e., a prior knowledge about the number of groups in the dataset. This has an exponential impact on the performance with the problem size, since several executions are needed to get the sought solution.

In order to overcome these issues, an algorithm called fuzzy minimals (FM) was proposed in [17]. This algorithm uses fuzzy sets theory to find clusters without any prior knowledge about the number of groups, the size or format of the dataset. Furthermore, the authors proved that this solution meet all requirements that a good classification algorithm should have (e.g., scalability, adaptability, stability, etc). Besides the different approaches for clusterization that have been proposed, the technological advances in microelectronics led to improvements in graphical accelerators. These improvements, in a relatively short period of time, became the driving force behind thousands heavy computing problems. Some of these applications based on fuzzy sets, provides solutions for image segmentation to help medical doctors diagnosing illnesses, simulation of wildfires, modeling geological phenomena, etc. On the other hand, other applications have only explored GPU’s capabilities to speedup their solution up to a few hundred times their sequential code [18,19].

In this paper, we consider the introduction of two novel parallel implementations of the FM, which addresses frequent challenges in clustering algorithms [20–23]. More precisely, our study is focused on the parallel implementation on GPU, exploiting its massively parallel architecture to deliver high performance at a low cost. We compare this solution both with the traditional sequential version and with a new parallel version using MPI. Numerical simulations showed that our solution on GPU provides a significant
speedup over the sequential version. In fact, the found solution on GPU gives results at least as fast as the expensive computing environments needed to run the MPI version.

The remainder of this paper is organized as follows. Section 2 presents some preliminaries on the FM algorithm and GPU. Section 3.1 is intended to motivate our parallel fuzzy solution on GPU. Section 4 shows the numerical results with a synthetically developed dataset and two real-world benchmarks. Finally, Section 5 outlines the main results of this paper, open problems and possible future developments.

2. Preliminaries

This section contains some general remarks on the FM algorithm and GPU technology. Thus, the next sections will be rendered as self-contained as possible in order to facilitate access to the individual topics.

2.1. Fuzzy minimals algorithm

FM is an algorithm for data clustering that avoids the need of previous knowledge about groups in the data. Proposed by Flores-Sintas, Cadenas and Martin [17], the combined metrics extracted from the Euclidean distance matrix and the inverse covariance matrix, making it possible to deduce a membership function to a group independent of the membership probability to other groups [17,24].

The first step of the algorithm consists of finding a metric related to the covariance matrix. This metric provides information about the homogeneity and isotropy of the feature space. We denote by $E$ the operator of statical mean. Thus, the elements of the covariance matrix are defined as follows:

$$g_{jk}(m) = E\left( (z^j - m^j)(z^k - m^k) \right), \quad (1)$$

where $M$ is the vector of the average samples. We note that (1) and the Mahalanobis distance for the all feature vectors in the sample allow us to determine a diagonalized covariance matrix of general element given by

$$g_{\alpha\beta}(x) = \delta_{\alpha\beta}c_\beta + x_\alpha x_\beta, \quad (2)$$

where $X$ is a vector. Of course, it follows from the tensorial form of (1). The covariance matrix defined by (2) is such that

$$g(x) = \frac{g(m)}{1 + q_k(x^k)^2}$$

where $q_k$ are the diagonal elements of the covariant metric tensor.

Now, we can define a metric for each point of discourse, which is associated with a Riemannian manifold hypersurface. The local properties of the hypersurface, and therefore the distance relationship between sample points, are given by the Christoffel symbols:

$$\Gamma^i_{kj} := -x^i g_{kj}.$$  

We can determine the algorithm’s objective function as well as the membership function. First, a sample of $n$ data points, displaying $F$ features, is given by $X = \{x_1, x_2, ..., x_n\} \subset \mathbb{R}^F$. Accordingly, the FM algorithm minimizes the objective function given by

$$J(v) = \sum_{x \in \mathcal{X}} \mu_{x\theta} d_{x\theta}^2,$$
where \( d_{xv}^2 \) is the Euclidean norm between two points \( x \) and \( v \), and \( \mu_{xv} \) is the membership function between an object \( x \) and the prototype \( v \). The membership function is given by

\[
\mu_{xv} = \frac{1}{1 + r^2 d_{xv}^2},
\]

where the factor \( r \) measures the isotropy in the dataset. Thus, \( r \) indicates the disruptions in the data that should result in new clusters. Moreover, we recall that the factor \( r \) can be computed by

\[
\sqrt{|C^{-1}|} \sum_{x \in X} \frac{1}{1 + r^2 d_{xm}^2} = 1,
\]

where \( C \) is the covariance matrix, \( m \) is the mean of \( X \) and \( d_{xm}^2 \) is the Euclidean norm between \( x \) and \( m \).

The FM algorithm is an iterative process that works on clusters representatives, called prototypes, given by

\[
v = \frac{\sum_{x \in X} \mu_{xv}^2 x}{\sum_{x \in X} \mu_{xv}^2}.
\]

Algorithm 1 describes the implementation of the FM algorithm. More precisely, we assume that two standard parameters, \( \varepsilon_1 \) and \( \varepsilon_2 \), indicate the committed error degree and the minimum difference between two potential prototypes, respectively. The minimum values of the objective function are the prototypes of each identified cluster. We note that there is no need to specify the number of clusters (being based on the computation of \( r \)).

**Algorithm 1 Fuzzy Minimals.**

**Input:** dataset \( X \), factor \( r \) and \( N \) objects

**Output:** prototypes found

**procedure** FuzzyMinimals\((X, r, N)\)

\[ V = \{\}\subset \mathbb{R}^F \]

for \( k = 1 \) until \( N \) do

\[ v_0 = x_k; t = 0; E_0 = 1 \]

while \( E(t) \geq \varepsilon_1 \) do

\[ t = t + 1 \]

\[ \mu_{xv} = \frac{1}{1 + r^2 d_{xv}^2}, \text{ using } v_{t-1} \]

\[ v_t = \frac{\sum_{x \in X} \mu_{xv}^2 x}{\sum_{x \in X} \mu_{xv}^2} \]

\[ E(t) = \sum_{\alpha=1}^{F} (v_{\alpha}^{t} - v_{\alpha}^{t-1})^2 \]

end while

if \( \frac{\sum_{\alpha=1}^{F} (v_{\alpha}^{t} - w_{\alpha})^2}{\varepsilon_2}, \forall v \in V \) then

\[ V = V + \{v\} \]

end if

end for

return \( V \)

**end procedure**

2.2. Graphics Processing Unit

GPUs have evolved from simple configurable graphics processors to high performance programmable units that can process complex graphical operations and, more in general, any massively parallel computation.
The current performance and low cost of GPUs led researchers and scientists to propose solutions based on this platform. Particularly, applications that demand high computational resources (e.g., clustering, pattern recognition, differential equation solvers, etc.) are strong candidates for parallelization on GPUs. In general, a GPU is composed by a set of streaming multiprocessors (SM), each of which has a significant number of computing cores (e.g., Nvidia Titan X).

There are several memory levels available on GPUs (i.e., global memory, local memory, shared memory and registers). In particular, each SM provides two high performance on-chip memories (registers and shared memory). Registers are privately used by threads but the number of registers per thread is limited. On the other hand, a shared memory address can be accessed by threads within the same block of threads, allowing their synchronization. Likewise, the maximum amount of shared memory per SM is small.

The processing in the cores occurs based on threads, which are grouped together into a block, while blocks are grouped into a grid. Only threads within the same block can communicate to each other via shared memory or synchronize their execution with barriers, because each block resides on its own SM. Threads from different blocks can do these operations via global memory, but that is a not a recommended solution. The introduction of Nvidia’s CUDA (Compute Unified Device Architecture) to their enabled GPUs allowed developers to design any general-purpose applications. A typical program using CUDA has a basic structure. Initially, the memory is allocated on GPU. Fist, data are copied from host (CPU) to device (GPU). Next, a kernel (function that runs on the device) is called passing the number of blocks and threads per block. When the kernel completes, data can be copied back from device to host. Finally, the memory allocated in the device is released.

3. Parallel FM

In this section, we show the methodology used for the parallelization of FM using GPU and MDPI.

3.1. Parallel FM on GPU

Our parallel solution on GPU can be divided into four high level phases. In the first phase (determination of factor R), we are the preprocessing phase. We determine the value of the factor R, i.e., the global parameter that measures the isotropy of the whole dataset. It is worth noticing that R is invariant with the kind of execution. In the second phase (data-partitioning), the dataset is divided into multiple partitions, where each partition has above 10% of the original dataset. Moreover, data are distributed following an uniform random distribution across all partitions. In the third phase (parallel FM), each block of threads processes a single data-partition, computing the prototypes and their initial clusters. This is the only phase executed on GPU. Next, a kernel (function that runs on the device) is called passing the number of blocks and threads per block. When the kernel completes, data can be copied back from device to host. Finally, the memory allocated in the device is released.

Hierarchical clustering does not provide a single final answer, but rather a tree-like structure called dendrogram. This flexible structure allows choosing clusters based on the level of the dendrogram. As the prototypes get closer, we see at once the level that should be used to produce the final results.

The Algorithm 2 shows the main program of our implementation. The program begins with the declaration of the blocks and threads per block that will be used. Next, the program reads all objects, storing them in a local variable X. This vector is organized into b logical partitions, and data are distributed uniformly across them (to identify cluster prototypes). The device does not have direct access to the host memory and vice versa. Therefore, we need to allocate a global memory space on the device and copy the data explicitly. Now, all the data are ready, thus the kernel PFMGPU is called, using the triple angle brackets syntax to indicate the number of blocks and threads per block. The parameters for this function
are the factor \( R \), a pointer to the vector \( \text{dev}X \), storing input data in the global memory, and a pointer to the vector \( \text{dev}V \) that will store identified prototypes. At the end of the kernel, the prototypes found are copied back to the host memory. Finally, the algorithm performs a hierarchical clustering of prototypes, showing the results and releasing the allocated memory.

**Algorithm 2** PFMGPU’s main program.

```plaintext
procedure MAIN
    b = number of blocks
    t = number of threads per block
    X = objects
    r = factor R
    Distribute data uniformly in \( b \) partitions
    \( \text{dev}X = \) pointer to global memory region storing input data
    \( \text{dev}V = \) pointer to global memory region storing output data
    Copy input from host memory stored in \( X \) to global memory in \( \text{dev}X \)
    \( \text{PFMGPU} \langle b, t \rangle (r, \text{dev}X, \text{dev}V) \)
    Copy output from global memory stored in \( \text{dev}V \) to host memory in \( V \)
    C = Hierarchical clustering of \( V \)
    Show \( C \) prototypes
    Free device memory
end procedure
```

Clearly, our method parallelizes threads in a GPU. As shown in Figure 1, each block from the complete dataset \( X \) has partition boundaries, based on \( b \) blocks, where \( t \) threads from the same multiprocessor can be executed. Threads in the same block have unique identifiers, thus we assign them to corresponding partition locations. Next, a parallel execution allows us to identify prototype candidates. The result is stored in the shared array \( V_s \), which is visible to all threads from the same block. When the iteration completes for that location, we skip \( n \) positions equal to the number of the threads to avoid reprocessing.

![Figure 1. GPU threads running in parallel.](image)

Algorithm 3 shows the kernel to compute the prototypes. We note that all objects are passed as parameters, thus any block identifies its partition. Moreover, each thread has to determine their start and end positions. Accordingly, there is no overlapping. These operations can be performed with CUDA’s built-in variables, i.e., `threadIdx` and `gridDim` which are the thread identifier in the block and the number of blocks in the grid, respectively. We declare the shared variables \( V_s, \text{tmp} \) and \( \text{size} \) on line 6, hence the threads within the same block can exchange data, avoiding global memory usage. Furthermore, the prototypes found in each data-partition are stored in \( V_s \), where \( \text{tmp} \) keeps prototype candidates of any thread and \( \text{size} \) contains the total number of prototypes found in the partition. All local variables are declared on lines 8 and 9. We use an inner loop, between lines 10 and 17, to
find prototype candidates. The prototype candidate found by any thread is stored in the shared array \( tmp \) on line 18. A barrier is created using \( \text{syncthreads} \). The loop between lines 21 and 26 verifies if the prototype candidate is not close to any previously added prototype. Before the kernel completes, the prototypes found are copied from shared memory to global memory. Therefore, the host can reference the prototypes stored in \( V \).

**Algorithm 3** FM kernel on GPUs.

**Input:** factor \( r \), dataset \( X \) and \( V \) for prototypes. **Output:** all kernels return void

**procedure** \( \text{PFMGPU}(r, X[ ], V[ ]) \)

\[
\begin{align*}
id &= \text{threadIdx} \dim \text{gridDim} \\
\text{start} &= \text{initial index of the partition} \\
\text{end} &= \text{final index of the partition} \\
\_\text{shared}_\_ \ V_s[ ], \ tmp[ ], \ size &= 0 \\
\text{for } k = \text{start} + \text{id}; \ k < \text{end}; \ k += \dim \ do \\
v_0 &= X[k] \\
t &= 0; \ E_0 = 1 \\
\text{while } E_t(1) \geq \epsilon_1 \ do \\
\mu_{x_0} &= \frac{1}{1 + r^2 d_{x_0}^2}, \text{ using } v_{t-1} \\
\sum_{x \in X} \mu_{x_0}^2 \\
v_t &= \sum_{x \in X} \mu_{x_0}^2 \\
E_t &= \sum_{a=1}^C (v_{(t)}^a - v_{(t-1)}^a)^2 \\
\text{end while} \\
\text{tmp}[\text{id}] &= v_{(t)} \\
\_\text{syncthreads}() \\
\text{if } \text{id} == 0 \ then \\
\text{for } i = 0 \ until \ \dim \ do \\
\text{if } \sum_{a=1}^C (\text{tmp}[i]^a - w^a)^2 > \epsilon_2, \forall w \in V_s \ then \\
V_s = V_s + \{ \text{tmp}[i] \} \\
size &= size + 1 \\
\text{end if} \\
\text{end for} \\
\_\text{syncthreads}() \\
\text{for } i = \text{id}; \ i < \text{size}; \ i += \dim \ do \\
V[i] = V_s[i] \\
\text{end for} \\
\_\text{syncthreads}() \\
\text{end if} \\
\text{end for} \\
\_\text{syncthreads}() \\
\text{end procedure}
\]

3.2. Parallel FM on MPI

Our parallel implementation using MPI is similar in spirit to that based on GPU. Thus, we have again the four phases described for the parallel FM on GPU (i.e., determination of factor \( R \), data-partitioning, parallel FM and hierarchical clustering). However, the final implementation using MPI is more straightforward than the version in GPU. Indeed, the mapping between the sequential and the parallel version is minimal.

Algorithm 4 shows the main program of our MPI implementation. First, we define the number of processes (i.e., the number of data-partitions) and the rank of the current process. Next, a coordinator process reads the objects and computes the factor \( R \), distributing the data across \( p \) logical partitions. Moreover, the coordinator process sends them to the peer
processes. All processes run the FM function by Algorithm 1. Finally, the peers complete their execution sending the result to the coordinator process. Now, the coordinator process can perform the hierarchical clustering of prototypes.

Algorithm 4 Main program in PFMMPI.

procedure MAIN
    \( p \) = number of processes
    \( rank \) = process identifier
    if \( rank == 0 \) then
        \( X \) = objects
        \( r \) = factor R
        Dist. objects uniformly in \( p \) partitions
        \( size = \frac{n}{p} \)
        for \( k = 1 \) until \( k < p \) do
            Send\((r, size, X[size * k])\)
        end for
    else
        Receive\((r, size, X)\)
    end if
    \( V = \text{FuzzyMinimals}(X, r, size) \)
    if \( rank > 0 \) then
        Send\((V)\)
    else
        for \( k = 1 \) until \( k < p \) do
            \( V = V + \text{Receive}(V) \)
        end for
    end if
    if \( rank == 0 \) then
        \( C = \text{Hierarchical clustering of } V \)
        Show \( C \) prototypes
    end if
end procedure

4. Numerical simulations and results

In this section, we analyze the performance of parallel FM algorithm. Obviously, we do not need to deal with correctness and quality of the FM algorithm already discussed in [24,25]. The results of this section involve a synthetically developed dataset and two real-world benchmarks.

Most popular fuzzy clustering algorithms rely on input parameters to find clusters. Thus, these algorithms require several executions to achieve the same performance of the FM algorithm. Comparison between the accuracy of the FM algorithm and other similar techniques has been extensively studied in recent years. Since the number of executions required by other solutions depend on input parameters provided by the user, we cannot perform a fair runtime comparison between FM algorithm and other similar techniques. Accordingly, our benchmarks are based on different implementations of the FM algorithm.

All results for PFMMGPU were collected on a single machine, using from one to eight blocks with 960 threads per block. In each run, the number of blocks matches the number of data-partitions. On the other hand, PFMMPI was executed on a cluster, using from one to eight nodes with one process per machine. Moreover, the number of processes matches the number of data-partitions. All machines (i.e., single and cluster machines) have the same configuration: CPU Intel Core i7 3.4Ghz, 16GB of RAM and GPU Nvidia GeForce GTS 450 with 4 SMs. Our goal is to compare a GPU-based implementation, based on massive parallelism of GPU cores, with a message passing implementation running on individually more powerful CPUs.
4.1. Ellipses dataset

The first dataset consists of three synthetically generated ellipses with a total of 10 thousand points, as shown in Figure 2. We benchmarked four different test cases for both PFMGPU and PFMMPI. Moreover, the dataset is divided according to the number of blocks (PFMGPU) or processes (PFMMPI). We note that executions with a similar number of partitions provide similar performance, hence sequential and parallel executions with a single partition found the same prototypes. Under the hypothesis of same partitions, multi-partition executions in PFMGPU or PFMMPI should also return the same output (clusters). Nevertheless, it turns out that different number of partitions result in different prototypes. These differences vanish with the hierarchical clustering for multi-partition executions.

Figure 2. Synthetically generated ellipses in a Cartesian coordinate system \((x, y)\).

Figure 3 shows a dendrogram generated from the found prototypes when the dataset is divided into four partitions. The coordinates of each cluster prototype are represented with respect to the \(x\)-axis. The same dendrogram is generated from both PFMGPU and PFMMPI with four partitions. In fact, the output in the two cases is the same. We note that three main clusters, at the dashed level, are identified. After hierarchically clustering prototypes, all parallel executions found three clusters as expected. Table 1 shows all prototypes found by each execution. We see that PFMGPU and PFMMPI return the same prototypes under the same number of partitions.

Table 2 shows execution times collected for the ellipsis benchmark. Running on a single partition, PFMGPU is about three times faster than both the sequential version and PFMMPI. However, the highest performance is achieved for PFMGPU with eight partitions. We see that PFMMPI also achieves the best performance with eight partitions, but its runtimes are higher with respect to PFMGPU. This depends on the method of work division adopted. We have also computed the MPI communication times using \(MPI_{Wtime}\) function, as shown in Table 2. This is due to the fact that communication overhead can influence the total runtimes. We note that the time spent on communication for PFMMPI takes place during the transfer of objects to/from peer nodes. Nevertheless, the impact is minimal on the final result due to the dataset file size. Moreover, the communication time is proportional to the number of processes, in spite of decreasing the amount of
data transferred to each host. This is due to on the overhead of connection functions, independent of the amount of data transferred. The performance of PFMMPI with a single partition is equivalent to the sequential version. In fact, both versions run on a single machine and the overhead due to MPI libraries is negligible.

Table 1. Prototypes found for the ellipsis benchmark.

| Execution | Number of Partitions | Coordinates of the Prototypes                                          |
|-----------|----------------------|-----------------------------------------------------------------------|
| Sequential| 1                    | (407.84, 395.64), (393.62, 1084.59), (1125.10, 749.52)                |
| Parallel  | 1                    | (407.84, 395.64), (393.62, 1084.59), (1125.10, 749.52)                |
| Parallel  | 2                    | (396.85, 394.44), (406.28, 1098.51), (1127.12, 759.52)               |
| Parallel  | 4                    | (359.52, 409.88), (391.70, 1099.75), (1124.16, 761.82)               |
| Parallel  | 8                    | (361.11, 403.26), (350.18, 1105.10), (1111.99, 749.87)               |

Table 2. Runtimes for the ellipsis benchmark.

| Execution | Partitions | Time       | MPI Comm. |
|-----------|------------|------------|-----------|
| Sequential| 1          | 170.596 s  | –         |
| PFMGPU    | 1          | 50.330 s   | –         |
| PFMGPU    | 2          | 10.282 s   | –         |
| PFMGPU    | 4          | 3.582 s    | –         |
| PFMGPU    | 8          | 1.706 s    | –         |
| PFMMPI    | 1          | 170.664 s  | 0.0 s     |
| PFMMPI    | 2          | 47.395 s   | 0.01763 s |
| PFMMPI    | 4          | 11.341 s   | 0.04424 s |
| PFMMPI    | 8          | 3.020 s    | 0.10290 s |

4.2. Well-defined schools dataset

Figure 4 shows our second benchmark, that consists of 20,237 geographical coordinates of schools extracted from U.S. Board on Geographic Names [26]. This dataset contains five different clusters. We doubled the number of objects in comparison to the first benchmark.
At this scale, the prototypes found by different executions appear to be the same, but as illustrated in Figure 6a–d, respectively. We note that the prototypes found by different executions appear to be the same. However, a more detailed analysis turns out that prototypes are slightly different. Accordingly, a minimal variation between executions with different partitions provide a high degree of precision.

In Table 3, the runtimes for this benchmark are shown. We see at once that it achieved lower execution times. Thus, the performance does not depend uniquely on the number of objects, but it also take into account other factors (density, number of clusters, distances, etc.). Despite running on a GPU with only four SMs, PFMGPU showed lower execution times. More precisely, the eight-partition execution of PFMGPU was faster than PFMMPI running on eight nodes. The time spent on communication for PFMMPI increased from the previous experiment. Nevertheless, it has a minimal impact on the performance.

**Figure 4.** Geographical coordinates of 20,237 USA schools (partial schools dataset).

**Figure 5.** Hierarchical clustering of prototypes.

**Figure 6.** Prototypes found for the partial schools dataset.
### Table 3. Runtimes for the partial schools dataset.

| Execution  | Partitions | Time   | MPI Comm. |
|------------|------------|--------|-----------|
| Sequential | 1          | 52.683 s | –         |
| PFMGPU     | 1          | 20.596 s | –         |
| PFMGPU     | 2          | 5.468 s  | –         |
| PFMGPU     | 4          | 1.512 s  | –         |
| PFMGPU     | 8          | 0.973 s  | –         |
| PFMMPI     | 1          | 52.720 s | 0.0 s     |
| PFMMPI     | 2          | 13.739 s | 0.039351 s|
| PFMMPI     | 4          | 3.667 s  | 0.121386 s|
| PFMMPI     | 8          | 1.203 s  | 0.249457 s|

#### 4.3. Complete schools dataset

Our final simulation is also based on data from U.S. Board on Geographic Names [26]. More precisely, we analyzed 40,646 school locations across USA (see Figure 7). We note that these data has no well-defined boundaries among clusters.

![Figure 7. Geographical coordinates of 40,646 USA schools (complete schools dataset).](image)

Figure 8a–c show dendrograms found for parallel executions with two, four and eight partitions, respectively. The coordinates of each cluster prototype are represented with respect to the x-axis. We see at once that all executions found five different clusters at the dashed level. The prototypes found are located in high density areas. Hence, the prototypes do not have to be at the center of the cluster as in K-Means based algorithms. The variation between dendrograms did not influence the precision of the final result. In Figure 9a–d, the prototypes found in tests with one, two, four and eight partitions are shown, respectively. From a clustering standpoint, the differences between prototypes are minimal for three main reasons (high volume of objects, data distribution across partitions and accuracy of our FM parallelization).

![Figure 8. Dendrograms for the complete schools dataset.](image)
As in Section 4.2, the results of our simulation are shown in Table 4. As in previous benchmarks, parallel executions achieved their highest performance with eight partitions (circa 60 times faster than the serial execution). From one to four partitions, PFMGPU was consistently faster than PFMMPI, but the execution with eight partitions gave similar runtimes for both implementations. This depends on the fact that our GPU has only four SMs. Thus, our performance decreases whenever we exceed the available resources. Moreover, the time that PFMMPI spent on communication increased at a lower ratio, which balances the latency overhead.

Table 4. Runtimes for the complete schools dataset.

| Execution | Partitions | Time     | MPI Comm. |
|-----------|------------|----------|-----------|
| Sequential | 1          | 248.284 s | –         |
| PFMGPU    | 1          | 133.945 s | –         |
| PFMGPU    | 2          | 40.166 s  | –         |
| PFMGPU    | 4          | 8.963 s   | –         |
| PFMGPU    | 8          | 4.724 s   | –         |
| PFMMPI    | 1          | 248.447   | 0         |
| PFMMPI    | 2          | 62.510 s  | 0.072839 s|
| PFMMPI    | 4          | 14.974 s  | 0.220595 s|
| PFMMPI    | 8          | 4.323 s   | 0.497454 s|

4.4. Timing analysis

A comparison with other clustering algorithms may not produce realistic or significant results. In fact, the number of executions and thus the total runtime depend on parameters provided by the user. Accordingly, a practical analysis would involve the parallel implementation of the FM algorithm introduced in [25]. However, the authors did not provide the implementation, making impossible any comparison. Consequently our analysis is constrained to the technology and methods shown in in [25]. We can assume that the processor used was an Intel Core i5 clocked at 1.6 Ghz. There are several models for this 4th generation processor. According to the PassMark benchmarks website, the slowest processor from that generation has a rate of 2256, which is about four times slower than the i7-3770 processor used in our MPI implementation. Moreover, the speed was about 1.27 ms/object with 8 partitions, which is much slower than the 0.11 ms/object achieved in our MPI implementation. Therefore, even if the speed is adjusted with the speed of the processors used in each simulation, our approach is at least three times faster. Another
possible solution is that we deal with the efficiency of our parallel implementations by scalability and speedups. These results are shown in Figure 10.

![Speedup GPUs](image1)

![Speedup MPI](image2)

**Figure 10.** Speedups for PFMGPU (top) and PFMMPI (bottom).

We see at once that both implementations are faster than the standard (and single process) implementation. In particular, we found that a speedup that generally quadruples when the number of partitions doubles. This depends on the hardware in use. Indeed, for the PFMGPU implementation we have that each partition is run in a different GPU SM, up to four partitions, and each SM is composed by hundreds of computing cores. On the other hand, for the PFMMPI implementation, although each cluster’s node executes a single MPI process, the CPU is multicore. Moreover, the major bottleneck with GPU is the data transfers between CPU and GPU memories. Therefore, the speedup achieved in the ellipsis test is higher in the case of GPU, as shown in Figure 10.

5. Conclusions

The results of this paper are based on the parallel version of the FM algorithm on GPU (PFMGPU). This approach provided high performance at a low computational cost. More precisely, the use of GPUs allowed the parallelization by a reasonably low-cost parallel hardware, without variations in the accuracy of FM clustering algorithm.
The benchmarks studied in this paper proved that PFMGPU can solve clustering problems several times faster than in a classical CPU. In particular, we achieved a solution at least 50 times faster than the sequential execution in hypothesis of eight partitions. The major drawback of our solution is that each partition must have at least 10% of the data. Thus, we have a strict limitation on the number of partitions. Nevertheless, PFMGPU provided better results than a conventional parallelization via MPI (PFMMPi). This showed that even low-end GPUs can offer better performance. We propose that further research should be undertaken in two directions. Firstly, it is relevant to study the influence of partitions smaller than 10% in the cluster accuracy. Secondly, we need to deal with forms to improve the parallelization inside a partition.

Author Contributions: All authors contributed equally to this article. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: The authors wish to express his gratitude to the reviewers for their constructive comments and suggestions.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations
The following abbreviations are used in this manuscript:

- CUDA: Compute Unified Device Architecture
- GPU: Graphics Processing Unit
- MPI: Message Passing Interface
- FM: Fuzzy Minimals (algorithm)
- PFMGPU: Parallel Fuzzy Minimals on GPU
- PFMMPI: Parallel Fuzzy Minimals on MPI
- SM: Streaming Multiprocessors

References
1. Zhao, L.; Pan, S.J.; Yang, Q. A unified framework of active transfer learning for cross-system recommendation. *Artif. Intell.* 2017, 245, 38–55. [CrossRef]
2. Yenilmez, F.; Girginer, N. Comparison of indicators of women’s labour between Turkey and EU member states by employing multidimensional scaling analysis and clustering analysis. *Appl. Econ.* 2016, 48, 1229–1239. [CrossRef]
3. Pan, L.; Zhang, M. Ju, P.; He, H.; Ishii, M. Vertical co-current two-phase flow regime identification using fuzzy C-means clustering algorithm and ReliefF attribute weighting technique. *Int. J. Heat Mass. Transf.* 2016, 95, 393–404. [CrossRef]
4. Yu, X.; Yuan, F.; Fu, X.; Zhu, D. Profiling and relationship of water-soluble sugar and protein compositions in soybean seeds. *Food Chem.* 2016, 196, 776–782. [CrossRef]
5. Sarkar, A.; Mouluk, U. Gene microarray data analysis using parallel point-symmetry-based clustering. *Int. J. Data Min. Bioinform.* 2015, 11, 277–300. [CrossRef]
6. Abid, S.H.; Quaez, U.J.; Contreras-Reyes, J.E. An information-theoretic approach for multivariate skew-t distributions and applications. *Mathematics* 2021, 9, 146. [CrossRef]
7. Zadeh, L.A. Fuzzy sets. *Inf. Control.* 1965, 8, 338–353. [CrossRef]
8. Niittymäki, J.; Turunen, E. Traffic signal control on similarity logic reasoning. *Fuzzy Sets Syst.* 2003, 133, 109–131. [CrossRef]
9. Bezdek, J.C. *Pattern Recognition with Fuzzy Objective Function Algorithms*; Kluwer Academic Publishers: Norwell, MA, USA, 1981.
10. Deb, S.K.; Bhattacharyya, B. Fuzzy decision support system for manufacturing facilities layout planning. *Decis. Support Syst.* 2005, 40, 305–314. [CrossRef]
11. Aloise, D.; Deshpande, A.; Hansen, P.; Popat, P. NP-hardness of Euclidean sum-of-squares clustering. *Mach. Learn.* 2009, 75, 245–248. [CrossRef]
12. Raykov, Y.P.; Boukouvalas, A.; Baig, F.; Little, M.A. What to do when K-means clustering fails: A simple yet principled alternative algorithm. *PloS ONE* 2016, 11, e0162259. [CrossRef]
13. Jung, Y.G.; Kang, M.S.; Heo, J. Clustering performance comparison using K-means and expectation maximization algorithms. *Biotechnol. Biotechnol. Equip.* 2014, 28, 544–548. [CrossRef]
14. Jaroš, M.; Strakoš, P.; Karásek, T.; Říha, L.; Vašatová, A.; Jarošová, M.; Kozubek, T. Implementation of K-means segmentation algorithm on Intel Xeon Phi and GPU: Application in medical imaging. *Adv. Eng. Softw.* 2017, 103, 21–28. [CrossRef]
15. Havens, T.C.; Bezdek, J.C.; Leckie, C.; Hall, L.O.; Palaniswami, M. Fuzzy c-means algorithms for very large data. *IEEE Trans. Fuzzy Syst.* **2012**, *20*, 1130–1146. [CrossRef]

16. Ludwig, S.A. MapReduce-based fuzzy c-means clustering algorithm: Implementation and scalability. *Int. J. Mach. Learn. Cybern.* **2015**, *6*, 923–934. [CrossRef]

17. Flores-Sintas, A.; Cadenas, J.; Martin, F. A local geometrical properties application to fuzzy clustering. *Fuzzy Sets Syst.* **1998**, *100*, 245–256. [CrossRef]

18. Ntinas, V.G.; Moutafis, B.E.; Trunfio, G.A.; Sirakoulis, G.C. Parallel fuzzy cellular automata for data-driven simulation of wildfire spreading. *J. Comput. Sci.* **2017**, *21*, 469–485. [CrossRef]

19. Zhuge, Y.; Ciesielski, K.C.; Udupa, J.K.; Miller, R.W. GPU-based relative fuzzy connectedness image segmentation. *Med. Phys.* **2013**, *40*, 011903. [CrossRef]

20. Hurtik, P.; Števuliáková, P. Pattern matching: Overview, benchmark and comparison with F-transform general matching algorithm. *Soft Comput.* **2017**, *21*, 3525–3536. [CrossRef]

21. Kassani, P.H.; Teoh, A.B.J. and Kim, E. Evolutionary-modified fuzzy nearest-neighbor rule for pattern classification. *Expert Syst. Appl.* **2017**, *88*, 258–269. [CrossRef]

22. Téllez-Velázquez, A.; Cruz-Barbosa, R. A CUDA-streams inference machine for non-singleton fuzzy systems. *Concurr. Comput.* **2018**, *30*, e4382. [CrossRef]

23. Zhang, C.L.; Xu, Y.P.; Xu, Z.J.; He, J.; Wang, J.; Adu, J.H. A fuzzy neural network based dynamic data allocation model on heterogeneous multi-GPUs for large-scale computations. *Int. J. Autom. Comput.* **2018**, *15*, 181–193. [CrossRef]

24. Soto, J.; Flores-Sintas, A.; Palarea-Albaladejo, J. Improving probabilities in a fuzzy clustering partition. *Fuzzy Sets Syst.* **2008**, *159*, 406–421. [CrossRef]

25. Timón, I.; Soto, J.; Pérez-Sánchez, H.; Cecilia, J.M. Parallel implementation of fuzzy minimals clustering algorithm. *Expert Syst. Appl.* **2018**, *48*, 35–41. [CrossRef]

26. United States Board on Geographic Names. Available online: [https://geonames.usgs.gov/domestic/index.html](https://geonames.usgs.gov/domestic/index.html) (accessed on 19 January 2017).