Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 135926, 15 pages
doi:10.1155/2012/135926

*Research Article*

# Novel Dynamic Partial Reconfiguration Implementation of K-Means Clustering on FPGAs: Comparative Results with GPPs and GPUs

**Hanaa M. Hussain,[1] Khaled Benkrid,[1] Ali Ebrahim,[1] Ahmet T. Erdogan,[1] and Huseyin Seker[2]**

[1] *School of Engineering, University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh EH9 3JL, UK*
[2] *Bio-Health Informatics Research Group, Centre for Computational Intelligence, De Montfort University, Leicester LE1 9BH, UK*

Correspondence should be addressed to Hanaa M. Hussain, h.hussain@ed.ac.uk

Received 3 May 2012; Revised 17 September 2012; Accepted 3 October 2012

Academic Editor: René Cumplido

K-means clustering has been widely used in processing large datasets in many fields of studies. Advancement in many data collection techniques has been generating enormous amounts of data, leaving scientists with the challenging task of processing them. Using General Purpose Processors (GPPs) to process large datasets may take a long time; therefore many acceleration methods have been proposed in the literature to speed up the processing of such large datasets. In this work, a parameterized implementation of the K-means clustering algorithm in Field Programmable Gate Array (FPGA) is presented and compared with previous FPGA implementation as well as recent implementations on Graphics Processing Units (GPUs) and GPPs. The proposed FPGA has higher performance in terms of speedup over previous GPP and GPU implementations (two orders and one order of magnitude, resp.). In addition, the FPGA implementation is more energy efficient than GPP and GPU (615x and 31x, resp.). Furthermore, three novel implementations of the K-means clustering based on dynamic partial reconfiguration (DPR) are presented offering high degree of flexibility to dynamically reconfigure the FPGA. The DPR implementations achieved speedups in reconfiguration time between 4x to 15x.

## 1. Introduction

Current technologies in many fields of studies have been utilizing advanced data collection techniques which output enormous amount of data. Such data may not be useful in their collected form unless they are computationally processed to extract meaningful results. Current computational power of General Purpose Processors (GPPs) has not been able to keep up with the pace at which data are growing [1]. Therefore, researchers have been searching for methods to accelerate data analysis to overcome the limitation of GPPs, one of which is the use of hardware in the form of Field Programmable Gate Arrays (FPGAs).

K-means clustering is one of the widely used data mining techniques to analyze large datasets and extract useful information from them. Previously, we have implemented the K-means clustering on FPGA to target Microarray gene expression profiles and reported encouraging speedups over GPPs [2]. However, the implementation was limited to single dimension and eight clusters only. An extended work on FPGA implementation of the K-means algorithm is presented in this work which includes a highly parameterized architecture, a novel single, and a multicore dynamic partial reconfiguration of the K-means algorithm. Although the proposed design is meant to target Microarray gene expression profiles, it can be adopted for use in other applications such as image segmentation. In this work we also compare our parameterized implementation with other FPGA implementation of the K-means algorithms. Furthermore, we will compare the performance of our design with one that has been recently implemented on Graphics Processing Units (GPUs), a technology that has been gathering a lot of interest in the computing community because of its high performance and relatively low cost.

The remainder of this paper is organized as follows. In Section 2 an overview about the K-means clustering algorithm is given. In Section 3, related works on K-means clustering acceleration are summarized which include both FPGA and GPU methods. In Section 4 the hardware implementation of the parameterized K-means clustering is presented. Then in Section 5 three novel implementations of the K-means clustering based on dynamic partial reconfiguration are given: the first is based on reconfiguring the distance kernel within the algorithm with different distance metrics, the second is based on reconfiguring the K-means core using Internal Configuration Access Port (ICAP), and the last one is based on reconfiguring multiple K-means cores also using ICAP. In Section 6 implementations results are presented and analyzed. Finally, in Section 7 summary of findings, discussion and conclusion are presented along with some remarks on future work.

## 2. K-Means Clustering

K-means clustering is one of the unsupervised data mining techniques used in processing large datasets by grouping objects into smaller partitions called clusters, where objects in one cluster are believed to share some degree of similarity. Clustering methods help scientists in many fields of studies in extracting relevant information from large datasets. To arrange the data into partitions, at first one needs to determine the number of clusters beforehand and initialize centers for each cluster from the dataset. There are several ways for doing this initialization, one way is by randomly assigning all points in the overall dataset to one of these clusters, then calculate the mean of each cluster and use the results as the new centers. Another way is to randomly select cluster centers from the whole dataset. The distance between each point in the dataset and every cluster center is then calculated using a distance metric (e.g., Euclidean, Manhattan). Then, for every data point, the minimum distance to all cluster's centers is determined and the point gets assigned to the closest cluster. This step is called cluster assignment and is repeated until all of the data points have been assigned to one of the clusters. Finally, the mean of each cluster is calculated based on the accumulated points and the number of points in that cluster. Those means become the new cluster's centers, and the process iterates for a fixed number of times, or until points in each cluster stop moving across to different clusters; Figure 1 illustrates the steps of the K-means algorithm.

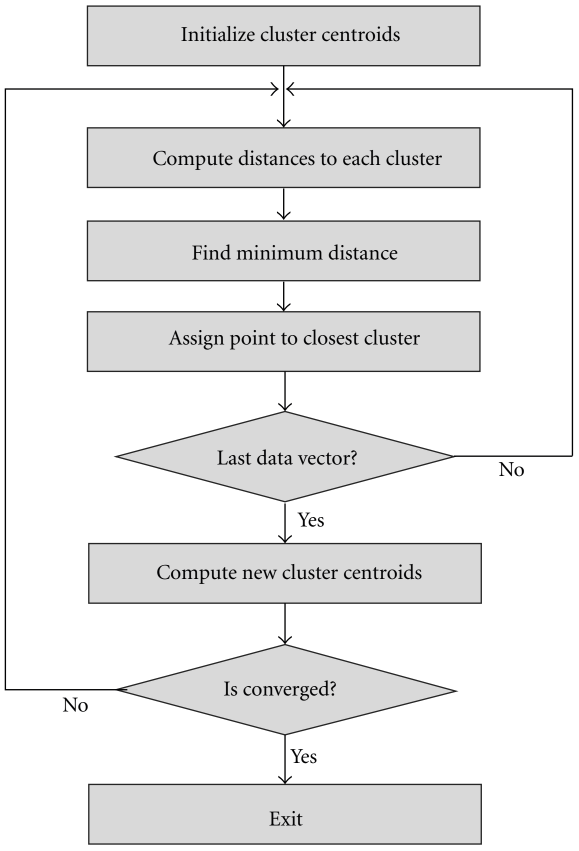

Initialize cluster centroids
Compute distances to each cluster
Find minimum distance
Assign point to closest cluster
Last data vector?
No
Yes
Compute new cluster centroids
Is converged?
No
Yes
Exit

Figure 1: The K-means algorithm.

The Euclidean metric given in (1) is widely used with K-means clustering and one that results in better solutions [3]:

$$D(P,C) = \sqrt{\sum_{i}^{M}(P_i - C_i)^2}, \quad (1)$$

where $P$ is the data point, $C$ is the cluster center, and $M$ is the number of features or dimensions. On the other hand, Euclidean distance consumes a lot of logic resources when implemented in hardware due to the multiplication operation used for obtaining the square operation. Therefore, previous groups working on hardware implementation of the K-means clustering for image segmentation have used the Manhattan distance shown in (2) as an alternative to the Euclidean distance:

$$D(P,C) = \sum_{i}^{M}|P_i - C_i|, \quad (2)$$

where $P$ is again the data point, $C$ is the cluster center, and $M$ is the number of features. Their results showed that it performed twice as fast as that obtained by Euclidean distance [3–9].

Distance computation is the most computationally demanding part and where most of the K-means processing time occurs. Therefore, accelerating K-means algorithm can be achieved by mainly accelerating the distance computation part, which is achieved using hardware.

## 3. Related Works on the Acceleration of K-Means Clustering

K-means has already been implemented in hardware by several groups; most were to target applications in hyperspectral imaging, or image segmentation. The following review will cover some of the work done on K-means clustering acceleration using FPGA and GPUs.

*3.1. K-Means Clustering Acceleration on FPGAs.* In 2000, Lavenier implemented systolic array architecture for K-means clustering [6]. He moved the distance calculation part to FPGA while keeping the rest of the K-means tasks on GPP. The distance computation involved streaming the input through an array of Manhattan distance calculation units of numbers equal to the number of clusters and obtaining the cluster index at the end of the array. The disadvantage of this approach was the communication overhead between the host and the FPGA. Lavenier tested his design on several processing boards, and one of the relevant speedups obtained compared to GPP was 15x [6, 7]. In addition, he found that the speedup of the systolic array was the function of the number of clusters and transfer rated between the host and the FPGA.

Between 2000 and 2003, Leeser et al. reported several works related to K-means implementation on FPGA, based on a software/hardware codesign approach [3–5]. Their design was partitioned between FPGA hardware and a host microprocessor, where distance calculation and data accumulation were done in hardware in purely fixed point while new means were calculated in the host to avoid consuming large hardware resources. They achieved a speedup of 50x over pure GPP implementation. Their design benefited from two things: the first was using Manhattan distance metric instead of the commonly used Euclidean metric to reduce the amount of hardware resources needed, and the second was truncating the bit width of the input data without sacrificing accuracy [8].

In 2003, Bhaskaran [9] implemented a parameterized design of the K-means algorithm on FPGA where all the K-means tasks were done in hardware, except the initialization of cluster centers which was done on a host. This design implemented the division operation within FPGA hardware to obtain the new means, using dividers from Xilinx Core Generator. However, this design was tested only on three clusters and achieved a speedup of 500x over Matlab implementation including I/O overhead [9].

*3.2. K-Means Clustering Acceleration on GPUs.* Several implementations of K-means clustering using Graphics Processing Units (GPUs) have been reported in the literature. In 2008, Fairvar implemented K-means on GPU and achieved speedup of 13.57x. (The GPU implementation took 0.724 s compared to 9.830 s on GPP) when clustering 1 million points into 4000 clusters using Nvidia's GeForce 8600 GT and a 2-GHz GPP host [10]. Another group [11] presented good results when implementing K-means using two types of GPUs. The speedups they achieved when clustering 200 K to 1 M on the Nvidia's GeForce 5900 were between 4x to 12x more than a Pentium 4, 1.5 GHz CPU, and up to 30x when using Nvidia's GeForce 8500 and Pentium 4, 3 GHz CPU. They also found that GPU performance was less affected by the size of the dataset as compared to GPPs. Another result reported by this group was related to the effect of the number of clusters on speedup, where achieved speedups were between 10x and 20x for clusters less than 20 using the Nvidia's GeForce 5900 GPU and more than 50x when there were more than 20 clusters. For 32 clusters, they reported a speedup of 130x on the Nvidia's GeForce 8500 GPU.

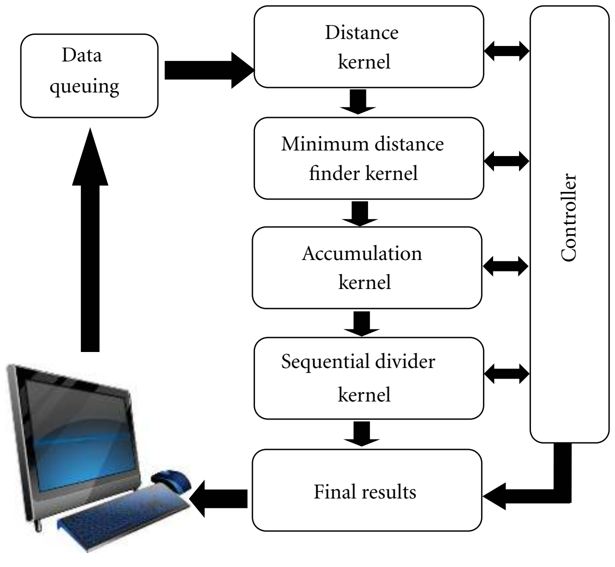


Figure 2: The main blocks of the K-means hardware design.

In 2010, Karch reported a GPU implementation of K-means clustering for accelerating color image segmentation in RGB space [12]. We compared the results published in [12] with our FPGA implementation and shall present this comparison in the results section. Furthermore, Choudhary et al. reported another GPU implementation using Nvidia's GeForce 8800 GT, which achieved speedups between 9x and 40x for datasets ranging from 10,000 to one million points when they were clustered into 20 partitions [13]. The group also reported that speedups of GPUs over GPPs increase as datasets grow largely in depth. From the above studies, it can be stated that GPUs outperform GPPs when datasets are large or when the number of clusters is large.

## 4. FPGA Hardware Design of the K-Means Clustering

In this work, a highly parameterized hardware design of the K-means algorithm is presented, which aims to carry all K-means tasks in hardware. The architecture of the whole design consists of a number of blocks which execute kernels within the K-means algorithm. The design generates the required hardware resources and logics based on the parameters entered by the user at compile time. These parameters are the wordlength of the input ($B$), number of clusters ($C$), number of data points ($N$), and dimensions of the input data ($M$). The design was captured in Verilog HDL language. Figure 2 summarizes the main kernels of the K-means clustering, which will be used to form a modular architecture. The design intends to perform all the K-means steps within the FPGA, including the division operation, and avoids directing any task to an offchip resource although this capability can be exploited when needed. In the following sections an overview about each block will be presented along with the timing of each block.

*4.1. Distance Kernel Block.* This block receives streaming input data stored in onchip Block RAMs or from an off-chip memory, along with the initialized or updated cluster's centers, and computes the distances between each data point and all clusters simultaneously. The hardware resources inferred by the synthesis tool to generate multiple distance processors (DPs) are based on the number of clusters and the number of dimensions of the dataset. Each DP is responsible for the computation of the distance between all the input dimensions and one of the cluster centers. Thus one DP is required for each cluster. The DPs work simultaneously such that the distances between every point to all clusters are computed in a few clock cycles, hence fully exploiting the parallelism associated with the distance calculation kernel. In this work, both the Euclidean and the Manhattan distance metrics were implemented; however, the Manhattan distance is chosen often times to simplify the computation and save logic resources; thus, the reported results are based on the Manhattan distance implementation. The datapath of this block depends on the number of dimensions ($M$) in the dataset as given by

$$\text{datapath of distance kernel} = \text{ceil}\left[\log_2(M)\right], \quad (3)$$

which corresponds to the stages needed to compute the distances between all the dimensions of the input and the clusters' centroids. Accordingly, the computation time for a single pass through the whole dataset is a function of the number of data points ($N$) in the set and the datapath, as shown in

$$\text{distance computation time} = \text{ceil}\left[\log_2(M)\right] + N. \quad (4)$$

This kernel has a throughput of 1 data point per clock cycle and a latency of $\text{ceil}[\log_2(M)]$ clock cycles, which is the same as the datapath of the distance kernel, this latency is one clock cycle only for the case of single dimension. Since there are $C$ DPs working simultaneously, the total outputs of this block are $C$ distances, each corresponding to the sum of distances between all the dimensions of one point vector and one cluster center resulting from a single DP.

*4.2. Minimum Distance Finder Kernel Block.* This block has the role of comparing the $C$ distances received from the previous block to determine the minimum distance and the associated index which correspond to the ID of the closest cluster to the data point. The block consists mainly of a comparator tree as shown in Figure 3, which has number of stages dependent on the number of clusters ($C$) as given by

$$\text{datapath of min. dist. finder kernel} = \text{ceil}\left[\log_2(C)\right]. \quad (5)$$

This block is pipelined to have throughput of one result per clock cycle, with latency equivalent to (5). The combined execution time of the above two blocks consisting of the distance computation and the minimum distance finder can be summarized in (6):

$$\begin{aligned}&\text{min. distance computation time}\\ &\quad = \text{ceil}\left[\log_2(M)\right] + \text{ceil}\left[\log_2(C)\right] + N.\end{aligned} \quad (6)$$

*4.3. Accumulation Kernel Block.* This block is responsible for accumulating the data points in the accumulator corresponding to a specific cluster index and incrementing the corresponding counter, keeping track of the number of points in each cluster along with the values of these points. The block receives the data point under processing along with the index of the cluster having the minimum distance, which was obtained from the previous block, and performs the accumulation and counting accordingly. The number of accumulators inferred by the HDL code is as shown in (7):

$$\text{accumulator numbers} = C \times M, \quad (7)$$

such that each cluster has $M$ number of accumulators associated with it. As for the inferred number of counters, it is equal to the number of clusters only, so that each cluster has a counter associated with it. Since a fixed point arithmetic is used in this work, extra care was taken in choosing the appropriate wordlength ($B$) for the distance, accumulator, and counter results to minimize hardware resources and avoid data overflow. The latter is achieved through error and range analysis of data in all blocks. Based on the $B$ of the input selected initially to represent the data and the number of points in the dataset to be processed, the accumulator and counter $B$'s were calculated as follows, respectively:

$$B_{\text{Accumulator}} = \log_2\left[\left(\text{Max. range of } B_{\text{Input}}\right) \times N\right], \quad (8)$$

$$\text{counter size} = \log_2[N]. \quad (9)$$

In the case of Microarray datasets for instance, data usually do not exceed 25,000 points for Human. Hence, 15 bits are found to be sufficient for each counter and 32 bits for each accumulator given that the input point is represented by 13 bits. In general, a program was written in Matlab to automate the range and error analysis to make the process of selecting the best wordlengths easy and efficient.

*4.4. Sequential Divider Kernel Block.* The divider kernel block is responsible for receiving results from the accumulation kernel and calculating the new cluster centers. The divider itself was generated using Xilinx Core Generator tool, which was found to be faster due to high pipelining when compared to other dividers. In the case of Microarray data, for instance, the generated divider uses 32 bits for the dividend and 15 bits for the divisor. These values were chosen based on the results of the accumulator/counter blocks as the role of the divider is to divide each accumulator result over the corresponding counter to obtain the new cluster centers. Once signaled to start, this block starts scheduling the data received to be serviced by the divider core serially as illustrated in Figure 4. The number of divisions that needs to be performed is the same as the number of clusters specified in the design parameters. After all the divisions are completed, results are packed to the output port of the divider block. The number of clock cycles taken by the divider to complete its work is a

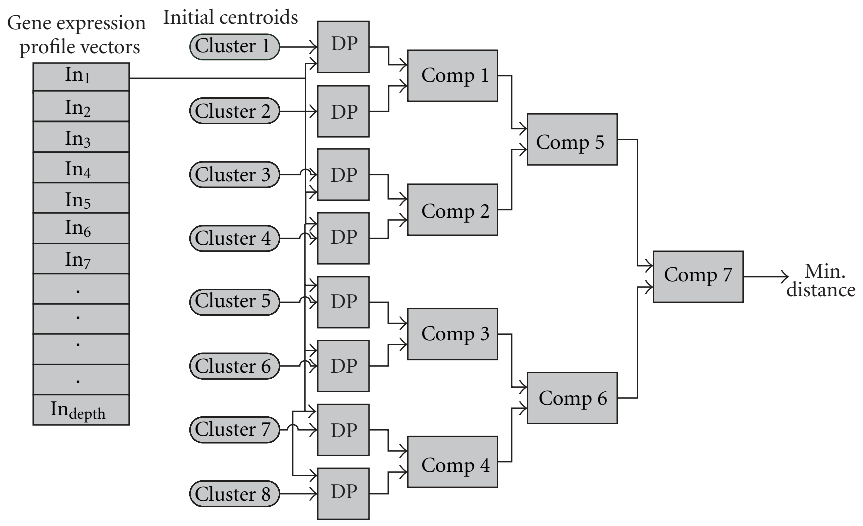


Figure 3: Block diagram of the minimum distance finder.

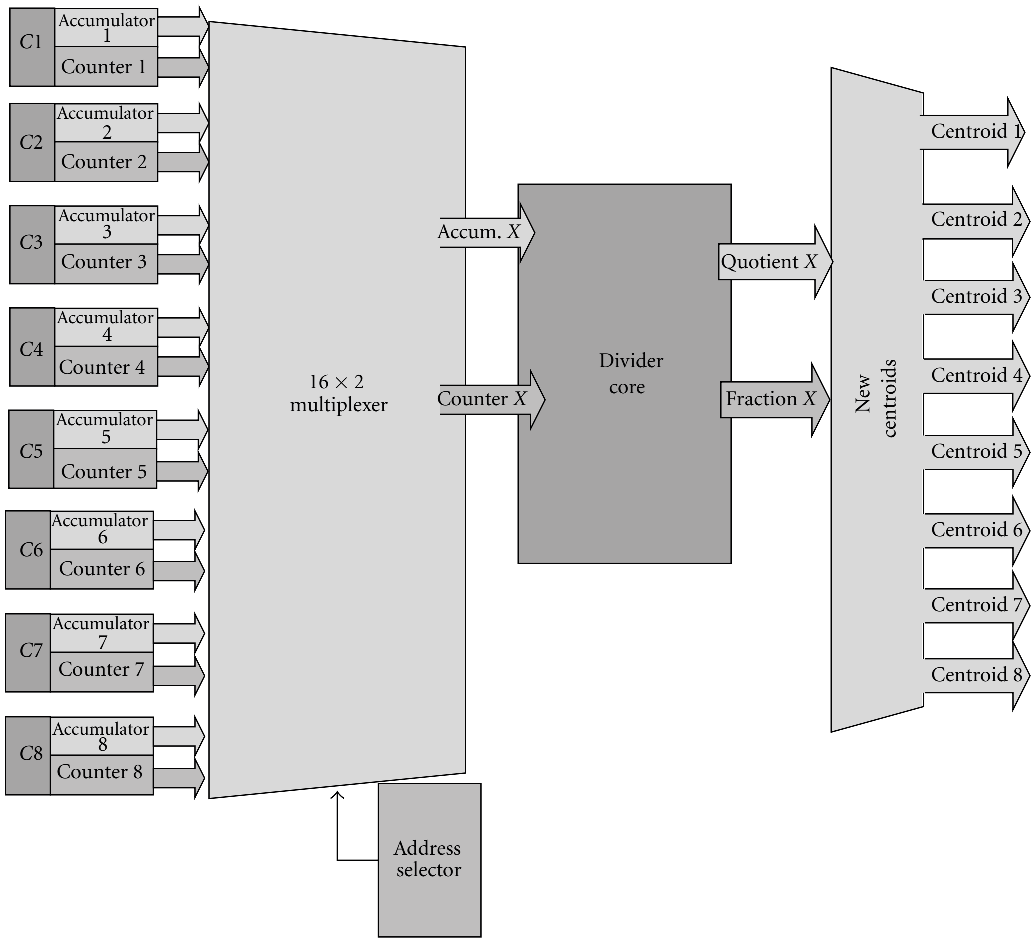


Figure 4: Sequential Divider Pipeline.

function of the divider latency and the number of clusters ($C$) as well as the number of dimensions ($M$) as shown in (10):

$$\text{divider time} = (\text{core latency}) + (C \times M), \quad (10)$$

where the core latency was 84 clock cycles based on the selected sizes of the dividend and divisor; however the remaining part will be subject to change as per the initial parameters entered by the user for $C$ and $M$.

The choice of using the pipelined divider from the core generator as opposed to a serial divider is that when comparing the performance of both in terms of area and timing, the serial divider was found to process one bit of the information at a time; thus, for a 32 bit dividend and 8 clusters the number of clock cycles needed was 256 as compared to 92 for the pipelined divider, based on single dimensional data only. This timing difference amplifies when the dimensions increase; for example, using 10 dimensions and 8 clusters causes the serial divider to take 2560 clock cycles while the pipelined divider takes only 164 clock cycles. On the other hand, the number of slices consumed by the serial divider is a lot less than the pipelined divider, where the latter consumed 1389 slices compared to 91 slices for the serial divider, when using Xilinx Virtex 4 FPGA. Since one of the project's aims was to accelerate the K-means algorithm, the pipelined divider was favored. However, other implementations based on multiple serial dividers are possible when FPGAs area is limited.

## 5. Novel Dynamic Partial Reconfiguration of the K-Means Clustering

The dynamic partial reconfiguration capability (DPR) of modern Xilinx FPGAs allows for better exploitation of FPGA resources over time and space. DPR allows for changing the device's configuration (i.e., functionality) partially and on the fly, leading to the possibility of fully autonomous FPGA-based systems. Therefore, DPR allows for the alteration of specific parts of the FPGA dynamically without affecting the configuration of other tasks placed onto the FPGA. This capability offers wider spectrum of applications for the K-means clustering serving different purposes which are explored here for the first time. For instance, DPR is useful in cases where users want to have the option to select specific distance metric when performing the K-means clustering or want to look at clustering results performed with different distance metrics since the choice of the distance metric affects the clustering performance. Another useful application of DPR with K-means is to use it for the implementation of server solution, where multiple K-means cores are configured on demand to work on different data as required by multiple users. A server solution is defined here as an application which allows a large FPGA to cater for the requirements of multiple users in a network whereby each user owns specific K-means core receiving streaming data and set with different parameters, for example, number of clusters. However, introducing changes to any of the K-means cores allocated onto the chip by a single user requires interrupting the operation of other K-means cores to reconfigure the FPGA. As such, the use of DPR to reconfigure specific K-means core within a multicore server solution system is investigated here to overcome the limitation of having to reconfigure the full chip and discover other advantages of DPR such as effect of the DPR on reconfiguration time. In addition, the multicore application of the K-means clustering can also be used in ensemble clustering. The latter is based on combing the results of repeated clustering using the same distance metric or alternative distances. Combing clustering results from repeated runs was found to improve the clustering accuracy when performed in GPP according to the work reported in [14]. However, running K-means several times is time consuming when done in GPP especially for large datasets. Consequently, implementing ensemble clustering on FPGA using multicore DPR approach will benefit from fast execution. In the following subsections, three different implementations of the K-means clustering algorithm based on DPR are presented.

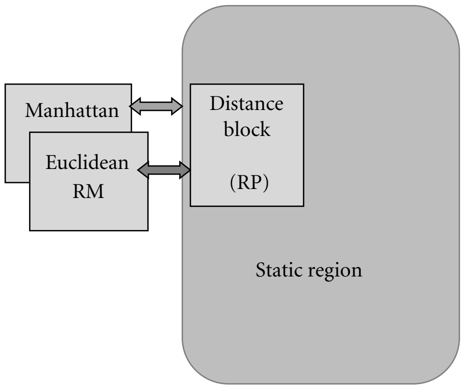


Figure 5: Illustrative diagram of the DPR implementation of the K-means core based on a reconfigurable distance kernel.

*5.1. DPR Based on Reconfigurable Distance Kernel.* To provide a solution to a user who wants to alter the distance metric kernel only without changing other blocks in the design or interrupting other running tasks on the FPGA, a DPR implementation of the K-means clustering based on setting the distance kernel as reconfigurable partition (RP) is proposed in this work. The RP can be configured with one of two possible Reconfigurable Modules (RMs), which correspond to the variations of the logic within the RP region. The two RMs correspond to the logic resources required to implement the distance kernel with either the Manhattan distance or the Euclidian as illustrated in Figure 5. More RMs could be created corresponding to other distance metrics such as the Hamming, Cosine, Canberra, Pearson, or Rank correlation coefficients; however, only the Manhattan and Euclidean distances are considered in this work due to their popularity with K-means clustering and to demonstrate a proof of concept.

The logic resources of the two distance metrics implemented in this work were found to be comparable in terms of the number of CLB slices and LUTs when synthesized, with the exception that the Euclidean metric required DSP48 blocks to implement the multiplication operation, which were not required for the Manhattan distance. However, this additional requirement was possible to cater for in this implementation and is not expected to impose serious shortage in resources as most modern FPGAs nowadays come with heterogeneous hardware resources that usually include DSP blocks. Furthermore, for the case when DSPs are not abundantly available or not available at all, the multiplication operation could still be performed using more of the CLB slices and LUTs, but this will increase the size of the RP region when compared with the case of using dedicated DSP blocks. The proposed DPR implementation is based on the case when the multiplication operation of the Euclidean distance metric is performed with DSP48 blocks.

To estimate the area requirement for the proposed DPR implementation, a non-DPR implementation based on using eight clusters, 13 bits wordlength ($B$), 2905 points ($N$), and single dimension ($M$) was first run using the two distance metrics. The place and route results showed that the CLB slices utilized for the Manhattan distance block were 277 as compared to 246 for the Euclidean distance both occupying only 4% of the total Xilinx XC4VFX12 floor area, with the Euclidean distance requiring eight DSP48 blocks only. The location of the RP region may be different depending on the FPGA used as different FPGAs have different number of DSP blocks and arrangements; consequently, one must make sure that enough of those blocks are included inside the RP region in order for the implementation to be successful. The above analysis of the area requirement for each of the two distances was necessary to evaluate the candidacy of this application for DPR implementation. In the case of the two areas being completely different, the DPR implementation would not have been feasible due to the significant loss of CLB slices within the RP regions causing the cost of the implementation to be considerable. In addition, if the RP is made so big, the partial reconfiguration time would be close to the full configuration losing the advantage of time saving when using DPR.

*5.2. DPR Implementation Based on Reconfigurable Single K-Means Core Using ICAP.* In this implementation, the K-means clustering core presented in Section 4 was modified to decrease the size of the core for simplicity; the modification was based on removing the divider since it was the kernel occupying the largest area while needed for short time during the clustering. This step is not expected to affect the remaining kernels, and future implementations would utilize larger FPGAs that can accommodate the complete core including the divider, or to perform the division operation using embedded processors. In the mean time, the new K-means implementation which excludes the divider is used and referred to as the K-means core.

This implementation is based on setting the aforementioned K-means core as a reconfigurable partition (RP) and reconfiguring it internally using Internal Configuration Access Port (ICAP). The latter has been used to access the FPGA configuration memory quickly with a bandwidth of 3.2 Gbps. To experimentally perform the DPR, an Internal Reconfiguration Engine (IRE) has been designed and tested by a colleague at the SLIg group in Edinburgh University.

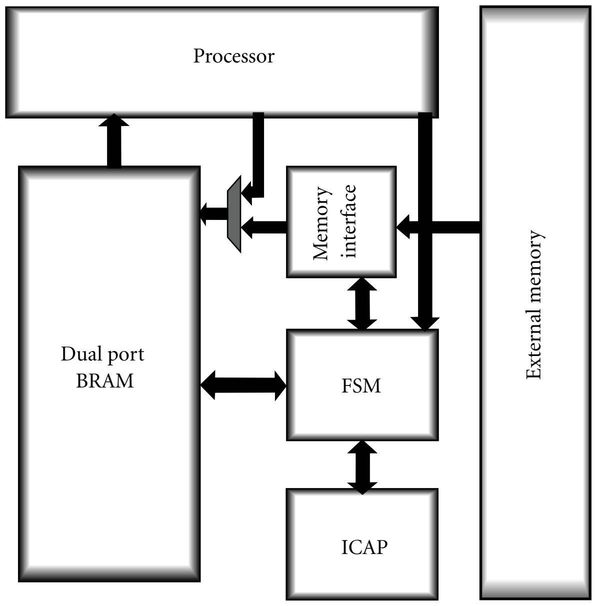


Figure 6: The architectural block diagram of the Internal Reconfiguration Engine (IRE) used to dynamically reconfigure the K-means core.

The block diagram of the IRE system used to dynamically reconfigure the K-means core is illustrated in Figure 6. The processor used in the IRE was based on Picoblaze soft-core processor and an external ZBT SRAM memory module.

Once the ICAP controller is enabled, the IRE starts the reconfiguration process of the K-means core using the partial bitstream. The processor passes the address of K-means partial bitstream in the external memory to the FSM which controls the ICAP signals and the flow of the partial bitstream data to the ICAP through a high speed dual port BRAM primitive used as a buffer. The main advantage of using this IRE is the fast reconfiguration time of the FPGA, which allows the variations of the K-means core to be implemented dynamically, those variations correspond to different internal memory contents, different parameters such as number of clusters or data wordlength. In addition, the IRE has an advantage of making the K-means core relocatable, which allows the system to maintain the operation of the K-means core when a fault occurs in the original location of the FPGA fabric hence providing small degree of fault tolerance; or when other processes need to be added to the FPGA hence providing a degree of flexibility in allocating the K-means core with respect to the newly introduced tasks. The latter is a particularly applicable in server solution applications.

*5.3. DPR Implementation Based on Reconfigurable Multiple K-Means Cores Using ICAP.* Based on the previous implementation, three K-means' cores were used to form a multicore DPR implementation, which can be reconfigured using the IRE. The IRE is capable of quickly reconfiguring each one of the cores at a time, and of relocating any of them within the same FPGA. In addition to the advantages mentioned in

Table 1: Performance results.

| GPP (ms) | Hardware (ms) | Speedup |
|---|---|---|
| 27.47 | 0.523 | ~53x |

the previous subsection, this implementation is particularly useful for server solution where cores are configured on demand upon request from multiple users. This feature allows the K-means cores allocated onto the FPGA to remain in operation while another core is being reconfigured elsewhere on the chip. Furthermore, DPR allows each user to exercise full control on the configuration of the owned task; thus, access to full chip reconfiguration can be granted to system administrator only to avoid frequent interruptions of operations. In this implementation, three cores were used only as proof of concept; additional cores could be added depending on the available resources and the application requirement.

## 6. Implementation Results

Each block in the design was implemented and tested on the Xilinx ML403 platform board, which houses the XC4VFX12 FPGA chip. The K-means design was captured in Verilog, simulated, synthesized, placed, and routed using Xilinx ISE 12.2 tool. Finally, a bitstream file was generated and downloaded to the board for testing. In the following subsections, performance results of our design, as well as comparison with GPP, FPGA, and GPU implementations are presented. Then results of the three DPR implementations of the K-means clustering will be presented.

*6.1. Comparison with GPP.* Implementation results when clustering a dataset of 2905 points and one dimension with input wordlength of 13 bits to 8 clusters showed that the complete design consumed 2985 slices (740 CLBs) and achieved a maximum clock frequency of 142.8 MHz. When comparing the runtime of this hardware implementation with an equivalent GPP implementation based on Matlab (R2008a) Statistical Toolbox running on 3 GHz Intel Core Duo E8400 GPP with 3 GB RAM, running Windows XP Professional operating system, the speedup results obtained are as shown in Table 1. Note that the Matlab Toolbox was chosen because it contains an optimized K-means function, and that it allows for converting data to fixed point easily before running the clustering using Matlab's Fixed-point Toolbox. The GPP time shown in Table 1 is the average time of 10,000 runs, with the initial clusters' centroids given as inputs to the algorithm; thus, the GPP implementation was made as close as possible to the FPGA implementation to ensure fair comparison. Note that the GPP implementation converged at 27 iterations while the hardware at 25.

*6.2. Comparison with Other FPGA Implementation.* In general, it is difficult to compare similar FPGA implementations because of the use of different FPGA families and chips, as well as different design parameters. Nonetheless, we have attempted a comparison here with the closest FPGA implementation to ours, namely, the one reported in [4]. Here, we compare our parameterized core design excluding the divider to make it compatible with the FPGA implementation reported in [4] which performed the division operation on a host. Both implementations were based on data size of 1024 × 1024 with 10 dimensions, 12 bits data, and 8 clusters. In both cases, data were stored offchip and streamed on to the FPGA. Comparative results are shown in Table 2. Obviously, our implementation is faster because it is based on a more recent FPGA technology, but it is also more compact using normalized slice/LUT count. More importantly, it is more flexible as it has a higher degree of parameterization compared to the implementation reported in [4].

Table 2: Implementation results.

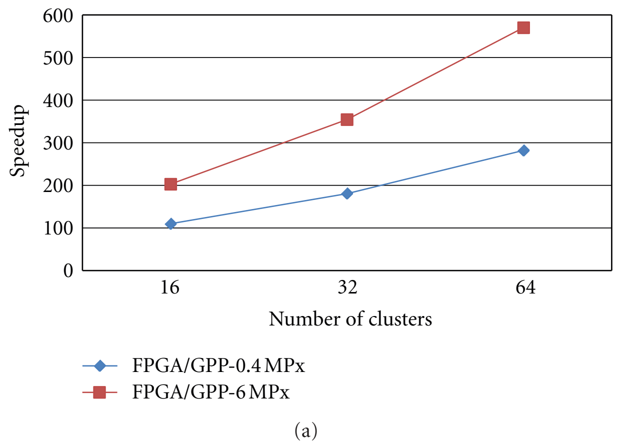
| Compare | Xilinx XCV1000 [4] | Xilinx XC4VFX12 |
|---|---|---|
| Slices | 8884/12288 | 5107/5549 |
| LUTs | 17768 | 10216 |
| Max. clock frequency | 63.07 MHz | 100 MHz |
| Single loop processing time | 0.17 s | ~0.07 s |

*6.3. Comparison with GPUs.* When comparing the performance of our FPGA K-means clustering implementation to a recent GPU implementation presented in [12] for an image processing application, the FPGA solution was found to outperform the GPU in terms of speed as shown in Table 3. The results shown were based on two different datasets, one was 0.4 Mega Pixel (MPx) in size and the other was 6.0 MPx. Both datasets were processed for 16, 32, and 64 clusters, and both were for a single dimension. The GPP and GPU results were based on 2.2 GHz Intel Core 2 Duo, with 4 GB memory and Nvidia GeForces 9600 M GT graphics card, running Microsoft Windows 7 Professional 64 bits. On the other hand, the targeted FPGA device was Xilinx XC4VSX35; the design used 13 bits to represent the dataset and could run at a maximum clock frequency of 141 MHz. The Virtex device used in this comparison is not a high end FPGAs, the latter can achieve higher speeds but we tried to limit the choice to a reasonable size that can accommodate the design and be reasonable for comparing with the above GPU. Both of images were too large to be stored within the FPGA, therefore, offchip memory was needed to store data which were streamed onto the FPGA pixel by pixel, and one data point was read every clock cycle. The processing times reported in [12] do not include the initialization of cluster's centers and the input/output stage, and similarly with the FPGA times reported in Table 3.

The speedup results of our FPGA implementation over the GPP and GPU results reported in [12] are shown in Figure 7. From the FPGA/GPP curve shown in Figure 7(a), it is clear that there is a linear relationship between the number of clusters and the attained acceleration. Similar observation was found when comparing GPU with GPP. Both observations confirm that FPGA and GPU outperform GPP as the number of clusters is increased. On the other

Table 3: Execution result of K-means in GPP, FPGA, and GPU, for single dimension data.

| Clusters | GPP avg. time per iteration (sec.) [12] | GPP avg. time for complete execution (sec.) [12] | GPU avg. time per iteration (sec.) [12] | GPU avg. time for complete execution (sec.) [12] | FPGA per iteration (sec.) | FPGA complete execution (sec.) |
|---|---|---|---|---|---|---|
| 0.4 MPx | | | | | | |
| 16 | 0.269 | 4.314 | 0.021 | 0.443 | 0.0028 | 0.0392 |
| 32 | 0.516 | 7.637 | 0.020 | 0.421 | 0.0028 | 0.042 |
| 64 | 1.004 | 12.78 | 0.023 | 0.508 | 0.0028 | 0.0454 |
| 6 MPx | | | | | | |
| 16 | 4.279 | 67.07 | 0.256 | 5.176 | 0.0425 | 0.723 |
| 32 | 8.144 | 110.7 | 0.247 | 4.439 | 0.0425 | 0.638 |
| 64 | 15.86 | 208.2 | 0.270 | 5.220 | 0.0425 | 0.723 |



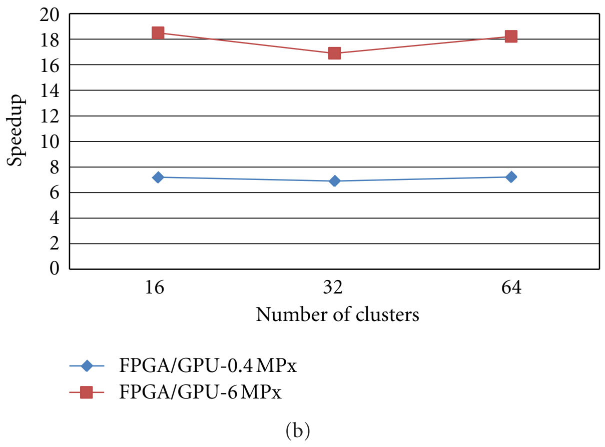


Figure 7: Speedup results of the FPGA implementation of K-means over both: (a) GPP and (b) GPU implementations.

hand, the FPGA/GPU curve shown in Figure 7(b) indicates that FPGA outperforms GPU in terms of execution time; this is due to higher exploitation of parallelism in FPGA. On the other hand, the FPGA/GPU acceleration is not greatly affected by the number of clusters (up to 64 clusters in our experiments) as found with GPP. As for the device utilization, the XC4VSX35 FPGA used in this comparison has 15,360 slices, which were enough to implement the logic required to accommodate the number of clusters shown in Table 3. With the 16 clusters, the implementation occupied 5,177 slices (33%), and with 32 and 64 clusters, the implementation occupied 8,055 slices (52%) and 13,859 (98%), respectively.

In addition, the effect of data dimensionality on performance of GPP, FPGA, and GPU implementations was investigated in this work based on GPP and GPU performance results reported in [15]. In [15], the authors reported the results of clustering Microarray Yeast expression profiles as shown in Table 4 where GPU achieved speedup of 7x to 8x over GPP for four and nine dimensions, respectively, while FPGA achieved 15x to 31x for the same dimensions based on dataset of 65,500 vectors, as illustrated in Figure 8 for the case of three and four clusters. When comparing the timing performance of single iteration of the K-means clustering for GPP, GPU, and FPGA implementations based on multidimensional data, GPU and FPGA were found to outperform GPP as shown in Figure 9(a). When the performance of our FPGA implementation was specifically compared with the GPU implementation in [15], the FPGA achieved speedup between 2x to 7x over GPU [16]. Note that the results reported in Table 4, Figures 8 and 9 are all based on XC4VSX35 FPGA and Nvidia 8600 GT GPU. In addition, FPGA outperformed GPU when the dimensions of data increased (only four and nine dimensions were studied) as shown in Figure 9(b), which indicates that FPGA maintained its performance as dimensions were increased while GPU experienced a drop in performance. The drop in performance in GPU as the dimensions increased is due to the way the implementation utilizes resources within the GPU when computing specific kernels, particularly with regards to memory bottlenecks associated with GPUs as data increase in size.

Furthermore, Figure 8 also highlights the performance of five-core implementation for three and four clusters with respect to GPU, which was first reported in [2]. The five-core implementation is clearly superior to the GPUs for problems requiring small or reasonable number of clusters and dimensions that can be mapped easily onto commercially available FPGA devices [16]. This illustrates the high potentials of FPGA in parallelizing tasks.

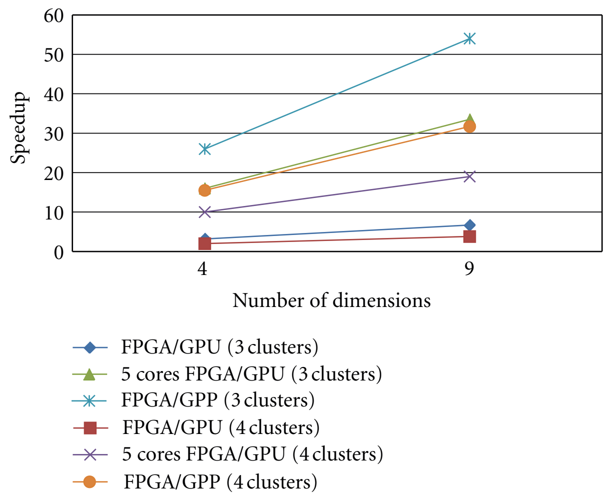


Figure 8: Effect of data dimensionality on the speedup of the FPGA implementation of the K-means over GPP and GPU. In addition, the performance of five-core FPGA implementation reported in [2, 16] is compared with the GPU implementation of [15].

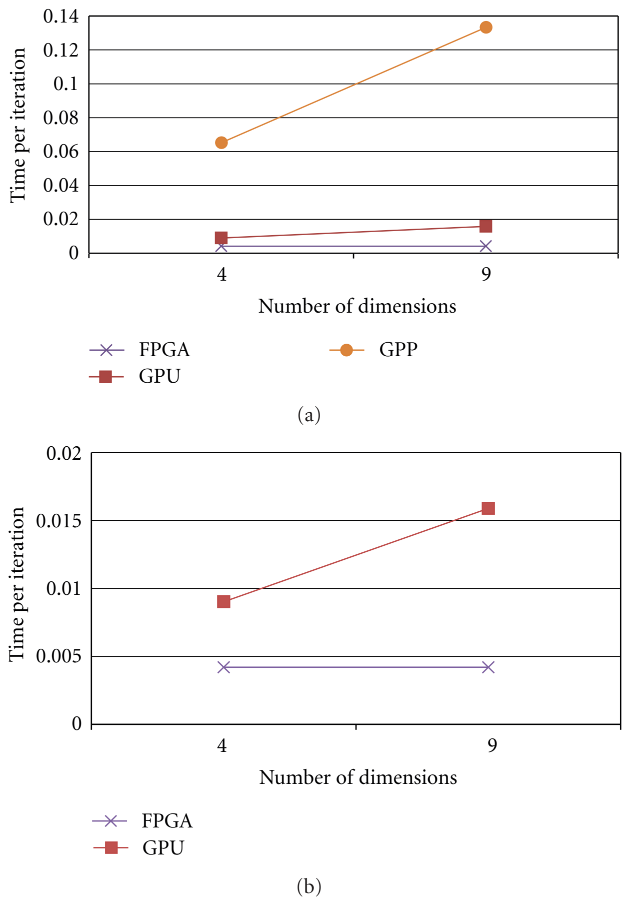


Figure 9: Effect of data dimensionality on the timing per iteration of the K-means clustering for (a) GPP, GPU and FPGA, and (b) GPU and FPGA only. All are based on four clusters; the GPP and GPU results are based on [15] whereby FPGA results are based on [16].

Table 4: Execution result of K-means in GPP, FPGA, GPU, for multidimension data.

| | GPP time per iteration (sec.) [15] | GPU time per iteration (sec.) [15] | FPGA time per iteration (sec.) |
|---|---|---|---|
| Dimensions = 4 | | | |
| 3 clusters | 0.0495 | 0.00623 | 0.0019 |
| 4 clusters | 0.0652 | 0.00902 | 0.0042 |
| Dimensions = 9 | | | |
| 3 clusters | 0.1031 | 0.0125 | 0.0019 |
| 4 clusters | 0.1333 | 0.01589 | 0.0042 |

Table 5: Comparison of power and energy of different K-means implementations.

| Platform | Power (watt) | Execution time for the 0.4 MPx image, with 16 clusters (sec.) | Energy (joule) |
|---|---|---|---|
| GPP | 120 | 4.314 | 517 |
| GPU | 59 | 0.443 | 26 |
| FPGA | 15 | 0.056 | 0.84 |

*6.4. Comparison in Power and Energy Consumption.* The power and energy consumption of the three K-means implementations were compared and reported in Table 5 based on the 0.4 Mpx image shown in Table 3. Both GPP and FPGA power figures were actually measured, while the GPU power was obtained from the Nvidia GeForce 9600 GT datasheet, reflecting the power rating of the device [17]. The results in Table 5 are based on using 13 bits to represent the 0.4 MPx image, 16 clusters and targeting the XC4VFX12 FPGA available on the ML 403 board with the image being stored in offchip memory. The FPGA is ~8x more power efficient than GPP and ~4x more power efficient than GPU. Consequently, the FPGA implementation is ~615x more energy efficient than the GPP and ~31x more energy efficient than the GPU as obtained from (11). Note that the FPGA implementation utilized 4909 slices (89%) of the targeted device as reported in [16].

In addition, when comparing the power consumption of the FPGA implementation of the single core K-means shown in Section 6.1 with the GPP implementation, the FPGA consumed 15 W only when actually measured as compared to 90 W in GPP; thus, the FPGA is six times more power efficient than the GPP while being 53 times faster. This has resulted in the FPGA implementation being 318 times more energy efficient than the equivalent GPP implementation as computed from (11). Note that the GPP power was measured while running the algorithm in Matlab in a loop, and GPP power was 70 W when idle

$$\text{energy efficiency} = \text{power efficiency} \times \text{speedup}. \quad (11)$$

*6.5. Comparison in Cost of the K-Means Implementation.* The cost of any computing platform depends on several factors, that is, purchasing cost, development cost, operating cost, and perhaps maintenance/upgrading cost. At first, purchasing any technology is associated with its technical

Table 6: Purchase cost of the three computing platforms with/without host.

| Platform | Purchasing cost W/O host (£) | Purchasing cost with host (£) | Normalised cost with host (£) |
|---|---|---|---|
| GPP | 747.55 | 747.55 | 1 |
| FPGA | 511 | 1260 | 1.69 |
| GPU | 127 | 874 | 1.17 |

specifications and available resources, for example, logic resources, peripherals, hardened IP blocks, and external memory. Computing platforms are offered within a wide range of device families allowing a user to select the device having the right combination of resources for the application in hand. Table 6 reports the cost of the computing platforms used in the K-means clustering implementations which were reported in Table 5 based on using the following computing platforms:

(i) GPP-3.0 GHz Intel Core 2 Due E8400 processor and 3 GB memory;

(ii) GPU-Nvidia GeForce 9600 M GT;

(iii) FPGA-Xilinx ML403 board.

Table 6 reveals that the GPP solution was the most cost effective in terms of purchasing cost when compared with FPGA and GPU followed by the GPU solution and last by the FPGA solution. Although Table 6 implies that GPP is the most cost effective in terms of purchasing cost, followed by GPU and last by FPGA, this does not mean that GPP or GPU is more economic solutions than FPGA given that performance per dollar and performance per watt are more realistic measures for the economic viability of the technology [18]. Although it was not feasible to measure all factors leading to performance per dollar and performance per watt, the power and energy consumptions shown in Table 5 leads to estimate FPGA to have the largest performance per watt. Consequently, FPGA is the most economically viable solution.

*6.6. DPR Implementation Based on Reconfigurable Distance Kernel.* The DPR implementation was created using Xilinx PlanAhead 12.2 tool following Xilinx hierarchical methodology and partial reconfiguration flow [19, 20], the distance block was set as RP as explained earlier. The actual DPR implementation shows that Euclidean distance occupied 238 slices within the RP region as opposed to 208 for the Manhattan distance; the resources were slightly different from those obtained using normal flow implementation.

Two main configurations were generated based on the two RMs and the associated full and partial bitstreams for each configuration were also generated. The full bitstream was 582 KB in size while the partial bitstream was 61 KB. In this work, JTAG cable was used to configure the FPGA, which has a bandwidth (BW) of 66 Mbps. Therefore, when fully configuring the FPGA, the configuration time was 70.55 ms as computed from (12). On the other hand, when the FPGA was partially reconfigured, the configuration time was

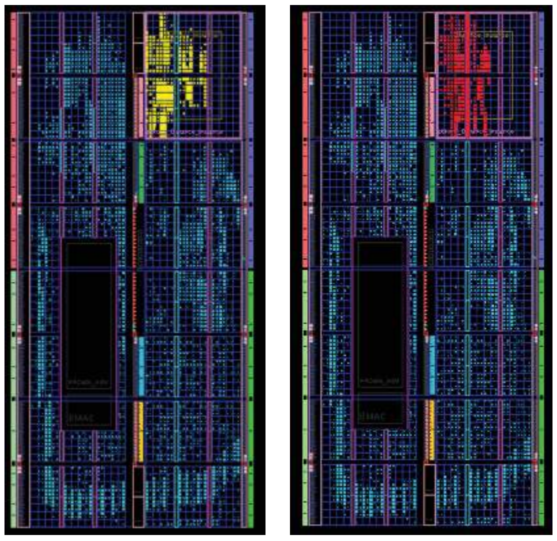

(a) Manhattan (b) Euclidean

Figure 10: The floorplan image of the DPR implementation of the K-means clustering based on reconfigurable distance metric: (a) Manhattan, and (b) Euclidean.

7.39 ms. Therefore, DPR offers significant time saving when needing to change the distance metric only leading to 10x speedup in configuration time over full device configuration. Therefore, in addition to being able to reconfigure specific part of the device without interrupting the operation of other tasks, DPR offers time saving

$$\text{configuration time} = \frac{\text{size of bitstream}}{\text{BW of configuration mode}}. \quad (12)$$

Using Internal Configuration Access Port (ICAP) as a configuration mode will offer larger bandwidth than JTAG, since ICAP have a bandwidth equivalent to 3.2 Gbps leading to small configuration time in the range of microseconds as compared to milliseconds when using JTAG. However, the speedup ratios between full and partial reconfigurations remain the same for the two configuration modes. The validity of such estimation as opposed to actual measurement will be investigated in the following subsection. Figure 10 illustrates the floorplan of the two implementations highlighting the area occupied by the RP. The image also highlights a disadvantage of the implementation which is associated with wasting some of the CLB slices within the RP region due to having to enclose enough DSP48 blocks; this issue is clearly device specific as the arrangement of DSP48 blocks affects the size of the RP region. The image shows that ~72% of the RP CLB slices are unused in the Manhattan distance case and ~69.5% for the case of the Euclidean distance.

*6.7. DPR Implementation Based on Single K-Means Core and ICAP.* The design containing both the K-means core and the Internal Reconfiguration Engine (IRE) was first synthesized using Xilinx ISE 12.2 to generate the .ngc file required for creating the DPR implementation in Xilinx PlanAhead

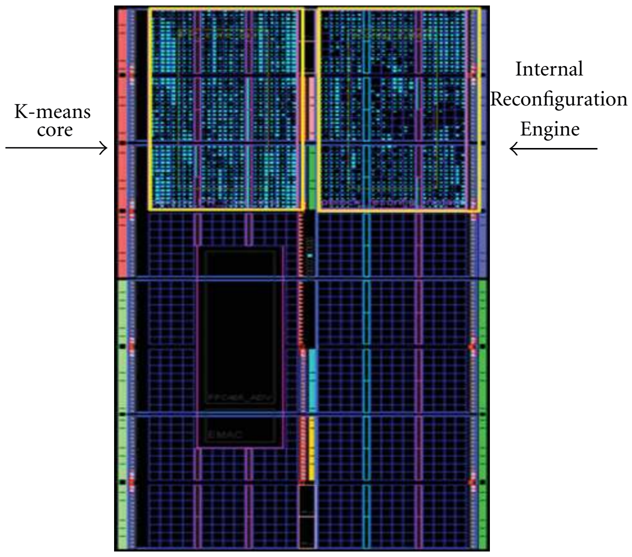


Figure 11: The floorplan image of the DPR implementation of the reconfigurable K-means core based on Internal Reconfiguration Engine (IRE).

12.2; in addition another .ngc file corresponding to the K-means core without I/O buffers was created to describe the Reconfigurable Module (RM), which corresponds to the K-means core (excluding the divider). Next, PlanAhead was used to construct the DPR implementation based on Xilinx ML 403 platform board and on setting the K-means core as a reconfigurable partition (RP).

The design was implemented, placed, and routed targeting Xilinx XC4VFX12 FPGA available in the ML403 board. The place and route results of the implementation showed that the K-means core occupied 1,178 CLB slices (∼22%) of the FPGA floor area while the IRE occupied 955 CLB slices (∼18%); the design was constrained to work at a 100 MHz clock speed. The implementation was run and verified, and bitstreams were created. Figure 11 illustrates the location and size of the implementation within the FPGA. The partial bitstream of the K-means core was found to be 140 KB while the full bitstream was 582 KB.

To test the DPR implementation, the partial bitstream was written to the ZBT SRAM available on board the ML403; note that the file was stored initially in a Compact Flash (CF) card which was also available on board the ML403, and that Microblaze was used to read the partial bitstream from the CF and write it to the ZBT SRAM. Second, the FPGA was fully configured using the full bitstream associated with our DPR implementation, this has invoked the K-means core and the IRE. The device by then was running the K-means clustering and was ready to be partially reconfigured upon enabling the ICAP controller in the IRE. A counter was used to measure the configuration time, which corresponds to the time in which the enable signal was asserted in the ICAP controller. Upon enabling the IRE, the partial bitstream was read from the ZBT SRAM and written to the configuration memory of the FPGA; as soon as the partial reconfiguration was finished, the IRE deasserted the enable signal, and time was measured. The measured partial reconfiguration time was 360 $\mu$s, as compared to ∼1455 $\mu$s for the full configuration. This shows that the DPR implementation of the single core K-means is ∼4x faster than full chip configuration.

In Section 6.6, the partial configuration time was estimated from (12) based on using JTAG, while the partial configuration time in this section was actually measured. To justify the validity of the results obtained in the previous subsection, the partial configuration time of the K-mean core presented in this section was obtained using (12) based on ICAP and checked against actual measurement to see how significant the overheads are. The partial reconfiguration time was found to be 350 $\mu$s when applying (12), which is different from the measured time by only 10 $\mu$s leading to same speedup in reconfiguration time (∼4x). This small difference could be attributed to the time needed to read the partial bitstream from the ZBT SRAM, time overheads result from delays in asserting or deasserting the enable signal. Therefore, estimating the configuration time gives relatively similar results to actual measurements, and one could rely on estimated results in predicting the performance of the implementation. Furthermore, the fact that the performance of the IRE was very high has led to small overhead times.

The IRE is also capable of relocating the K-means core to another location in the chip by modifying the partial bitstream to reflect the new desired location given that the new location is compatible with the original core in terms of resources, that is, Block RAMs, CLB slices, and so forth. However, due to limited resources in the available device, relocatability could not be tested on the FPGA. On the other hand, other smaller designs were successfully relocated within the FPGA using the same IRE to validate its capability in re-allocating tasks [21].

*6.8. DPR Implementation Based on Multiple K-Means Cores and ICAP.* The three-core DPR implementation was implemented with Xilinx XCVLX60 FPGA which has 26,624 CLB slices; thus, it was able to accommodate the three K-means' cores and the IRE as illustrated in Figure 12. The performance of this implementation was estimated based on the performance of the single core implementation, due to the unavailability of a large FPGA to actually test the implementation. The IRE system allows for one of the three cores to be partially configured at a time; thus, the partial reconfiguration time for each K-means core is the same as that of the single core being 360 $\mu$s as compared to 5407.5 $\mu$s for the full configuration. The former was measured in Section 6.6, while the latter was estimated from (12) based on a full bitstream of 2,163 KB in size and ICAP bandwidth of 3.2 Gbps. Consequently, DPR is 15x faster in reconfiguration time than full configuration when one of the K-means' cores need to be reconfigured. As for the case when the whole three cores need to be reconfigured, 1080 $\mu$s would be required reducing the reconfiguration speedup of this DPR implementation to 5x only. This is mainly because the ICAP has to reconfigure each core sequentially.

An important observation was made regarding the speedup in reconfiguration time relative to the size of the

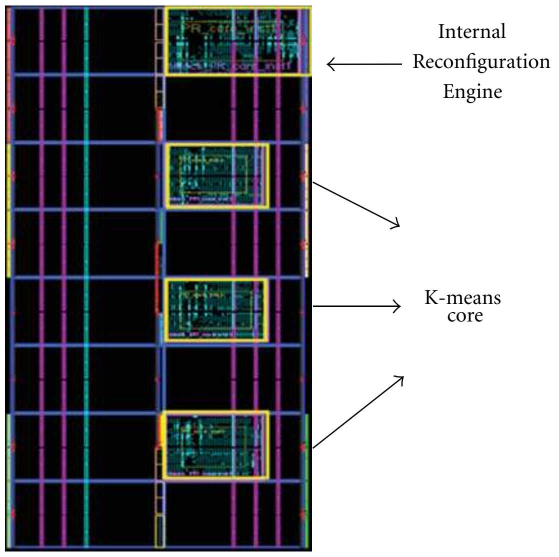


Figure 12: The floorplan image of the DPR implementation of the three K-means' cores based on Internal Reconfiguration Engine (IRE).

FPGA device. It was found that when the FPGA was large, the full bitstream would normally be large leading to longer configuration time, while the partial bitstream would be the same for the single K-means core in both the small and large FPGAs. Therefore, when the size of the FPGA is large, the speedup in configuration time is expected to be higher than that of a small device given that the size of the RP in the two devices is identical. This is particularly important for server solution as such an application is usually expected to utilize large FPGA. For instance, when XC4VFX12 FPGA was used, the speedup in reconfiguration time was 4x as compared to 15x for the XCVLX60, both based on the same PR implementation.

## 7. Summary and Discussion

The design and implementation of a highly parameterized FPGA core of K-means clustering were presented in this paper. This outperformed equivalent GPP and GPU implementations in terms of speed (two orders and one order of magnitude, resp.). In addition, the FPGA implementation was more energy efficient than both GPP (615x) and GPU (31x). This makes the FPGA implementation highly desirable although FPGAs still suffer from a relatively higher cost of purchase and development compared to GPPs and GPUs. The lack of standard API tools and hardware boards is a big contributor in this.

In addition, the performance of both FPGA and GPU was found to be superior to GPP when increasing the number of clusters. This is mainly attributed to the fact that FPGA and GPU scale better with the number of clusters, whereas GPPs do this computation sequentially. On the other hand, when comparing the timing performance of FPGA and GPU, the FPGA excelled the GPU's performance for the particular devices used in the comparison reported in Table 3. This is attributed to the higher level of parallelism exploited in FPGA than in the GPU.

As for the dimensionality effect, when the number of clusters were increased for different data dimensions, the speedup of FPGA as compared to GPU was almost constant as shown in Figure 7(b), which emphasizes the fact stated earlier that the two technologies scale well as the number of clusters was increased. However, FPGA outperformed the GPU implementation when the dimensions ($M$) of the data were increased as reported in Table 4; this is attributed to memory bottlenecks in GPUs when the size of the data is large.

However, the above findings are device specific and could not be generalized unless fair comparison is made using higher end GPUs and FPGAs. Even then, the large variation in the size of FPGAs within the same family range makes it difficult to assess the performance of the two technologies, especially that variations in GPUs within the same family range are much smaller. Furthermore, other issues arise when high end devices are used such as the cost of purchasing the high end device, with FPGAs being more expensive than GPUs and power consumption issue where GPUs and GPPs consume more power than FPGAs [18]. Nevertheless, our comparative study was an attempt to highlight main performance bottlenecks such as memory limitations in GPUs, and issues surrounding the implementation of K-means in GPPs, GPUs, and FPGAs when number of clusters or data dimensions change, and to provide some guidelines for appropriate device selection based on the application in hand.

Another aim of the work was to investigate the benefits from using dynamic partial reconfiguration to set a specific kernel within the K-means algorithm as reconfigurable partition (RP) to serve specific applications. When setting the distance kernel as RP, the reconfiguration time was found to be 10x faster than reconfiguring the full device. The purpose of such implementation was to be able to change the type of the distance metric at run time without interrupting the operation of tasks in other parts of the FPGA and to achieve this as quickly as possible. Two distance metrics were considered in this work to demonstrate the feasibility of the concept, one was the Manhattan distance and the other was the Euclidian; however, the concept could be easily expanded to include a library of distance metrics if the application in hand requires such variability in distance metrics.

Furthermore, the results of reconfiguring a single K-means core and three K-means cores using an Internal Reconfiguration Engine (IRE) were presented; the latter was based on using ICAP to dynamically reconfigure the FPGA. The IRE illustrated true dynamic reconfiguration capability at high performance with negligible overheads. In terms of reconfiguration speedup, the single core implementation was four times fast when using small chip and 15 times fast when large chip was used. As for the three-core implementation, the speedup in reconfiguration time varies according to the number of cores to be configured at a time, for partially reconfiguring single core, two cores, and three cores the speedups were 15x, ~8x, and 5x, respectively. Consequently, it can be stated that multicore DPR has best performance in terms of configuration time when used to reconfigure one core at a time due to having short partial reconfiguration time, and this advantage becomes less or even lost as more cores get reconfigured simultaneously leading to the same performance in configuration time as non-DPR implementation (normal flow).

## 8. Conclusion and Future Work

A highly adaptive FPGA implementation of K-means clustering has been presented in this work which outperformed GPP and GPU in terms of speed and energy efficiency as well as scalability with increased number of clusters and data dimensions. Furthermore, the FPGA implementation was the most economically viable solution. Additionally, three novel DPR implementations of the K-means clustering were presented which allowed for dynamic partial reconfiguration of FPGA offering the advantage of reconfiguration flexibility, short partial reconfiguration time of selective tasks on chip while ensuring continuous operation of other tasks.

Future work will be centered on improving the division operation in the K-means clustering with the aim to reduce sizing and making the divider shared among multi K-means' cores by harnessing the dynamic partial reconfiguration capability to time-multiplex the divider. Additionally, considerations will be given to harnessing embedded processors to implement the division operation in conjunction with dynamic partial reconfiguration. Furthermore, applying K-means ensemble clustering in FPGA based on multicore dynamic partial reconfiguration to target Microarray data that will be implemented. Moreover, comparison with optimized multicore processor will be carried out.

## Acknowledgments

The authors would like to thank the "Public Authority of Applied Education and Training" in Kuwait for sponsoring this study and to also thank "Kuwait Foundation for the Advancement in Sciences" for its contribution to this study.

## References

[1] P. Kumar, B. Ozisikyilmaz, W.-K. Liao, G. Memik, and A. Choudhary, "High performance data mining using R on heterogeneous platforms," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, Workshops and Phd Forum (IPDPSW '11)*, pp. 1720–1729, 2011.

[2] 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 *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '11)*, pp. 248–255, 2011.

[3] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski, "Algorithmic transformations in the implementation of K-means clustering on reconfigurable hardware," in *Proceedings of the ACM/SIGDA 9th International Sysmposium on Field Programmable Gate Arrays (FPGA '01)*, pp. 103–110, February 2001.

[4] M. Leeser, P. Belanovic, M. Estlick, M. Gokhale, J. J. Szymanski, and J. Theiler, "Applying reconfigurable hardware to the analysis of multispectral and hyperspectral imagery," in *Imaging Spectrometry VII*, vol. 4480 of *Proceedings of SPIE*, pp. 100–107, August 2001.

[5] M. D. Estlick, *An FPGA implementation of the K-means algorithm for image processing [M.S. thesis]*, Department of Electrical and Computer Engineering, Northeastern University, Boston, Mass, USA, 2002.

[6] D. Lavenier, *FPGA Implementation of the K-Means Clustering Algorithm For Hyperspectral Images*, Los Alamos National Laboratory, LAUR, Los Alamos, Ill, USA, 2000.

[7] M. Gokhale, J. Frigo, K. Mccabe, J. Theiler, C. Wolinski, and D. Lavenier, "Experience with a hybrid processor: K-means clustering," *Journal of Supercomputing*, vol. 26, no. 2, pp. 131–148, 2003.

[8] J. Theiler, M. Leeser, M. Estlick, and J. J. Szymanski, "Design issues for hardware implementation of an algorithm for segmenting hyperspectral imagery," in *Imaging Spectrometry VI*, vol. 4132 of *Proceedings of SPIE*, pp. 99–106, August 2000.

[9] V. Bhaskaran, *Parametrized implementation of K-means clustering on reconfigurable systems [M.S. thesis]*, Department of Electrical Engineering, University of Tennessee, Knoxville, Ten, USA, 2003.

[10] R. Farivar, D. Rebolledo, E. Chan, and R. Campbell, "A parallel implementation of K-means clustering on GPUs," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '08)*, pp. 340–345, Las Vegas, Nev, USA, July 2008.

[11] S. A. A. Shalom, M. Dash, and M. Tue, "Efficient K-means clustering using accelerated graphics processors," in *Proceedings of the 10th International Conference on Data Warehousing and Knowledge Discovery (DaWaK '08)*, vol. 5182 of *Lecture Notes in Computer Science*, pp. 166–175, 2008.

[12] G. Karch, *GPU based acceleration of selected clustering techniques [M.S. thesis]*, Department of Electrical and Computer Engineering and Computer Sciences, Silesian University of Technology in Gliwice, Silesia, Poland, 2010.

[13] A. Choudhary, D. Honbo, P. Kumar, B. Ozisikyilmaz, S. Misra, and G. Memik, "Accelerating Data Mining Workloads: current approaches and future challenges in system architecture design," *Wiley Interdisciplinary Reviews*, vol. 1, pp. 41–54, 2011.

[14] M. C. P. De Souto, S. C. M. Silva, V. G. Bittencourt, and D. S. A. De Araujo, "Cluster ensemble for gene expression microarray data," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '05)*, vol. 1, pp. 487–492, August 2005.

[15] S. A. Shalom, M. Dash, and M. Tue, "GPU-based fast k-means clustering of gene expression profiles," in *Proceedings of the 12th Annual International Conference on Research in Computational Molecular Biology (RECOMB '08)*, Singaphore, 2008.

[16] H. Hussain, K. Benkrid, H. Seker, and A. Erdogan, "Highly parametrized K-means clustering on FPGAs: comparative results with GPPs and GPUs," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '11)*, pp. 475–480, 2011.

[17] Nvidia Corp., GEForce 9600 GT datasheet, 2012, http://www.nvidia.com/object/product_geforce_9600gt_us.html.

[18] K. Benkrid, A. Akoglu, C. Ling, Y. Song, X. Tian, and Y. Lue, "High perfomance biological pairwise sequence alignment: FPGA vs. GPU vs. CellBE vs. GPP," *International Journal of Reconfigurable Computing*, vol. 2012, Article ID 752910, 15 pages, 2012.

[19] Xilinx Corp., Hierarchical Design Methodology guide, ug748, v13.3, 2011, http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/Hierarchical_Design_Methodology_Guide.pdf.

[20] Xilinx Corp., Partial Reconfiguration guide, ug702, v12.3, p. 103, 2010, http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/ug702.pdf.

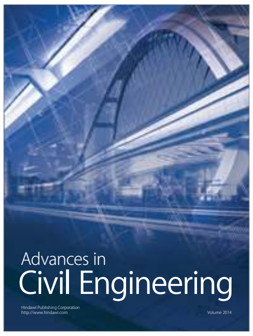
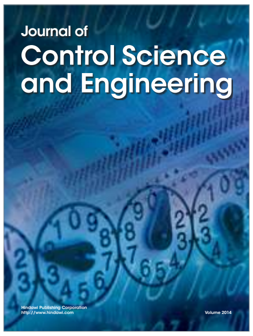
[21] X. Iturbe, K. Benkrid, T. Arslan, C. Hong, and I. Martinez, "Empty resource compaction algorithms for real-time hardware tasks placement on partially reconfigurable FPGAs subject to fault ocurrence," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '11)*, pp. 475–480, November 2011.

Journal of
Engineering

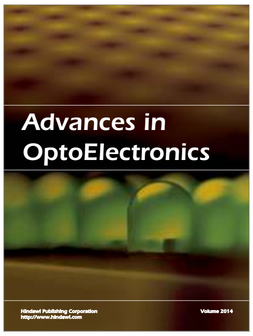
The Scientific World Journal

International Journal of
Rotating Machinery

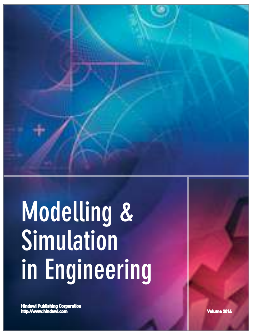
Journal of
Sensors

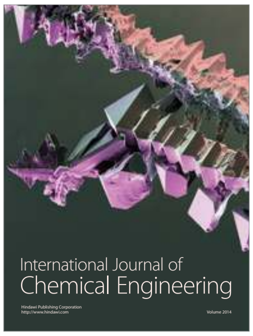
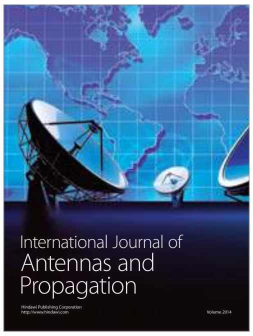
International Journal of
Distributed Sensor Networks

Advances in
Civil Engineering

Journal of
Control Science and Engineering

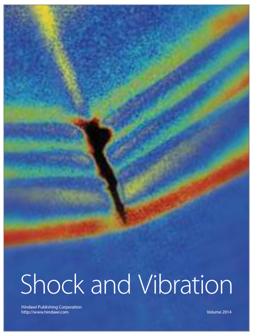
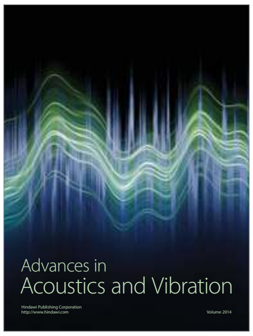
Journal of
Robotics

Journal of
Electrical and Computer Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and Observation

Modelling & Simulation in Engineering

International Journal of
Aerospace Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and Propagation

Active and Passive Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration