IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 31, 2020, accepted February 24, 2020, date of publication February 27, 2020, date of current version March 10, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2976900

Parallel Implementation of K-Means Algorithm on FPGA

LEONARDO A. DIAS[1], JOÃO C. FERREIRA[2,3], (Senior Member, IEEE), AND MARCELO A. C. FERNANDES[1,4,5]

[1]Laboratory of Machine Learning and Intelligent Instrumentation (LMLII), nPITI-IMD, Federal University of Rio Grande do Norte, Natal 59078-970, Brazil
[2]INESC TEC, University of Porto, 4200-465 Porto, Portugal
[3]Faculty of Engineering, University of Porto, 4200-465 Porto, Portugal
[4]Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte, Natal 59078-970, Brazil
[5]John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138, USA

Corresponding author: Marcelo A. C. Fernandes (mfernandes@dca.urn.br)

This work was supported in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) under Grant 001. This work was partially financed by the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, through national funds, and co-funded by the FEDER, where applicable.

ABSTRACT The K-means algorithm is widely used to find correlations between data in different application domains. However, given the massive amount of data stored, known as Big Data, the need for high-speed processing to analyze data has become even more critical, especially for real-time applications. A solution that has been adopted to increase the processing speed is the use of parallel implementations on FPGA, which has proved to be more efficient than sequential systems. Hence, this paper proposes a fully parallel implementation of the K-means algorithm on FPGA to optimize the system's processing time, thus enabling real-time applications. This proposal, unlike most implementations proposed in the literature, even parallel ones, do not have sequential steps, a limiting factor of processing speed. Results related to processing time (or throughput) and FPGA area occupancy (or hardware resources) were analyzed for different parameters, reaching performances higher than 53 millions of data points processed per second. Comparisons to the state of the art are also presented, showing speedups of more than 15573× over a partially serial implementation.

INDEX TERMS Parallel implementation, FPGA, K-means algorithm, reconfigurable computing.

I. INTRODUCTION

In recent years, technological advances of digital devices resulted in a significant increase in the amount of digital data processed and stored, which in turn are generated in a variety of fields, including health, traffic, climatology, mobile devices, and social networks [1], [2]. Analyzing this massive amount of data and extracting relevant information has become an essential process in decision making for various organizations in several areas such as finance, banking, healthcare, and communication [3]. Therefore, organizations have been facing a challenging scenario when processing such massive amounts of data due to increased demand for results in shorter time frames. Consequently, the development of computational solutions for systems operating in real-time has become a difficult task [4].

A solution that has been widely adopted to meet the demand for high-speed processing (or high-throughput) is to devise parallel implementations of the relevant algorithms. Parallel execution allows different sections to operate with different sets of data concurrently [5], [6]. In addition, the reconfigurable computing implementations using field-programmable gate arrays (FPGAs) combined with parallelization techniques proposed in the literature have shown satisfactory results when compared to systems based on sequential solutions [5], [7]. FPGAs are widely used to implement these algorithms in parallel, as processing time and cost are significantly reduced [8]. The FPGA is an array of reconfigurable logic blocks that allows the implementation of several logic circuits that can operate independently, enabling parallel processing of different data simultaneously [9]. Therefore, it answers the demand for reduced processing time by allowing the parallel implementation of algorithms used to analyze massive datasets.

This paper presents and evaluates a fully parallel implementation of the K-means algorithm on an FPGA with focus on high performance to extract data patterns from massive datasets in a short time. The K-means algorithm is widely used in the process of data clustering because it allows finding patterns and correlations in data by similarity,

The associate editor coordinating the review of this manuscript and approving it for publication was Akansha Singh.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

in an unsupervised way; in addition it is a relatively simple technique [10], [11].

The most commonly found implementations of the K-means algorithm in the literature, including parallel ones, generally have sequential sections, thus limiting the processing speed compared to fully parallel implementations. Therefore, this paper's main contributions are:

- A complete, fully parallel hardware implementation without additional embedded processors or software;
- A detailed description of the modules implemented in hardware enabling the replication of the work;
- A thorough speed and area occupation analysis based on the post-synthesis results for reconfigurable hardware;
- FPGA synthesis and analysis of the architecture for three different distance metrics and fixed-point sizes for assisting future implementations in the selection of the best metrics for a specific application;
- FPGA synthesis and analysis of the implementation of the square root function for the Euclidean distance metric.

The results show that the implementation proposed here has general applicability to situations where large data amounts must be processed under strict time restrictions, therefore enabling its adoption in real-time applications.

The next subsection presents an overview of the most relevant related works.

A. RELATED WORKS

In the last years, applications of K-means algorithm involving parallel and distributed, hardware only and hybrid (software and hardware) implementations have been reported in the literature.

In [5], a parameterized K-means algorithm was fully implemented on FPGA, in a General Purpose Processor (GPP) and also in a Graphics Processing Unit (GPU), in order to compare the speedup between each implementation. In the case of FPGAs, the data points and the centroids are stored in memories, which can be internal or external. Focusing on a reduction of the area the input data (present in the dataset) and the centroids updating are implemented serially, and the distance metric of the data point and all clusters are obtained simultaneously, on a parallel scheme. A speedup of about 6.7× was achieved by the FPGA implementation in comparison to a GPU-based one; the speedup over a GPP-based implementation was 54×. However, in addition to sequential steps, the constant accesses to memory for writing and reading data points and centroids reduce the processing time performance of the implementation.

In [7], the K-means algorithm has been implemented in different FPGAs using MapReduce model to compare the performance regarding speedup. The similarity distance metric adopted is based on Euclidean distance and the cluser process is performed by the map function. Therefore a key-value pair list containing the data point and its nearest cluster centroid is generated. This list is then sorted based on centroid values by a shuffling function, allowing to group the data according to their clusters. Afterwards the reduce function is performed in a parallel scheme, updating all centroids simultaneously. This proposal was implemented in two FPGAs dedicated to the map function (called mappers) and one dedicated to reducing function (called reducer). The FPGAs are Xilinx Kintex-7 XC7k325T devices, and the tests presented were performed for 32 mappers and 12 reducer functions. The proposed design has a speedup of about 20.6× compared to the same implementation in software, despite the bottleneck caused by the communication between the FPGAs. The resource usage for both mappers is 36% of the registers, 69% of the look-up-tables (LUTs), 49% of the DSP and 33% of block RAM memory, and for the reducer 44% of the registers, 71% of LUTs, 34% of the DSP and 24% of block RAM memory. However, the communication between different FPGAs and the access to memory blocks limits the processing time. In addition, the clustering step based on the map function includes sequential processes.

A hybrid implementation is proposed in [12] to compare the speedup regarding an ARM processor. A hybrid implementation executes only part of the algorithm steps on hardware and the rest on software. This hybrid proposal aims to reduce FPGA area overhead and improve processing speed. The FPGA contains the circuits to calculate the similarity distance metric, which is based on the Manhattan distance, and the circuits to group data points with the nearest centroid. The software is responsible for updating the centroid values, to avoid creating division circuits, needed for this step, on the FPGA, thus reducing the area overhead. A 32-bit floating-point representation for data points is also used to provide a high resolution. The resulting area overhead for the FPGA is 11560 LUTs, 11171 registers, 10 RAM blocks (BRAM) and 28 DSPs. A speedup of about 10× over the software version running on the ARM processor is achieved. However, the area overhead is relatively high considering that only part of the K-means algorithm is running on the FPGA. In addition, a bottleneck caused by the communication between FPGA and the ARM processor limits the processing time.

In the work presented in [4], the K-means algorithm is completely developed on FPGA and also GPP. External memories are used to store the data set and initial centroid values, and internal block memories are used to store each data point and centroid for processing. This implementation allows several data points to be processed in parallel, but the similarity distance metric to each centroid is obtained sequentially. Therefore, the processing speed is reduced when compared to a fully parallel implementation. The implementation uses 33% of the Virtex-6 xc6vlx240t total resources for a speedup of about 368× over the GPP version. In addition to the sequential steps, there is a communication bottleneck caused by frequent memory accesses.

In the work detailed in [13], like in [7], the K-means algorithm is implemented with the aim of accelerating Hadoop clusters. A hybrid architecture is also proposed.

The similarity metric is implemented on FPGA using the Euclidean distance and centroid updating is implemented in software. The hybrid implementation was chosen to reduce FPGA area overhead. A 4× speedup has been obtained compared to using Apache Mahout Machine Learning Libraries, a distributed linear algebra framework written in Scala [14].

Similar to [7], the work presented in [15] also developed a implementation based on MapReduce model. The K-means algorithm is completely developed in a Zynq xc7z045ffg600-2 FPGA and allows data points to be processed in parallel. The dataset is split according to the number of mapper circuits, were each mapper is responsible to process a dataset slice. The mappers are responsible to obtain the distance measurement and assign data to nearest cluster. In addition, one reduce circuit is used to update the all the centroids. The implementation achieved a throughput of 28.74Gbps and occupied 47.61% and 81.51% of of registers and LUTs, respectively.

Therefore, based on those papers, it is clear that the use of FPGAs to accelerate the processing time of massive amounts of data is feasible and effective when compared to general-purpose implementations or sequential systems. However, as mentioned, most implementations have sequential steps and frequent accesses to memories, which can limit their use in real-time applications. Hence, this paper proposes a parallel implementation of each k-means process, making real-time applications possible and reliable.

B. PAPER ORGANIZATION

The remainder of this paper is organized as follows: Section II explains the K-means algorithm and its operation. Section III shows a detailed description of the architecture proposed in this paper, while section IV presents and analyses the results obtained from the described implementation, including a comparison to other works. Finally, Section V presents some concluding remarks.

II. THE K-MEANS ALGORITHM

K-means algorithm allows datasets to be partitioned and grouped based on similarity metrics. Each group is called a cluster and created based on similarity metrics. Therefore, the algorithm aim is to generate K clusters, by assigning data points of a dataset to the closest representative data, which in turn is called centroid [11]. Thereby, K-means is often used for recognition and to find patterns in massive datasets [16].

The cluster number, K, is an integer, and for each k-th cluster, a centroid, c_k, is assigned. The initial value of each k-th centroid, is randomly generated by choosing random data points in the dataset, which is the most used way as can be seen in the following proposed papers: [17]–[20]. It can also be generated by other algorithms [16].

The set of centroids, $\mathbf{c}[m]$, is defined as

$$\mathbf{c}[m](n) = [c_1[m](n), c_2[m](n), \ldots, c_K[m](n)] \tag{1}$$

where m represent the number of bits that describes each centroid and n represents the n-th iteration.

Algorithm 1 K-Means Pseudocode

1: Initialise $\mathbf{c}[m]$ centroids randomly;
2: **while** $\mathbf{c}[m](n+1) \neq \mathbf{c}[m](n)$ **do**
3:   **for** $j \leftarrow 1$ *to* J **do**
4:     **for** $k \leftarrow 1$ *to* K **do**
5:       Compute the distance $d_k(p_j[m], c_k[m])(n)$ according to equation (3) or equation (4);
6:       $\mathbf{d}(n) \leftarrow d_k(p_j[m], c_k[m])(n)$;
7:     **end for**
8:     **for** $k \leftarrow 1$ *to* K **do**
9:       **if** $\mathbf{d}(k-1) \leq \mathbf{d}(k)$ **then**
10:         $c_k \leftarrow p_j[m]$;
11:       **end if**
12:     **end for**
13:     Update c_k according to equation (6);
14:   **end for**
15:   $n \leftarrow n+1$;
16: **end while**

In every n-th iteration of the algorithm, the similarity between a cluster centroid and a data point, $p_j[m]$, of a dataset, $\mathbf{x}[m]$, is obtained. A dataset, $\mathbf{x}[m]$, of J data points, can be represented as

$$\mathbf{x}[m] = [p_1[m](n), p_2[m](n), \ldots, p_J[m](n)]. \tag{2}$$

That similarity of a k-th cluster, which is represented by its centroid, $c_k[m]$, and a j-th data point, $p_j[m]$, is defined based on the distance between them. Therefore, this distance metric determines to which centroid the data point is assigned. Afterward, the centroid will be updated with the mean value of all data points assigned to it. The process is then repeated, but the distance is now obtained in regard to the new centroid value (after updated) until their values do not change or a predefined number of iterations has been performed.

The Algorithm 1 presents the K-means pseudocode. This code details all the variables and procedures that will be used in the implementation to be presented in the following sections. It starts by randomly generating the first set of centroids, $\mathbf{c}[m]$, as shown in equation 1. Therefore, one centroid of m bits for each k-th cluster, as shown in line 1.

As can be seen from line 3, at each n-th iteration, the distance of every j-th data point, $p_j[m]$, in relation to each k-th centroid, $c_k[m]$, is calculated. This distance for each k-th centroid is often obtained, according to [11], by Minkowski equation, which is defined as follows

$$d_k(p_j, c_k) = (\sum_{i=1}^{D} |p_{j,i}[m] - c_{k,i}[m]|^r)^{1/r} \tag{3}$$

where D represent the number of data dimension/attributes, and r defines which distance metric is used. For $r = 1$ manhattan distance is obtained and for $r = 2$ the euclidian distance.

It is also common to adopt the squared euclidean distance to avoid the complexity of a square root function required by

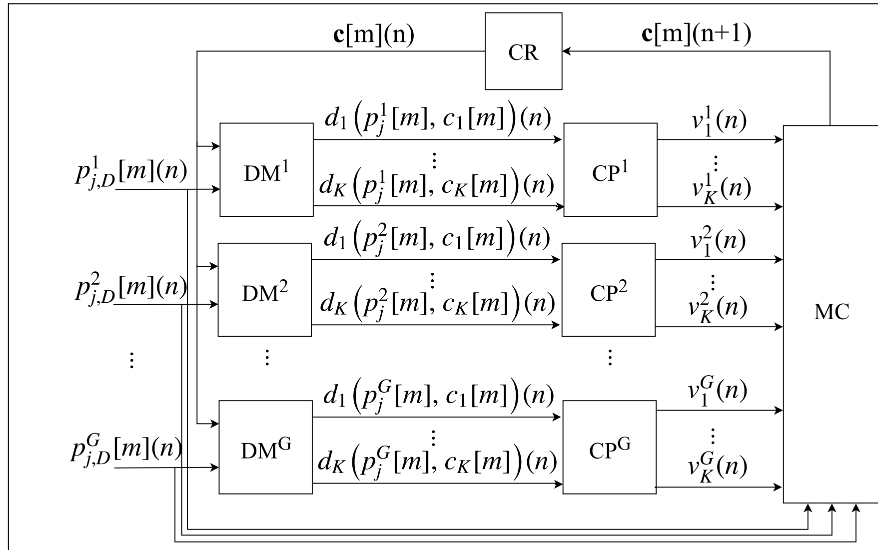

FIGURE 1. General architecture of the proposed parallel K-means algorithm implementation.

euclidean distance ($r = 2$) and maintain an accurate results compared to manhattan distance. The squared euclidean distance is derived from euclidean distance, and defined as follow

$$d_k(p_j, c_k)^2 = \sum_{i=1}^{D} |p_{j,i}[m] - c_{k,i}[m]|^2. \quad (4)$$

At each n-th iteration, a vector of distances, $\mathbf{d}(n)$, stores the distance of a j-th data point, $p_j[m]$, in relation to each centroid, $c_k[m]$, and it is represented as

$$\mathbf{d}(n) = [d_1(p_j[m], c_1[m])(n), \ldots, d_K(p_j[m], c_K[m])(n)] \quad (5)$$

where d_k is the k-th distance obtained according to equation (3) or equation (4).

After calculating the distance, the data point, $p_j[m]$, is assigned to the k-th cluster with the closest centroid, $c_k[m]$, in other words, the data point is assigned to the cluster centroid with the minimum distance present in the vector of distances, $\mathbf{d}(n)$, as can be seen in lines 8 to 12 in the Algorithm 1. According to [11] and [21], the most used distance metric is euclidean distance, shown in equation (3) for $r = 2$, because it provide more accuracy compared to manhattan distance [22].

Lastly, each centroid, $c_k[m]$, present in the set of centroids, $\mathbf{c}[m]$, is updated with the mean value of all data points assigned to it, according to the following equation

$$c_k[m] = \frac{1}{Z} \sum_{w=1}^{Z} p_{j,w}[m] \quad (6)$$

where Z represent the total amount of data points in that cluster, that is, assigned to this centroid. The process is then repeated in the next n-th iteration if the new centroid values, $\mathbf{c}[m](n+1)$ are different from the actual values, $\mathbf{c}[m](n)$, as can be seen in line 2.

It is noticeable in Algorithm 1 that the larger the dataset and the centroid number, the greater is the number of iterations. Thus, when applied to massive datasets, the number of iterations is very high, resulting in a high computational complexity, mainly due to the calculation of the distance metric. Hence, it is clear the need for high-speed processing.

III. IMPLEMENTATION DESCRIPTION

The entire K-means algorithm, presented in the Algorithm 1, was developed using a parallel architecture focusing on accelerating the data processing speed regardless of the dataset, taking advantage of the available FPGA hardware resources, similarly to [23]. It is shown in the Figure 1, the general architecture of this proposal. The figure details in block diagram the main modules of the proposed implementation, which in turn were encapsulated in order to make the general visualization of the architecture less complex.

The algorithm flowpath of this implementation is organized in four different main modules, as shown in Figure 1, where each of them represents a K-means step. Firstly, the Centroid Register (CR) module stores every cluster centroid, $c_k[m]$, of the set, $\mathbf{c}[m]$. The Distance Metric (DM) module is responsible to define the similarity by calculating the distance between each j-th data point, $p_j[m]$, to each k-th centroid, $c_k[m]$, while the Clustering Process (CP) module defines to which centroid the data point is assigned. Lastly, the Mean Centroid (MC) module update the value of each k-th centroid in the set, $\mathbf{c}[m]$.

To be fully implemented in parallel, the architecture proposed here is replicated according to a parallelization degree, called here as g. This parameter allows a total of G data points, $p_j^g[m]$, to be entered simultaneously, wherein $g = 1, 2, \ldots, G$, as can be observed in the Figure 1. In order to process these g data points, the DM and CP modules are replicated G times. Therefore, the algorithm flowpath

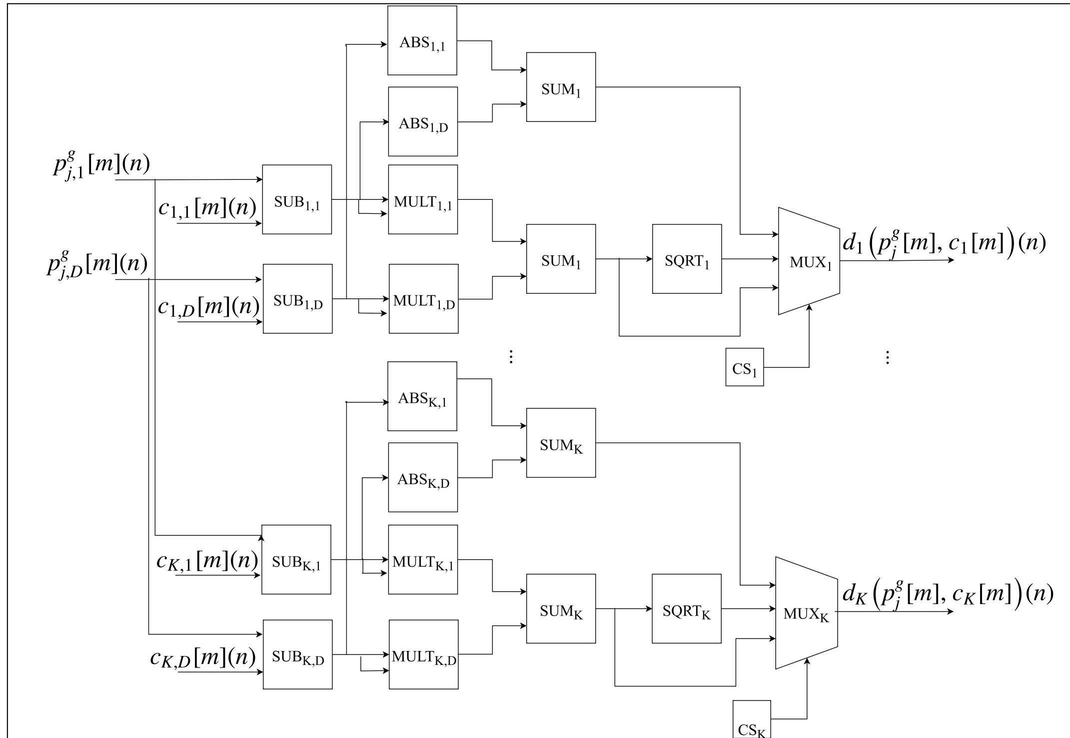

FIGURE 2. A g-th distance metric (DM) submodule for a j-th data point, $p^g_j[m]$, and K centroids, $c_k[m]$.

is executed for G different data points simultaneously. It is important to emphasize that this implementation can be replicated for several data points to be processed in parallel. In addition, note that the implementation is scalable, in other words, every circuit and parameter of the implementation can be replicated, limited only by the resources of the used FPGA. Hence, this proposal can be used to process any Big Data, that is, any data dimensions/attributes, cluster centroids, etc. required for a dataset.

The initial centroid values present in the set, $\mathbf{c}[m]$, randomly chosen, according to the Algorithm 1, are generated outside the FPGA and stored in CR module through Ethernet Gbit, and then updated by the mean value of the data points nearby. This update process occurs at every n-th iteration. When the centroids values do not change, the algorithm stops running to indicate that all data points are assigned to their respective cluster. The runtime of the algorithm can also be stopped by a predetermined number of iterations.

Those modules shown in Figure 1 are made up of submodules, that has its specific implementations, also in parallel, that will be detailed in the next subsections.

A. DISTANCE METRIC MODULE

A Distance Metric (DM) module, has the purpose of calculating the distance of a j-th D-dimensional data point, $p^g_{j,D}[m]$, to each k-th centroid, $c_{k,D}[m]$, present in the set, $\mathbf{c}[m]$, at each n-th iteration, to indicate their similarity. This is the first K-means step realized after initializing the centroids.

It is shown in Figure 2, how each g-th DM module is built. In order to calculate that distance metric, according to equations (3) and (4), mentioned in section II, this module is composed by the following submodules: subtractors ($SUB_{k,D}$), multipliers ($MULT_{k,D}$), absolute ($ABS_{k,D}$), adders ($SUM_{k,D}$), square root functions ($SQRT_k$) and multiplexers (MUX_k). As the purpose of this proposal is a completely parallel implementation, in addition to replicating this module G times, to obtain the similarity for G data points simultaneously, its submodules are also replicated to obtain the distance of a j-th data point, $p^g_j[m](n)$, to each k-th centroid, $c_k[m](n)$, in parallel, that is, in only one iteration. Hence, these submodules are replicated according to the number of centroids and dimensions, as can be seen in Figure 2.

Firstly, for each data dimension/attribute, D, the submodule $SUB_{k,D}$ subtract a centroid value, $c_{k,D}[m]$, from a data point value, $p^g_{j,D}[m]$. The result generated is multiplied by itself in the subsequent submodule $MULT_{k,D}$, and it is also obtained the absolute value in $ABS_{k,D}$ submodule. Each $MULT_{k,D}$ and $ABS_{k,D}$ value is then summed by the submodules SUM_k. Lastly, the submodule MUX_k is used to define which equation should be adopted. As can be observed in the Figure 2, according to the position of the mux data selector (CS_k), the manhattan distance, defined in the equation (3) for $r = 1$, the euclidean distance, defined in the equation (3) for $r = 2$, or the squared euclidean distance, defined in the equation (4), is performed. The submodules $ABS_{k,D}$ and SUM_k are used to perform manhattan distance, while euclidean distance is performed by $SQRT_k$ submodule path,

and the submodules $MULT_k$ and SUM_k are used for squared euclidean distance.

The resultant vector of distances, $\mathbf{d}(n)$, shown in equation (5), is then obtained for each k-th cluster, in parallel, at each n-th iteration. Hence, k distances, $d_K(p_j^g[m], c_K[m])$, are genetared simultaneously for each data point, $p_j^g[m]$.

In order to estimate the scalability, the total amount of each submodule, necessary to perform this step, shown in Figure 2, can be defined by the cluster and dimension number. Thus, the amount of subtractors, multipliers and absolute is defined as

$$total_{SUB,MULT,ABS} = K * D \tag{7}$$

while the total amount of adders is

$$total_{SUM} = (2 * (D - 1)) * K \tag{8}$$

The number of multiplexers and square root function submodule is defined according to the centroid number, so a total of K $SQRT_k$ is created. Note that SUM_k is created only if there is more than one dimension/attribute, in other words, this submodule is not necessary for $D = 1$.

This module and its submodules are implemented in fixed-point to reduce the number of bits (m) compared to floating-point implementations. The adders, SUM_k, and absolute, ABS_k, increases only 1 bit in the total number of bits, so its output size has been set to full. Meanwhile, the multipliers submodules, $MULT_{k,D}$, can double the number of bits, thus the size of its output was limited to increase the number of bits only by 2, as the data points used are normalized between 0 and 1. In case Euclidean distance is chosen, the data is converted to floating-point to execute the operation in $SQRT_k$ submodule in only one iteration and then converted again to fixed-point. Thereby, this submodule does not increase the number of bits. Hence, as can be observed in the Figure 2, the bit width, m, increases only 2 bits for manhattan distance and 3 bits for euclidean and squared euclidean.

Each k-th distance, d_k, is then passed to next process, the Clustering Process, to determine which cluster the data point should be assigned. This DM module was developed for those three different distance metrics in order to analyze the the tradeoff between them, and also the complexity of $SQRT_k$ submodule concerning to consumption of area and processing speed.

B. CLUSTERING PROCESS MODULE

A Clustering Process (CP) module, has the purpose of assigning each j-th data point, $p_j^g[m](n)$, to the closest k-th cluster centroid, $c_k[m]$, at each n-th iteration, based on the distance vector, $\mathbf{d}(n)$, shown in equation (5), generated in DM module. As the cluster number is predefined, the condition to realize this step is $K \geq 2$, otherwise, the entire dataset will be in the same cluster, and this module is not required.

This assignment task is realized by comparing the distance values, $d_k(p_j^g[m], c_k[m])$, received from the previous submodule. In order to realize that, this module is composed of the

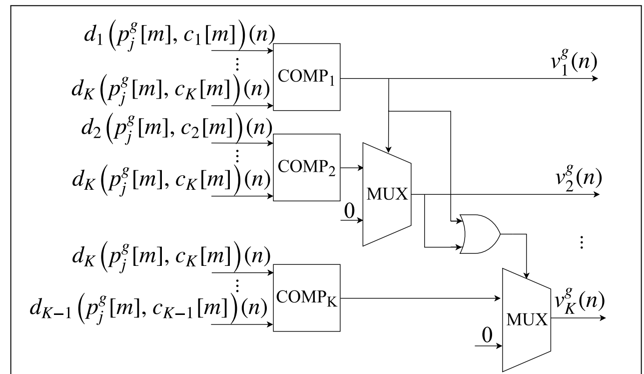

FIGURE 3. A g-th clustering process (CP) submodules for K centroids.

following submodules: comparators ($COMP_k$), logical OR gates and multiplexers (MUX). Since this implementation is completely parallel, CP module is replicated G times to assign G data points, $p_j^g[m](n)$, to their respective cluster simultaneously, as can be seen in Figure 1. In addition, its submodules are also replicated in order to compare every k-th distance in the vector of distances ($\mathbf{d}(n)$), in parallel, obtaining the lowest in only one iteration.

As can be observed in the Figure 3, at each n-th iteration, each submodule $COMP_k$ receives as input all K distances values (d_k), related to each k-th centroid, $c_k[m]$. Then it checks if its k-th distance, that is, the distance present in the first input, is lower than the others by comparing them.

Each k-th $COMP_k$ has an output value represented by v_k^g, which in turn is a boolean value, and defined as

$$v_k^g(n) = \begin{cases} 1, & \text{if } d_k \leq d_j, \quad \forall j, \ 1 \leq j \leq K \ \text{ where } j \neq k. \\ 0, & \text{otherwise.} \end{cases}$$

where v_k^g indicates that $p_j^g[m](n)$ is close to its respective k-th centroid when it assumes the bit 1 value. When $p_j^g[m](n)$ is not close to the k-th centroid distance of $COMP_k$, v_k^g assumes a bit 0 value.

Thereby, suppose a dataset that needs to be grouped into k clusters, were $c_1[m]$ to $c_k[m]$ are their respectively centroids. Considering the distance of a j-th data point, $p_j^g[m]$, regarding to first centroid being the shortest distance, that is, $d_1(p_j^g[m], c_1[m]) < d_k(p_j^g[m], c_k[m]), \forall k, 1 \leq k \leq K$ and $k \neq 1$, the output of $COMP_1$, $v_1^g[m]$, is set to bit 1. Hence, the remaing comparators outputs, $v_k^g[m]$, is set to bit 0 through the MUXs and OR gates.

In case a data point is equally distant between two centroids, the comparator output that has the lowest k-th index will be assigned to bit 1, and a bit 0 will be assigned to the remaining.

Therefore, at each n-th iteration, each g-th CP module generates k boolean values, $v_k^g[m]$, as shown in Figure 1, in which just one of them is set to 1, while the remaining is equal to 0. This helps the next step, Mean Centroid, recognize which cluster the data point belongs.

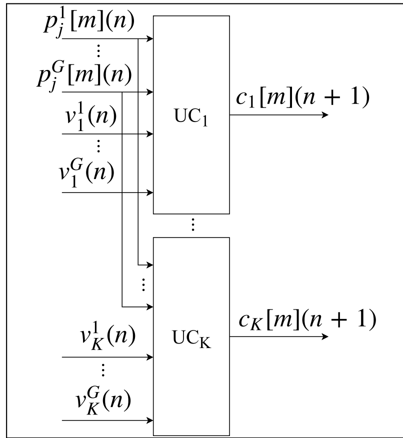

FIGURE 4. Mean Centroid (MC) submodules for K centroids.

The total number of each submodule, necessary to define which centroid the data point is closer, as shown in the Figure 3, can be defined based on the cluster number. Therefore, the amount of comparators, $COMP_k$, is equal to k, while the amount of MUX, is defined as

$$total_{MUX} = K - 1 \tag{9}$$

and the number of logical OR gates, in its turn, is defined by

$$total_{gates} = K - 2 \tag{10}$$

This module was developed using only logical submodules, thus it operates with boolean values, which requires only one bit, and also do not increase the total number of bits as the previous module.

C. MEAN CENTROID MODULE

The Mean Centroid (MC) module, is responsible to update every centroid present in the set of centroids, $\mathbf{c}[m]$, by calculating the mean value of all data points, $p_j^g[m]$, assigned to a determined centroid, $c_k[m]$, at each n-th iteration, as can be seen in line 13 of Algorithm 1.

As can be observed in Figure 1, the MC module, different from others, is not replicated. It receives as input, all G data points, $p_j^g[m]$, and also each k-th output of every g-th CP module, $v_k^g(n)$. However, its submodules are replicated in order to update each centroid, $c_k[m]$, in parallel, as shown in Figure 4. As can be seen, for each k-th centroid of the set $\mathbf{c}[m]$, there is a update centroid submodule, UC_k. Every k-th submodule is responsible to update its k-th centroid, $c_k[m]$, were each UC_k receives a total of G data points, $p_j^g[m]$, and also G CP outputs, $v_k^g(n)$, regarding its respective k-th centroid.

As mentioned in section II, the mean value is obtained according to equation (6), so the update centroid submodule, UC_k, consists of following circuits: adders (called here as $SUM - N^k$ and $SUM - D^k$), accumulators (called here as $ACC - N^k$ and $ACC - D^k$), a divisor (DIV^k), a register (REG^k), a comparator ($COMP^k$) and multiplexers (MUX), as shown in Figure 5. The suffix $-N$, in the adder and accumulator, indicate that these circuits are used to generate the divisor numerator, and the suffix $-D$ is used to indicate these circuits are used to generate the divisor denominator. Hence, the divisor numerator is generated by $SUM - N^k$ and $ACC - N^k$ circuits, respectively, and the divisor denominator is generated by $SUM - D^k$ and $ACC - D^k$. These circuits are also replicated for each data dimension/attribute, D.

At each n-th iteration, the values of $v_k^g(n)$ are summed in $SUM - D^k$ submodule and accumulated in $ACC - D^k$, generating the denominator. These values are also used as data selector of the first $MUXs$, controlling the data points being summed in $SUM - N^k$ and accumulated in $ACC - N^k$ to form the numerator, as can be observed in Figure 5.

As mentioned in the previous subsections, when a data point, $p_j^g[m]$, is assigned to a centroid, $c_k[m]$, the input $v_k^g(n)$ assumes the bit 1 value. Therefore, if $v_k^g(n) = 1$, the g-th data point at that n-th iteration, is summed in $SUM - N^k$ of this k-th centroid. In addition to that, $SUM - D^k$ sums the G values of that k-th $v_k^g(n)$, to determine the data point number present in the cluster. The result of the adders is then accumulated by $ACC - N^k$ and $ACC - D^k$, respectively. Note that for $v_k^g(n) = 0$, that is, when there are no data points close to that centroid, the value in both accumulators does not change as both adders receive only zeros.

After each j-th data point present in the dataset has been entered in the algorithm, a division operation is performed by the circuit DIV^k, based on the values of both accumulators, generating the new centroid value, $c_k[m](n+1)$. This new centroid value is then stored in the register REG^k, and also sent to Centroid Register module, through the last MUX.

If there aren't any data points assigned to the centroid, that is, when $SUM - D^k = 0$, the new value, $c_k[m](n+1)$, is defined by the REG^k previously stored. This is accomplished through the comparator, $COMP^k$, used as a data selector for the last MUX, which checks if $SUM - D^k \neq 0$ to propagate the division result, otherwise, it propagates REG^k, which in turn has the initial centroid, $c_k[m]$, randomly generated, as its initial value.

The numerator and denominator values are internally converted to floating-point in order to realize division operation in just one iteration, and also to not increase the number of bits.

The total amount of each circuit, necessary to perform this step, shown in Figure 5, can be defined by the cluster number, K, and dimensions, D. The number of $SUM - N^k$, $SUM - D^k$, $ACC - N^k$, $ACC - D^k$, DIV^k and REG^k is defined as

$$total_{circuits} = K * D \tag{11}$$

while the amount of MUX necessary can be defined as

$$total_{MUX} = G + D \tag{12}$$

and the amount of comparators is equal to cluster number.

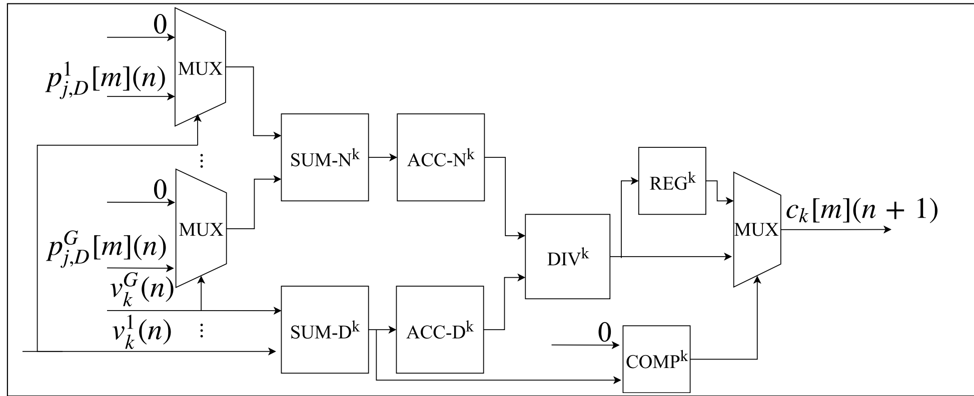

FIGURE 5. Circuits that constitute the k-th update centroid (UC^k) submodule.

D. CENTROID REGISTER MODULE

The Centroid Register (CR) module, is used to store each k-th centroid, $c_k[m](n)$, present in the set of centroids, $\mathbf{c}[m]$, and, as shown in Figure 1, it is not replicated. This module is constituted by a set of m-bit registers, in which for each dimension/attribute (D), of every k-th centroid, there is a register. Hence, the total amount of registers required is defined as

$$total_{REG} = D * K. \tag{13}$$

The initial value of each k-the centroid, $c_k[m](n)$, plays a fundamental role in the resultant clusters generated by the K-means algorithm. In the tests performed with the proposed implementation, they were randomly generated as shown in the line 1 of Algorithm 1. As mentioned before, these values are generated outside the FPGA and stored in their respective register, therefore the initial value can also be generated by a second algorithm to improve the results, as the K-means++.

The MC module is responsible to update this module with the new centroid values, $c_k[m](n+1)$, at each n-th iteration. After updated, the algorithm steps are repeated until each k-th centroid of $\mathbf{c}[m]$ do not change, as shown in line 2 of Algorithm 1. This indicates that each j-th data point, $p_j^g[m]$, present in a dataset is assigned to the correct cluster. The algorithm can also stop after complete a predefined number of iterations.

Thus, this implementation allows the insertion of G different data points, $p_j^g[m]$, and also assign them to K centroids, in parallel, at each n-th iteration. The centroid values are updated after a total of $\frac{J}{G}$ iterations, to indicate that all data points are grouped in their respective cluster.

IV. EXPERIMENTAL RESULTS

The development of this project was accomplished using the development platform provided by the FPGA manufacturer, in this case, Xilinx [24]. This platform allows the user to develop systems using the block diagram strategy instead of VHDL or Verilog. This approach allows the developer to maintain a greater focus on the system architecture in a simpler way and without giving up control of low-level

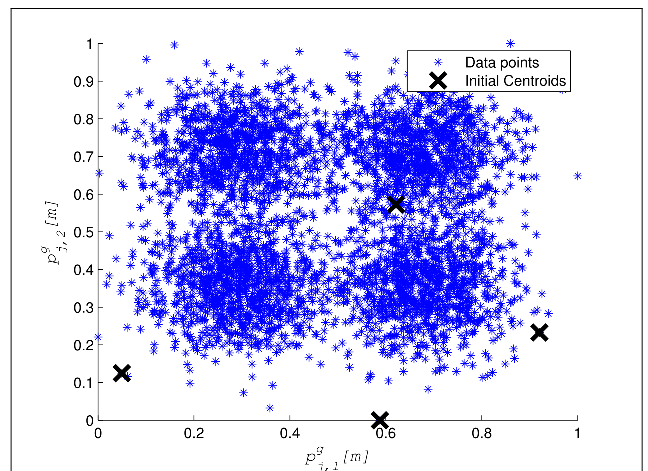

FIGURE 6. Synthetic two-dimensional gaussian dataset and its initial centroids, randomly generated.

configurations [25]. All results were obtained for an FPGA Virtex-6 xc6vlx240t-1ff1156. This FPGA has a total of 37680 slices containing 301440 registers (flip-flops), 150720 LUTs that can be used as memory or any logic and 768 DSP cells.

Initially, in order to validate the K-means algorithm implemented in this paper, simulations were performed for a synthetic Gaussian dataset, compounded of two-dimensional data points, $p_{j,2}^g[m], \forall j, 1 \leq j \leq 4096$. The parallelization degree was set to $g = 4$, and also the number of cluster centroids, that is $K = 4$. As mentioned in the section III-A, each j-th data point and k-th centroid is represented in fixed-point, then the bit width was defined as $m = 14$ bits, wherein 12 bits are dedicated to the fractional part. It is shown in the Figure 6 the gaussian dataset and the initial centroid values, randomly generated, while in the Figure 7 is shown the resultant clusters and final centroids. As can be observed, even with overlapping data, the implementation was able the recognize and cluster the data. A video demonstration of the proposal is presented in [26].

Once validated, several syntheses were performed to analyze the influence of this proposal in relation to area

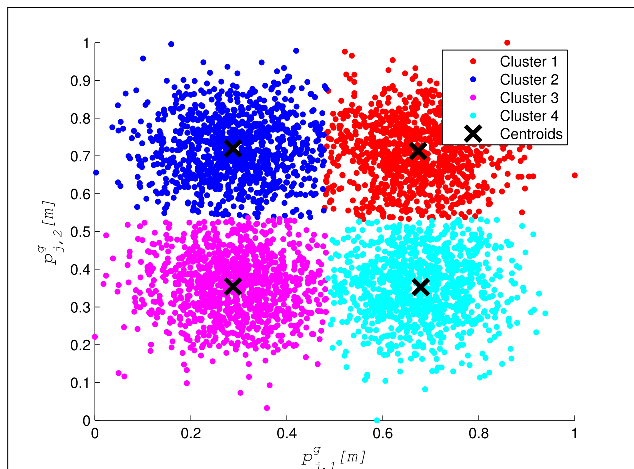

FIGURE 7. Clustered two-dimensional gaussian dataset after K-means runtime for $K = 4$.

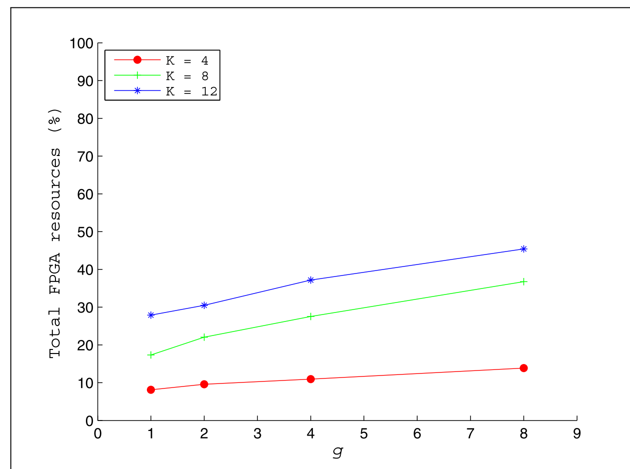

FIGURE 8. Total resources used by varying g and K for $D = 2$ and $m = 14$.

TABLE 1. K-means synthesis on FPGA for $g = 4$, $K = 4$, $D = 2$ and $m = 14$.

Distance	Virtex-6 xc6vlx240t-1ff1156				
	Registers	LUTs	Multipliers	Total resources	R_g (DPS)
MD	530	12258	0	10.02%	53.18M
ED	578	23594	32	18.7%	29.31M
SED	530	12433	32	10.73%	47.69M

occupation and processing time of the FPGA. Firstly, these syntheses were performed for the different distance measurements presented in section II, that is, Manhattan distance (MD), Euclidean distance (ED) and Squared Euclidean distance (SED), respectively. In addition, syntheses were also performed by varying the following parameters: parallelization degree (g), cluster centroid number ($c_k[m]$), data dimension/attributes (D) and bit width (m).

It is presented in Table 1, the synthesis results obtained for each mentioned distance. The parameters were defined as $g = 4$, $K = 4$, $D = 2$ and $m = 14$, for analysis purposes only. From the second to fifth columns are the FPGA resources, that is, the number of registers, LUTs, multipliers and total resources used in the FPGA, respectively, while in the sixth column is the sample rate, R_g. This rate represents the implementation throughput, in other words, the number of data points per second (DPS) processed by the algorithm, implying in the rate that the clusters are updated. Therefore, allowing to estimate the data processing time, according to the following equation

$$R_g = \frac{1}{T_g} * g \tag{14}$$

where T_g is the time required for each n-th iteration.

As can be seen, regarding the area occupation, the number of registers required to implement Manhattan distance is $\approx 8.3\%$ fewer compared to Euclidean distance and equal to Squared Euclidean, while the number of LUTs is $\approx 48\%$ and $\approx 1.4\%$ fewer, respectively. In addition, multipliers are not needed for that distance. This is due to the fact that Manhattan does not require the $SQRT_k$ and $MULT_k$ submodules in the DM module, thus resulting in lower resources used. Concerning processing speed, Manhattan distance allows a processing of more than 53 millions of data points per second, reaching a higher throughput compared to Euclidean and Squared Euclidean, up to 44.88% and 10%, respectively. However, despite the fewer resources required and higher processing speed, Manhattan distance is a less accurate metric [22].

The Euclidean distance is the most complex, requiring $\approx 8.3\%$ more registers and $\approx$ 47% more LUTs than Squared Euclidean and it is also 38.5% slower (lower throughput). This is due to the complexity of square root function, the operation performed in $SQRT_k$ submodule. On the other side, the number of multipliers is equal as both distances has it defined according to equation (7). Nevertheless, Squared Euclidean distance showed a better tradeoff for accuracy, area occupation and processing speed.

Despite that, it is important to emphasize that there are unused FPGA resources. Even for Euclidean distance, the most complex metric, only 18.7% of the total FPGA resources has been used. Hence, the parallelization degree, g, can be increased to achieve even higher processing speed and also implement different systems.

Afterward, syntheses were performed varying the parameters, which in turn, had its values defined based on configurations of previous experiments found in the literature together with some empirically obtained configurations. The benchmark datasets from UCI Machine Learning Repository [27] and School of Computing from the University of Eastern Finland [28], has been chosen for the tests.

Firstly, synthesis were performed varying the parallelization degree (g) for three cluster number: $K = 4$, $K = 8$ and $K = 12$. It is shown in Figure 8, the total resources used in the FPGA for $g = 1$, $g = 2$, $g = 4$ and $g = 8$, respectively. The remaining parameters were set to $D = 2$ and $m = 14$.

As can be observed, the resources used increases as cluster number and parallelization degree increases, but not linearly.

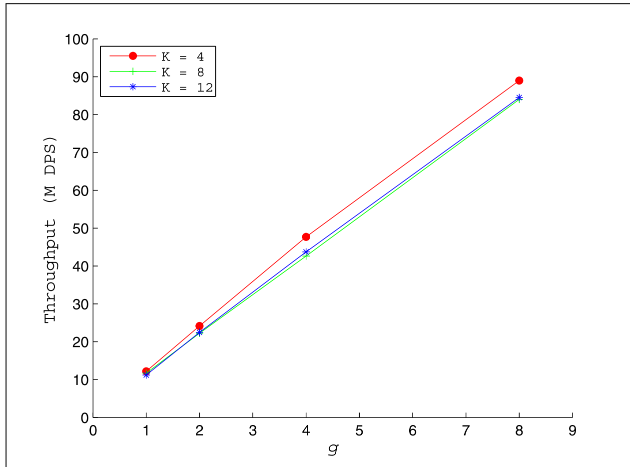

FIGURE 9. Throughput obtained by varying g and K for $D = 2$ and $m = 14$.

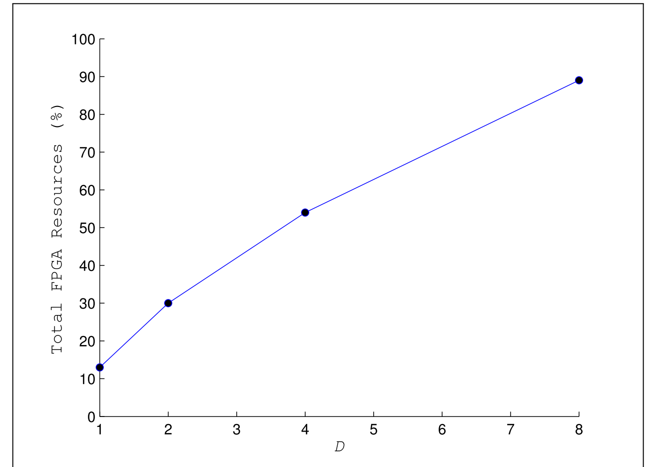

FIGURE 10. Total resrouces used by varying D for $g = 4$, $K = 8$ and $m = 16$.

Regarding parallelization degree, for $K = 4$, the total resources increases from 8.14% to 13.87% by doubling g, while for $K = 8$ and $K = 12$ the resources used increases from 17.35% to 36.76% and 27.89% to 45.42%, respectively. As mentioned in section III, this parameter, allows to process G data points, $p_j^g[m]$, in parallel, at each n-th iteration. Hence, increasing g results in an increased number of DM and CP modules, thus the total resources. Concerning cluster number, a resource increase for $g = 1$ from 8.14% to 27.89% is obtained with the increase of K from 4 to 12, while for $g = 8$ is obtained an increase from 13.87% to 45.42%. This is due to the fact that, the submodules of each g-th DM and CP modules are replicated in order to process every g-th data point, $p_j^g[m]$, regarding each k-th centroid, $c_k[m]$, in parallel, at each n-th iteration. In addition to that, the submodules of MC modules are also replicated to update each k-th centroid also in parallel. Thereby, the number of modules and submodules are increased with these parameters resulting in that increased number of resources.

These parameters also influence the throughput, as shown in Figure 9. The throughput for $g = 1$ slightly decreased from $R_g = 12.18$M DPS to $R_g = 11.69$M DPS and $R_g = 11.20$M DPS for $K = 4$, $K = 8$ and $K = 12$, respectively. While for $g = 8$, the respective throughputs obtained was $R_g = 88.99$M DPS, $R_g = 84.01$M DPS and $R_g = 84.50$M DPS. However, for a fixed cluster number, an increase in g resulted in a higher throughput. As can be observed, the throughput obtained for $K = 4$ was $R_g = 12.18$M DPS, $R_g = 24.17$M DPS, $R_g = 47.70$M DPS and $R_g = 88.99$M DPS, for $g = 1$, $g = 2$, $g = 4$ and $g = 8$, repectively, while for $K = 12$, it was obtained $R_g = 11.20$M DPS, $R_g = 22.48$M DPS, $R_g = 43.73$M DPS and $R_g = 84.50$M DPS. Thereby, double g almost doubled R_g. This increase in processing speed, resulting from the increase in g, is not linear due to the join between each g-th CP and MC modules, as shown in Figure 1. What also slightly affect the speed when varying the cluster number, but not drastically as the implementation is completely parallel.

Therefore, using parallel techniques allows reducing the processing time by increasing the number of data points processed simultaneously.

Secondly, synthesis were also performed for different data dimension/attributes (D), while the other parameters were set to $g = 4$, $K = 8$, and $m = 16$. It is shown in Figure 10, the total resources used for $D = 1$, $D = 2$, $D = 4$ and $D = 8$ respectively. As can be observed, it grows from 13% to 89% with the increase of dimensions, but not linearly. This is due to the fact that, the number of submodules present in each module increase, such as the amount of $SUB_{k,D}$, $MULT_{k,D}$ and SUM_k in each g-th DM module.

Regarding processing speed, a throughput of $R_g = 42.53$M DPS, $R_g = 38$M DPS, $R_g = 33.33$M DPS and $R_g = 31.96$M DPS was obtained for $D = 1$, $D = 2$, $D = 4$ and $D = 8$, respectively, as shown in Figure 11. Thereby, increasing the number of dimensions increases the critical path and reduce the processing speed, as there is an increase in the total of adders submodules, SUM_k, in each g-th DM module, as well as $SUM - N^k$ and $SUM - D^k$, in MC module.

Finally, synthesis were performed for different bit widths (m), while the other parameters were set to $g = 1$, $K = 4$ and $D = 2$. It is shown in Figure 12, the total resources used in the FPGA for $m = 8$, $m = 14$, $m = 16$ and $m = 20$, respectively. Each synthesis were performed for a total of $m - 2$ bits to the fractional part. As can be observed, the area occupation grows only from 7% to 9% with the increase of bits, which is not a significant increase compared to varying the other parameters.

Concerning processing speed, a throughput of $R_g = 11,93$M DPS, $R_g = 11,49$M DPS, $R_g = 11,33$M DPS and $R_g = 10,70$M DPS was obtained for $m = 8$, $m = 14$, $m = 16$ and $m = 20$, respectively, as shown in Figure 13. The processing speed wasn't significantly afected. Despite this, it is important to emphasize that reducing bit width prevents the algorithm from converging to a local minima. For $m = 8$, the algorithm didn't converged.

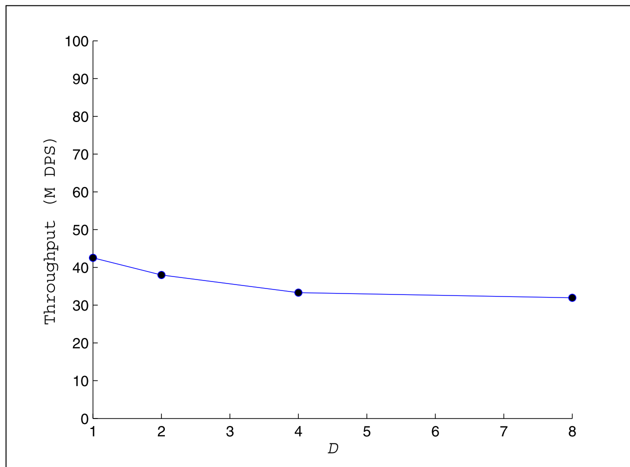

FIGURE 11. Throughput obtained by varying D for $g = 4$, $K = 8$ and $m = 16$.

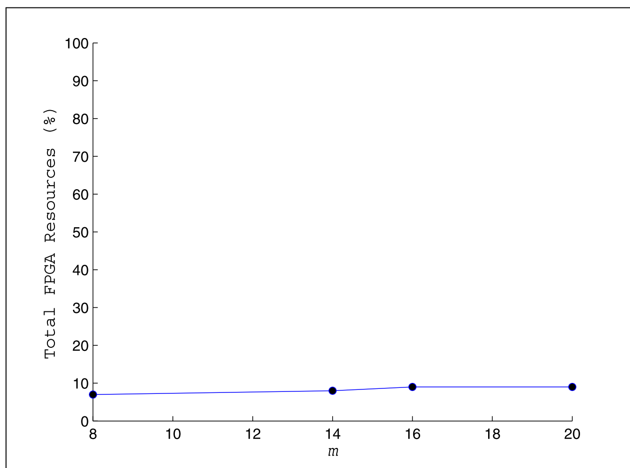

FIGURE 12. Total resources used by varying m for $g = 1$, $K = 4$ and $D = 2$.

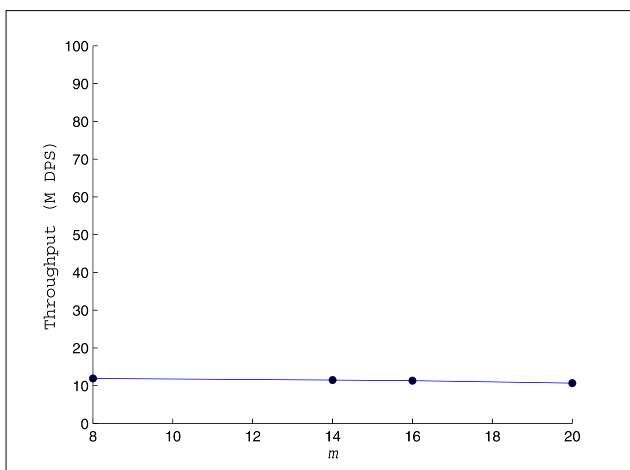

FIGURE 13. Throughput obtained by varying m for $g =$, $K = 4$ and $D = 2$.

Therefore, it is clear that the use of parallel techniques increase the required resources, but also increases processing speed significantly. Given that, the resources available in the FPGA is the only limiting factor to increase processing speed to the desired level. Despite this, given the simplicity and high degree of parallelization of the algorithm and proposed implementation, it is possible to process a significant amount of data in a short time making the application feasible for real-time processing. Although there isn't a test performed that used all resources, g, K and D are the parameters that most increase the area occupation.

TABLE 2. Comparison of area overhead and processing speed for $g = 8$, $K = 8$ and $D = 1$.

Reference		[4]	AUTOR
Area ocupation	Registers	14132	512
	LUTs	15009	18450
	Multipliers	88	64
Processing Speed	T_g	$1350,37\mu s$	$0,085611\mu s$
	R_g (DPS)	0,006M	93.44M

A. COMPARISONS WITH THE STAT OF THE ART WORKS

Following, results obtained with this implementation is compared to equivalent results found in works present in state of the art. These comparisons were made with the greatest similarity of parameters as possible.

It is presented in Table 2 a comparison with the implementation proposed in [4], developed in a Virtex-6 xc6vlx240t FPGA, the same as this proposal. In the second to third rows are presented the FPGA resources used, that is, the number of registers, LUTs, and multipliers, while in the fourth and fifth are shown the time per iteration and the throughput, respectively, according to equation 14. This comparison was performed for the "Wholesale customer dataset" from UCI Machine Learning Repository, and the parameters set to $g = 8$, $K = 8$ and $D = 1$ to match those of the work in comparison.

As can be seen, regarding the total resources available on FPGA, this work occupies 96% and 27.3% fewer registers and multipliers, respectively, but 18.6% more LUTs. As mentioned in section I-A, in [4], the step that generates the distance of a j-th data point to each k-th centroid, is obtained in a serial scheme and there is also a constant access to memories, resulting in a high number of registers. While in the implementation proposed here, these steps are completely performed in parallel, resulting in the increased amount of LUTs. Regarding processing time, the time of each n-th iteration is significantly different, also due to sequential processing. Hence, this proposal achieves a speedup of $\approx 15573\times$.

A comparison for area occupation and processing speed were also made about the proposed implementation of [5], for two cluster values, $K = 8$ and $K = 16$, respectively, and it is shown in Table 3. It is presented in the second and third columns the cluster and total resource numbers, while in the fourth and fifth columns are presented the time per iteration and throughput, respectively.

The proposal of [5], for $K = 8$ were developed in a Virtex-4 xc4vfx12 FPGA, the parameters were set to $D = 10$ and $m = 12$, and the algorithm step that updates centroids

TABLE 3. Comparison of area overhead and processing speed for $K = 8$ and $K = 16$.

Reference	K	Total resources	T_g	R_g (DPS)
[5]	8	5107(92%)	$0,07s$	0,0000143M
	16	5177(34%)	$0,0028s$	0,000357M
AUTOR	8	22397(59%)	$0,109733\mu s$	9,11M
	16	19111(51%)	$0,114573\mu s$	8,72M

TABLE 4. Comparison of area overhead for $g = 1$, $K = 12$ and $D = 2$.

Reference	FPGA	Registers	LUTs	R_g (DPS)
[7]	Kintex-7 xc7k325t	183742(45%)	149267(73%)	13.96M
AUTOR	Virtex-6 xc6vlx240t	1728(0,6%)	32585(21%)	11.2M

wasn't developed on the FPGA, just the distance metric and clustering steps. While for $K = 16$, the entire algorithm has been performed on the FPGA and parameters were set to $D = 1$ and $m = 13$. Meanwhile, in the proposal of this work, the entire algorithm were developed in a Virtex-6 xc6vlx240t, and the parameters were set to $g = 1$, $D = 8$ and $m = 16$ for $K = 8$ and $g = 1$, $D = 2$ and $m = 16$ for $K = 16$. As can be observed, for $K = 8$, only 59% of the total resources were occupied in this implementation, while in [5] 92% were occupied, even not implementing the step to update centroids on the FPGA, and for $K = 16$, it was occupied 51% and 34%, respectively. However, since the implementations were developed on different FPGAs, the total resource number cannot be directly compared. In addition, as the Virtex-6 has a more recent FPGA technology and a higher number of logic cells, the implementation presented here requires a significantly higher number of resources. This is due to the high parallelism adopted here, while most steps of the implementation proposed by [5] is performed in a serial manner.

Concerning processing time, each iteration has a time significantly higher in [5], mainly because of the division operation, performed as fixed-point arithmetic for $K = 16$, which generated a clock latency of 84 cycles, and also by performing it for each centroid in a sequential scheme as only one divisor circuit was implemented. In addition to that, for $K = 8$ the step to update centroids is performed in a host and not on FPGA, increasing the time of an iteration due to the constant communication between the FPGA and host. Hence, a significantly low throughput is achieved regarding the implementation proposed here, $\approx 637063\times$ and $\approx 24426\times$ slower, for $K = 8$ and $K = 16$ respectively.

A comparison was also performed for the proposed implementation in [7]. It is shown in Table 4. The second column presents the FPGA used for the implementation, in the third and fourth are presented the number of registers and LUTs, respectively, while the fifth column presents the throughput. The parameters were set to $g = 1$, $K = 12$ and $D = 2$.

As can be observed, the implementation proposed here used only 0, 6% of the total register number, and also 21% of the LUTs, while the proposal of [7] used 45% and 73%, respectively. Since the FPGA used are different, the resources cannot be directly compared, but it is important to emphasize that the Kintex-7 FPGA has more resources and an advanced technology compared to a Virtex-6. Therefore, the resources used for this work is significantly lower. This is due to the extra hardware needed to realize the communication between the host and mapper-reducer FPGAs in [7]. Concerning the processing time, the proposal in [7] is limited by this communication between mapper-reduce FPGAs and achieved a maximum of 13, 96M DPS, only $1.2\times$ faster than the implementation proposed here, even using an advanced FPGA. This is due to the fact that the implementation proposed in [7] obtains the distance metric for each data point about each centroid in a sequential scheme, as the map function, and also use multiple FPGAs.

TABLE 5. Comparison of area overhead and processing speed for $K = 4$.

Reference		[15]	AUTOR
Area ocupation	Registers	208152(47.61%)	565(0,2%)
	LUTs	178185(81.51%)	19567(12%)
	Multipliers	412(45.78%)	128(16%)
Processing Speed	T_g	$8500\mu s$	$0,098\mu s$
	R_g (DPS)	0,0014M	81.63M

Another comparison was performed for the proposed implementation in [15] and shown in Table 5. In the second to third rows are presented the FPGA resources used, that is, the number of registers, LUTs, and multipliers, while in the fourth and fifth are shown the time per iteration and the throughput, respectively. The parameters in this paper proposal were set to $g = 8$, and $D = 2$ while the work being compared were set to $g = 12$, and $D = 4$, both for $K = 4$.

As can be observed, the implementation proposed here used only 0, 2% of the total register number, and also 12% of the LUTs, while the proposal of [15] used 47.61% and 81.51%, respectively. Since the FPGA used are different, the resources cannot be directly compared, but it is important to emphasize that the compared proposal has a high number of dimensions and data points being processed and that Virtex-6 FPGA has more resources and an advanced technology. Despite that, this area overhead in [15] is due to extra hardware needed to realize the control and communication between mapper-reducer circuits and memories. Concerning the processing time, the proposal in [15] is limited by the communication and control between mapper-reducer circuits and the constant access to memories in between them, achieving a maximum of 0, 0014M DPS, $\approx 58307\times$ slower than the implementation proposed here, even processing more data points per iteration.

V. CONCLUSION

This work presented a parallel implementation of the K-means algorithm on an FPGA. All implementation details of the proposal were presented and analyzed in terms of processing speed and hardware area occupancy.

Based on the obtained results, it can be affirmed that the implementation proposed was in fact validated and fulfill its objective of being a parallel implementation of high performance of a K-means algorithm.

The synthesis results confirmed that the present proposed parallel implementation of K-means on FPGA is able to optimize, in a viable time, critical applications that require short time constraints or a large amount of data to be processed in a short interval and even for real-time applications, reaching throughputs higher than 53 millions of data points processed per second (M DPS).

Beyond the high performance achieved, since the implementations do not occupy the total resources of the FPGA, it is possible for other systems to also be embedded in the FPGA, or increase the parallelization degree increasing even further processing time.

Comparisons with state of the art were also discussed, showing that fully parallel implementation can achieve high throughputs compared to proposals that involve sequential schemes.

The experiments carried out can help future implementations of the K-means algorithm, to easily choose the best distance metric as well as the degree of parallelization, and even the parameter m, allowing to level, according to the desired purpose, the speed processing, and hardware area.

REFERENCES

[1] I. Yaqoob, I. Hashem, A. Gani, S. Mokhtar, E. Ahmed, N. Anuar, and A. Vasilakos, "Big data: From beginning to future," *Int. J. Inf. Manage.*, vol. 36, no. 6, pp. 1231–1247, Dec. 2016.

[2] S. Ayani, K. Moulaei, S. D. Khanehsari, M. Jahanbakhsh, and F. Sadeghi, "A systematic review of big data potential to make synergies between sciences for achieving sustainable health: Challenges and solutions," *Appl. Med. Inform.*, vol. 41, no. 2, pp. 53–64, 2019.

[3] N. Koseleva and G. Ropaite, "Big data in building energy efficiency: Understanding of big data and main challenges," *Procedia Eng.*, vol. 172, pp. 544–549, 2017.

[4] R. Raghavan and D. G. Perera, "A fast and scalable FPGA-based parallel processing architecture for K-means clustering for big data analysis," in *Proc. IEEE Pacific Rim Conf. Commun., Comput. Signal Process. (PACRIM)*, Aug. 2017, pp. 1–8.

[5] H. M. Hussain, K. Benkrid, A. T. Erdogan, and H. Seker, "Highly parameterized K-means clustering on FPGAs: Comparative results with GPPs and GPUs," in *Proc. Int. Conf. Reconfigurable Comput. (FPGAs)*, Nov. 2011, pp. 475–480.

[6] G. Venkatesh and K. Arunesh, "Map reduce for big data processing based on traffic aware partition and aggregation," *Cluster Comput.*, vol. 22, no. S5, pp. 12909–12915, Feb. 2018.

[7] Y.-M. Choi and H. K.-H. So, "Map-reduce processing of K-means algorithm with FPGA-accelerated computer cluster," in *Proc. IEEE 25th Int. Conf. Appl.-Specific Syst., Archit. Processors*, Jun. 2014, pp. 9–16.

[8] V. Tirumalai, K. G. Ricks, and K. A. Woodbury, "Using parallelization and hardware concurrency to improve the performance of a genetic algorithm," *Concurrency Comput., Pract. Exper.*, vol. 19, no. 4, pp. 443–462, 2007.

[9] N. Instruments. (2011). *Understanding parallel hardware: Multiprocessors, hyperthreading, dual-core, multicore and fpgas*. [Online]. Available: http://www.ni.com/tutorial/6097/en/

[10] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical Machine Learning Tools and Techniques*. San Mateo, CA, USA: Morgan Kaufmann, 2016.

[11] K. M. A. Patel and P. Thakral, "The best clustering algorithms in data mining," in *Proc. Int. Conf. Commun. Signal Process. (ICCSP)*, Apr. 2016, pp. 2042–2046.

[12] J. Canilho, M. Vestias, and H. Neto, "Multi-core for K-means clustering on FPGA," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2016, pp. 1–4.

[13] C.-C. Chung and Y.-H. Wang, "Hadoop cluster with FPGA-based hardware accelerators for K-means clustering algorithm," in *Proc. IEEE Int. Conf. Consum. Electron. Taiwan (ICCE-TW)*, Jun. 2017, pp. 143–144.

[14] Mahout. (2018). *Mahout-Machine Learning Applications*. Accessed: May 31, 2018. [Online]. Available: https://mahout.apache.org/

[15] Z. Li, J. Jin, and L. Wang, "High-performance K-means implementation based on a simplified map-reduce architecture," 2016, *arXiv:1610.05601*. [Online]. Available: http://arxiv.org/abs/1610.05601

[16] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable K-means++," *Proc. VLDB Endowment*, vol. 5, no. 7, pp. 622–633, 2012.

[17] K. Neshatpour, A. Koohi, F. Farahmand, R. Joshi, S. Rafatirad, A. Sasan, and H. Homayoun, "Big biomedical image processing hardware acceleration: A case study for K-means and image filtering," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2016, pp. 1134–1137.

[18] P. Arora, Deepali, and S. Varshney, "Analysis of K-Means and K-Medoids algorithm for big data," *Procedia Comput. Sci.*, vol. 78, pp. 507–512, Jan. 2016.

[19] H. M. Hussain, K. Benkrid, H. Seker, and A. T. Erdogan, "FPGA implementation of K-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data," in *Proc. NASA/ESA Conf. Adapt. Hardw. Syst. (AHS)*, Jun. 2011, pp. 248–255.

[20] K. Neshatpour, M. Malik, and H. Homayoun, "Accelerating machine learning kernel in Hadoop using FPGAs," in *Proc. 15th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2015, pp. 1151–1154.

[21] Z. Kakushadze and W. Yu, "K-means and cluster models for cancer signatures," *Biomolecular Detection Quantification*, vol. 13, pp. 7–31, Sep. 2017.

[22] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski, "Algorithmic transformations in the implementation of K-means clustering on reconfigurable hardware," in *Proc. ACM/SIGDA 9th Int. Symp. Field Program. Gate Arrays (FPGA)*. New York, NY, USA: ACM, 2001, pp. 103–110, doi: 10.1145/360276.360311.

[23] N. Nedjah and L. de M. Mourelle, "An efficient problem-independent hardware implementation of genetic algorithms," *Neurocomputing*, vol. 71, nos. 1–3, pp. 88–94, Dec. 2007.

[24] Xilinx. (2008). *System Generator for DSP*. Accessed: May 31, 2015. [Online]. Available: http://www.xilinx.com

[25] A. Suzuki, T. Morie, and H. Tamukoh, "A shared synapse architecture for efficient FPGA implementation of autoencoders," *PLoS ONE*, vol. 13, no. 3, Mar. 2018, Art. no. e0194049.

[26] L. A. Dias. (2019). *Video Demonstration*. Accessed: Oct. 21, 2019. [Online]. Available: https://drive.google.com/file/d/1iAScU-r772wNHYl72xk9H3RuqcHQartD/view

[27] M. Cardoso. (2014). *Wholesale Customer Data Set*. Accessed: Apr. 10, 2018. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/wholesale+customers

[28] *Clustering Basic Benchmark*. Accessed: 2015. [Online]. Available: http://cs.uef.fi/sipu/datasets/

LEONARDO A. DIAS was born in Sousa, Brazil. He received the Associate degree in industrial automation from the Federal Institute of Paraíba, in 2014, and the M.S. degree in electrical engineering from the Federal University of Paraíba, Paraíba, Brazil, in 2016. He is currently pursuing the Ph.D. degree in electrical engineering with the Federal University of Rio Grande do Norte, Natal, Brazil. He is also a part of the Research Group on Embedded Systems and Reconfigurable Hardware, where the main research topic is the acceleration of clustering algorithms through reconfigurable computing (RC) in FPGA. His research interests include artificial intelligence, embedded systems, and reconfigurable hardware.

JOÃO C. FERREIRA (Senior Member, IEEE) received the Licenciatura and Ph.D. degrees in electrical and computer engineering from the University of Porto, Porto, Portugal, in 1989 and 2001, respectively. He is currently an Associate Professor with the Faculty of Engineering, University of Porto. He is also a Senior Researcher with the Instituto de Engenharia de Sistemas e Computadores—Tecnologia e Ciência, Porto. His current research interests include dynamically reconfigurable systems, application-specific architectures for cognitive radio and sensor networks, and adaptive embedded systems. Dr. Ferreira is also a member of ACM and Euromicro.

MARCELO A. C. FERNANDES was born in Natal, Brazil. He received the B.S. degree in electrical engineering and the M.S. degree in electrical engineering from the Federal University of Rio Grande do Norte, Natal, in 1997 and 1999, respectively, and the Ph.D. degree in electrical engineering from the University of Campinas, Campinas, Brazil, in 2010. From 2015 to 2016, he was a Visiting Researcher with the Centre Telecommunication Research (CTR), King's College London, London, U.K. He is currently an Associate Professor with the Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte. He is also a Visiting Scholar with the John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, USA. He is also the Leader of the Research Group on Embedded Systems and Reconfigurable Computing (RESRC) and, coordinator of the Laboratory of Machine Learning and Intelligent System (LMLIS). He is the author and coauthor of many scientific articles and practical studies with reconfigurable computing on FPGA to accelerate artificial intelligence algorithms. His research interests include artificial intelligence, digital signal processing, embedded systems, reconfigurable hardware, and the tactile Internet.