FPGA Implementation of K-means Algorithm for Bioinformatics Application: An Accelerated Approach to Clustering Microarray Data

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

[1]System Level Integration Group, School of Engineering, The University of Edinburgh
King's Buildings, Mayfield Road, Edinburgh, EH9 3JL, UK
{h.hussain, k.benkrid, Ahmet.Erdogan} @ed.ac.uk
[2]Bio-Health Informatics Research Group, Center for Computational Intelligence
De Montfort University, Leicester, LE1 9BH, UK
hseker@dmu.ac.uk

Abstract

The Microarray is a technique used by biologists to perform many genome experiments simultaneously, which produces very large datasets. Analysis of these datasets is a challenge for scientists especially as the number of genome databases is increasing rapidly every year. K-means clustering is an unsupervised data mining technique used widely by bioinformaticians to analyze Microarray data. However, K-means can take between a few seconds to several days to process Microarray data depending on the size of these datasets. This puts a limit on the complexity of biological problems which can be asked by bioinfomaticians, and hence may result in an incomplete solution to the problem. In order to overcome such problems, we propose a highly parallel hardware design to accelerate the K-means clustering of Microarray data by implementing the K-means algorithm in Field Programmable Gate Arrays (FPGA). Our implementation is particularly suitable for server solution as it allows for processing many different datasets simultaneously. We have designed, and implemented five k-mean cores on Xilinx Virtex4 XC4VLX25 FPGA, and tested them on a sample of real Yeast Microarray data. Our design achieved about 51.7x speed-up when compared to a software model while being 206.8x more energy efficient.

1. Introduction

Microarray is a technique used in genome experiments to measure expression level of many thousands of genes simultaneously. Today, Microarrays have become one of the key techniques for biologists who want to study genomes and uncover the regulating mechanisms of genes by monitoring their expressions [1]. It is also known as *mini laboratory* since it can be used to run thousands of experiments at the same time. Microarray has been around for fifteen years only, and more research is anticipated as a result of the rapid growth in genome databases resulting from the human genome project, which identified 27,628 genes in human [2].

Gene expression profiles help scientists discover functions and structures of unknown genes as well as understand how genes are regulated. For example, if Microarray is used to test samples at rest and others during cellular division, the expression profiles of the samples can give scientists an indication to which genes were active during a particular division stage and thus infer gene role, function, or involvement in disease formation. Another example is the use of expression profiles of cancer cells and normal cells, which can help in diagnosing different forms of cancers and making an appropriate decision regarding the course of treatment; this can also be used to study the effect of a specific treatment such as chemotherapy drug on the cancerous cells [1]-[3]. The number of Microarray based studies to identify genes involved in the above mentioned studies is growing exponentially, and Microarray data mining is expected to play a role in advancing the technology toward more clinical use as both diagnostic and prognostic tool [3].

Microarray experiments return results in the form of an image as shown in Figure 1 below, which is then quantized into an intensity matrix and stored in a 2D matrix, where rows specify the features which are usually genes and the columns specify the number of samples or the varying experimental conditions such as time points, a person, a cell line, a mouse, or any other living organism. The data file usually contains the experimental data such as gene profiles, sample information, gene annotation, and experiment description [2]-[5]. To process the huge Microarray data, scientists use data mining methods such as K-means clustering, hierarchical clustering, or self organizing maps, to separate data into smaller groups and extract meaningful information from them. This process is expected to discover a group of genes that have similar biological functions and behaviour and help reduce the number of genes for further experimental analysis (e.g., drug/vaccine development). Partitioning data into clusters is one of the useful techniques in bioinformatics applications where specific number of clusters is usually desired. However, when a datasets is so large, the number of clusters becomes a drawback in data partition due to difficulty in determining the number of clusters beforehand [1].

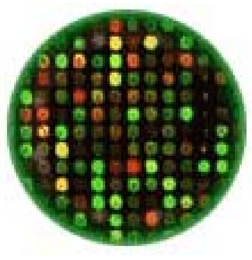

Figure 1. Scanned Microarray Image [4]

In the work presented in this paper, we choose to implement the K-means clustering in hardware due to its popularity in processing Microarray data on one hand, and to the parallelism that can be exploited from the algorithm on the other hand, which makes it a good candidate for hardware acceleration. When clustering Microarray data using K-means, we expect the genes grouped into one cluster to share some degree of similarity in terms of expression characteristics, and thus have similar roles or functions. The work presented in this paper is the first of its kind to our knowledge to apply K-means clustering in hardware for Microarray data and to fully use resources within the Virtex4 FPGA without need of using any off chip resources.

2. Clustering

Clustering in general aims to group data which share some degree of similarity by forming subgroups. When clustering gene expression profiles, subgroups of co-expressed genes are believed to be for genes that are participating in the same pathway [2], [3], [5]. There are a number of clustering techniques which are used in clustering Microarray data such as K-means, self-organizing maps, support vector machines, hierarchical clustering, and many others. In this work, we have chosen to implement K-means clustering in hardware because it is one of the commonly used techniques by scientists studying Microarray, and because the algorithm itself can benefit from hardware acceleration.

2.1. K-means Clustering

K-means clustering is one of the unsupervised computational methods used to group similar objects in to smaller partitions called clusters so that similar objects are grouped together [3], [5]. The algorithm aims to minimize the within cluster variance and maximize the intra cluster's variance. The technique involves determining the number of clusters at first and randomly assigning cluster centroids to each cluster from the whole datasets; this step is called initialization of cluster centroids. The distance between each point in the whole datasets and every cluster centroid is then calculated using a distance metric (e.g. Euclidean distance) [6]-[9]. Then, for every data point, the minimum distance is determined and that point is 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 for each cluster is calculated based on the accumulated values of points in each cluster and the number of points in that cluster. Those means are then assigned as new cluster centroids, and the process of finding distances between each point and the new centroids is repeated, where points are re-assigned to the new closest clusters. The process iterates for a fixed number of times, or until points in each cluster stop moving across to different clusters. This is called convergence. The steps of the K-means can be summarized in Figure 2 below.

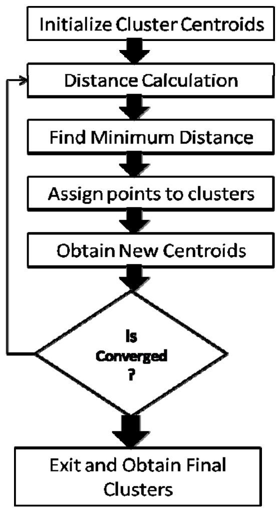
Initialize Cluster Centroids
Distance Calculation
Find Minimum Distance
Assign points to clusters
Obtain New Centroids
Is Converged ?
Exit and Obtain Final Clusters

Figure 2. Computational steps of the k- means algorithm

Euclidean metric is considered as it is one of the widely used distance metrics incorporated with K-means clustering and one that is easy to implement. Also it results in a best solution [8]-[9]. Euclidean distance is given in (1) below:

$$D(P,C) = \sqrt{\sum_{i}^{n} | P - C |^2} \qquad (1)$$

where P is the data point, C is the cluster center, and n is the number of features. However, the Euclidean distance consumes a lot of computational resources when implemented in hardware due to the multiplication operation used for obtaining the square operation; this reduces the amount of parallelism that can be exploited due to the need for calculating distances over and over. This is especially true when using a large number of clusters. Thus, previous groups working on hardware implementation of K-means clustering for image segmentation have used an alternative distance metric called the Manhattan distance shown in (2) below:

$$D(P,C) = \sum_{i}^{n} |P - C| \tag{2}$$

where P is again the data point, C is the cluster center, and n is the number of features. Their results showed that it performed faster than the Euclidean metric , because it does not require calculating the square offering better exploitation of parallelism and speed twice than that obtained by Euclidean distance [6]-[12]. However, the accuracy of this distance measure was found to be slightly inferior to the Euclidean metric, but results were still within an acceptable error as presented in [8]-[10]. The time needed to complete the clustering method for a whole datasets depends on the size of the set and the selected number of clusters: the larger they are, the longer it will take to compute the distances. 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 accelerating the distance computation part, which, as mentioned above can be made possible by using simpler distance metric such as the Manhattan metric and by moving this task to hardware such as FPGA, where we can benefit from the parallelism it can offer.

2.2. Related Work in Hardware Implementation of K-means Clustering Method

The work done in hardware implementation of K-means has been mostly in the area of multispectral and hyperspectral imaging with few in other areas. The following review is for related works ordered chronologically.

In 2000, Dominique Lavenier at Los Alamos National Laboratory implemented systolic array architecture of K-means clustering on a number of FPGA boards [6]. He aimed to parallelize the most computationally intensive part, which is the distance calculation, by running 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. His work involved storing the image in a host, streaming it to the FPGA for processing, and sending results back to the host for calculating the new means, then again streaming the new means back to the FPGA. This approach was effective in allowing for any data size to be processed, however, the disadvantage was the communication overhead between the host and the FPGA. Lavenier tested his design on several processing boards, and one of the relevant speed-up he obtained was 15x for the case of considering the data transfer rate between host and board [6], [7].

In 2001, Michael Estlick *et al.* implemented K-means in hardware using software/hardware co-design [8]. Their design was partitioned between hardware and host microprocessor, where distance calculation was calculated in hardware in purely fixed-point while new means were calculated in the host to avoid consuming large hardware resources. The design was tested on a platform board called the Wildstar, which housed three FPGAs, 40MB ZBT SRAM. Before the clustering begins, the data is moved from the host and stored in the ZBT SRAM while the initial centroids were stored in registers within the FPGA chip. The hardware implementation achieved a speed up of 50x more than the 500 MHz Pentium III host processor. 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 bitwidth of the input data [8]-[10].

In 2002, Pavel Belanoviv [11] created a library of hardware modules for floating point arithmetic. As an application to use this library, the authors of [11]-[13] implemented K-means clustering using a hybrid implementation of fixed and floating-point arithmetic instead of the purely fixed point arithmetic previously implemented in [9]. The throughput of the hybrid design distance calculation was 8 pixels per clock cycle as compared to one pixel per clock cycle achieved in [9]. Results showed that their hardware occupied larger slices within the FPGA than in [9], and had smaller throughput.

In 2003, Venkatesh Bhaskaran [14] implemented a parameterized implementation of K-means algorithm on FPGA. All of the K-means steps were done in hardware, except the initialization of cluster centroids which was done in a host. This was the first work to implement the division operation within the hardware to obtain the new means, using dividers from Xilinx Core Generator. However, his design was tested only on three clusters and achieved a speed-up of 500x over Matlab implementation including the I/O overhead. One disadvantage of his implementation was that the board he used did not have any memory capability, which restricted the size of image that can be processed at one time to a size that can be accommodated by the FPGA Block RAMs, a clear limitation of the design [14].

In 2003, Filho *et al.* [15] implemented a hybrid implementation of K-means using Euclidean distance metric and obtained speed-up of 2x over software implementation even though the former was running at 12.5 times lower frequency than the latter.

In 2007, Xiaojun Wang [16] proposed a variable precision floating point divider for efficient FPGA implementation. His work was an extension of the work presented in [11] and [12]. As an application, he implemented K-means clustering in FPGA and utilized a floating point divider to calculate new means within the FPGA. This approach required the use of an extra block to convert the fixed-point data to floating point, and then after the division was done, another floating to fixed-point converter was needed. He compared the speed up of the hardware implementation using his floating point divider with a hardware implementation doing the division in host and concluded that no speed-up or advantage of implementing the division in hardware was gained. The only advantage of such implementation according to Wang was to free the host to work on other tasks while K-means clustering was performed in hardware [16], [17].

Additional work has been reported in literature about hardware implementation of K-means in other areas such as document clustering [18] and anomaly detection in computing networks [19].

Given that today's FPGAs have resources that outnumber those used in some of the above mentioned implementations, we anticipate better timing performance in our design and more efficient use of power. Our work varies from the above mentioned implementations in that it is the first attempt to implement K-means fully in hardware to process Microarray data. We are also targeting a server solution where multiple K-means cores operate together in the same FPGA. In addition, our approach makes use of the larger Block RAMs capabilities available in recent FPGAs, which can accommodate the commonly used Microarray datasets, in addition to the capability of using off chip memory for cases where datasets are larger than the available Block RAMs.

3. Methodology for Hardware Implementation

Most FPGA designs are done in fixed-point due to the cost and complexity of using floating point. Therefore, before designing our hardware we need to first implement the design in floating point using Matlab or C/C++ and then convert it to fixed-point. For fixed-point designs, the wordlength of the input data, intermediate, and final results need to be defined as detailed in the following sections.

3.1. Pre-processing Analysis

We begin by looking at samples of Microarray data to determine common data size, dynamic range, precision, and memory requirement to see whether data can be accommodated within the FPGA or if an external memory solution is required. As an example which we will use throughout this paper, we looked at expression profiles of 14 hour Microarray experiment of Yeast Saccharomyces Cerevisiae, commonly known as baker's yeast, and found that expression profiles have a size of 6400×7 [20]. Such matrix contained irrelevant data commonly known as noise, which usually consists of empty cells or missing profiles, zero expression profiles, low variance over time, and low entropy profiles. One important pre-processing step is to filter out the data and remove genes containing such noise. This was done using the Matlab Bioinformatics toolbox, which included a number of functions to remove genes with missing expression profiles and empty spots, to filter out genes with low variance over time, to remove genes that have very low absolute expression values, and to remove genes with low entropy values [21]. This filtering reduced the above matrix to 415 × 7, making it easier to handle, and limited the datasets to those containing relevant information only. Although we have noted earlier that our work aims to provide a solution for clustering large datasets, we will only use this filtered 416 × 7 yeast datasets to test our software and hardware throughout this paper. Our design will still hold for larger datasets.

3.2. Wordlength Analysis

Before we go any further, we need to check the dynamic range of the datasets and check its suitability for fixed-point implementation, since a high dynamic range will mean large wordlength and this may render fixed-point costly. To determine the fixed-point wordlength we performed the range and precision analysis discussed below, similar to what was suggested in [22].

3.2.1. Fixed-point Range Analysis

The aim of this step is to obtain the minimum number of bits required to represent the integer part of the signed input using (3) below:

$$QI \geq \left[\log 2\left(\left|Range\right|\right)\right] \qquad (3)$$

where QI is the integer part, Range is the difference between the maximum and minimum values in the datasets. Back to our previous example, the range of our datasets was [-3.2780, 3.5460], therefore applying (3) results in a minimum of 3 bits required to represent the integer part of our input datasets.

3.2.2. Precision Analysis

The aim now shifts towards obtaining the number of bits required to represent the fractional part of the datasets, given that we decided that three digits were needed after the decimal point by examining the datasets, thus the precision needed is 0.001. The fractional bits can be obtained using (4) below:

$$QF \geq \left\lceil \log 2\left(\frac{1}{Precision}\right)\right\rceil \qquad (4)$$

where QF is the fractional part's bit length. Back again to our previous example, applying (4) results in a minimum of 10 bits to represent the fractional part of our datasets. The total wordlength for representing the data is *QI + QF,* therefore, we need 13 bits to represent our input datasets.

3.2.3. Wordlength of Distances and Accumulators

Repeating the above range and precision analysis for every step in the K-means algorithm such as the distance and accumulation parts is very important to make sure that we do not get an overflow or underflow condition. Accumulating values will result in a wordlength requirement above the 13 bits we used for representing the input data. If we choose a random high value to represent the distance and accumulator results we risk using more resources in the FPGA than we actually need. Therefore, performing the above analysis will optimize our implementation as we will eliminate the use of extra bits for representing those results. We have run our Matlab K-means algorithm, examined the distances, accumulators results, and applied (3) and (4) to them. We found that 15 bits (5 integers, 10 fractional) were required to represent the distances, and 25 (15 integers, 10 fractional) bits for the accumulators to achieve a precision level of three fractional digits. We have automated the above steps in Matlab, to provide an easy tool for performing this analysis on any datasets.

3.3. Convert Algorithm to Fixed Point

Having determined the required integer and fractional bit lengths, we used Matlab Fixed-point toolbox to convert the floating point K-means algorithm to fixed-point, which will be used to compare with the hardware implementation.

3.4. Memory requirement

The next step is to decide on the memory size needed for both the input data and the results, and where to store the input, centroids, and results. Also at this point we need to know what FPGA we are using and whether we will have access to external memory. The ideal situation would be is to have enough Block Rams to store our datasets, and avoid memory access bottleneck. However, this is not always the case especially when using huge datasets, thus streaming data from an external memory is a possible option.

4. Hardware Architecture and design

Our hardware design consists of three main blocks carrying out the whole K-means clustering. Figure 3 below shows the main processing blocks needed for our hardware design. The first block is used to calculate distances, which is the most computationally intensive part as it requires calculating distances between each data point and every cluster centroid all in parallel. In our current design we use eight clusters, therefore we have eight distance calculating processing elements working in parallel.

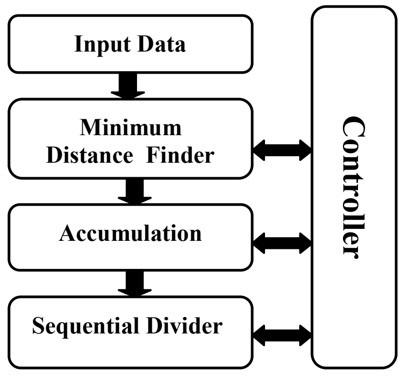

Figure 3. The main hardware blocks

The input data is stored in Block RAMs within the FPGA, and the initial centroids are stored in registers within the FPGA. The first block reads one data point from the on chip Block RAMs every clock cycle, obtains the eight distances simultaneously, and then obtains the absolute values of these distances. Secondly, these eight absolute distances run through a comparator tree as shown in Figure 4 below to obtain the minimum distance for each data point and its index, this takes two clock cycles for it to complete. This whole process is fully pipelined to have a throughput of one result every clock cycle, but it has a latency of four clock cycles: one for reading the input data point and obtaining the distances, one for obtaining the absolute of these distances, two for the comparison tree to obtain the minimum distance and its index. The second block consists of assigning points to one of the eight clusters based on the results of the previous block, and counting the number of points in every cluster. The outputs of this block are eight accumulators and eight counters. This block has a throughput of one result per clock cycle with latency of one clock cycle. However, the final results will be passed to the next block when accumulators finish assigning all points to the accumulator. The third block is a sequential divider which calculates the new means in hardware; the latency of the divider is 60 clock cycles and the throughput is one result per clock cycle. And since we are calculating eight means sequentially, the total time in which the divider will be active is 68 clock cycles. The divider itself was obtained using the core generator tool available with Xilinx ISE Design Suite 12.2. The divider block is activated by the previous block that is when the second block finishes assigning all points to clusters. Figure 4 shows a simplified architecture of our main pipeline.

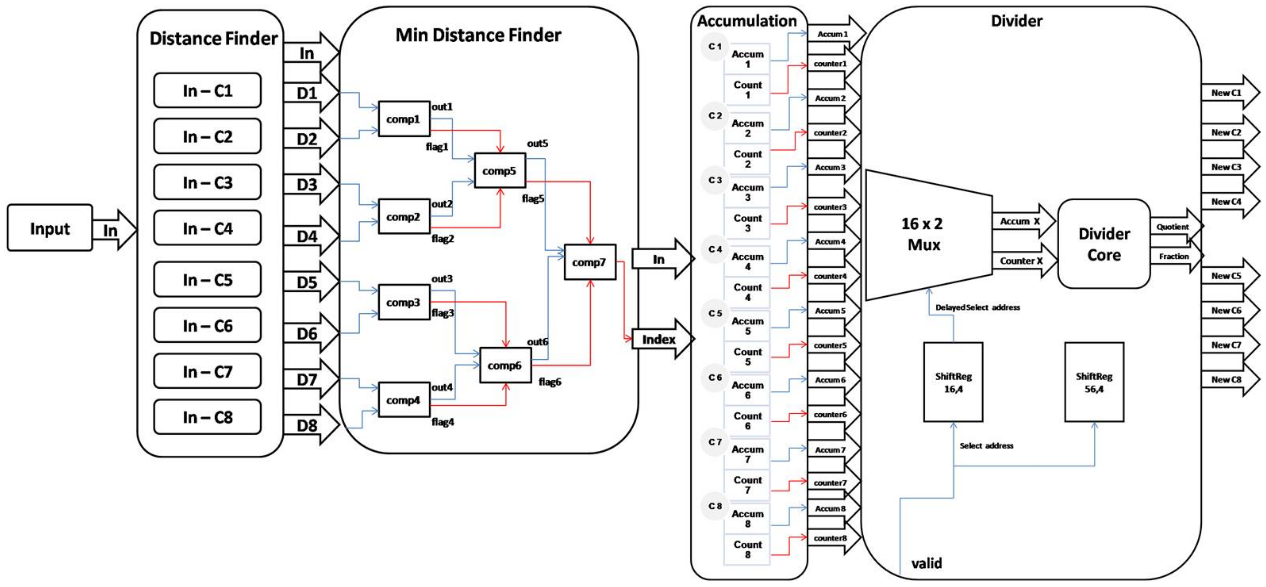

Figure 4. The main pipeline showing the three main processing blocks

The process iterates until an end condition is reached and tested for inside the controller, which we decided to be reaching a point where previous centroids do not vary from newly calculated ones, with an acceptable 1% error. The number of cycles the whole pipeline takes to complete is modelled in (6) below:

$$\begin{matrix}\textit{Time to write} \\ \textit{to Block RAM}\end{matrix} + \begin{matrix}\textit{Time to process} \\ \textit{all data}\end{matrix} + \begin{matrix}\textit{Divider} \\ \textit{Time}\end{matrix} \qquad (6)$$

This is for the first iteration, but for subsequent iterates, (6) is reduced to (7) below:

$$\textit{Time to process all data + Divider Time} \qquad (7)$$

The above hardware design was first captured in Verilog hardware description language. Then it was functionally simulated using Mentor Graphics ModelSim SE 6.0 software. Finally, it was synthesized, translated, mapped, placed and routed using Xilinx ISE Design Suite 12.2, to target Xilinx Virtex4 XC4VFX25. Current design works to cluster a data set into eight clusters only and we aim to expand it and parameterize it in terms of the number of clusters in subsequent work.

5. Implementation Results

Our design above was first implemented in Matlab, and tested using real data from the yeast Microarray datasets mentioned earlier. A testbench was then written to verify the Verilog model, and run the simulation using the real yeast datasets. Simulation results show that it takes 2971 clock cycles, to cluster 2905 points (415 × 7) based on (7), assuming that data are already written to Block RAMs. The algorithm converged after 25 iterations, thus taking a total of 74275 clock cycles. This result does not take into consideration the time needed to write data to the FPGA Block RAMs, which is also 2905 clock cycles. However, the result does include the time to write results to the FPGA Block Rams.

5.1. Software Implementation

Matlab was used to model the K-means algorithm using Manhattan distance metric, and the model was tested on an Intel Core2 Duo 3.0 GHz CPU, with 3 GB RAM running on Windows XP Professional operating system. The average execution time of the model for 1000 runs of the algorithm was 0.0062 ± 1.22e-4 s, with minimum execution time of 0.0060 s and maximum execution time of 0.0072 s. These results are based on initial centroids being pre defined and given as an input to the algorithm.

5.2. Hardware Synthesis Results

First we have implemented our design on Xilinx XC4VLX25-10SF363 using just a single core, and we have achieved a maximum clock frequency of 126 MHz. Table 1 shows the implementation results of our hardware design, which was obtained using Xilinx ISE 12.2. We have found that our single core occupies 2,208 slices, which is only 20% of the FPGA floor area.

Table 1. Implementation results for the single core model on Xilinx XC4VLX25-10SF363

Logic Utilization	Used	Available	Utilization (%)
Block RAMs	5	72	6
Occupied slices	2,208	10,752	20
Total slice registers	3,022	21,504	14
# of 4 input LUTs	2,948	21,504	13
# of Bounded IOBs	7	240	2
# used as BUFGs	5	32	15
Clock Frequency	126 MHz		

The distribution of occupied slices across the main design blocks is shown in Table 2 below, which gives an indication about the total logic consumed by each of the individual blocks. It indicates that the design occupies only 20% of the FPGA floor area, 7% of which is used in the actual computation performed by both the Minimum Distance Finder Block and the Accumulation-Counting Block, while 8% of the floor area is consumed by the Divider Block, which represents the highest slice logic in the whole design as shown in Table 2.

Table 2. Distribution of slices across all design blocks

Blocks	Used slices	Utilization (%)
Minimum Distance Finder	323	3
Accumulation and Counting	470	4
Divider	967	8
Controller	432	4
Input RAM	16	1
Results RAM	1	0
Total	2,208	20

Having only 20% of the device floor area occupied by the design, we found a potential to replicate the whole design five times before we run out of floor area, this approach can accelerate the run time of the algorithm by five times and provide a server solution for processing multiple datasets simultaneously. We have implemented this approach using the same datasets and obtained the results shown in Table 3. Table 3 shows that we have obtained a maximum frequency of 124 MHz, and consumed 99 % of the FPGA floor area.

To obtain the hardware timing, we used both the simulation result obtained from Modelsim, and the clock frequency of our hardware implementation, and then applied (8) as shown below:

$$\frac{Hardware}{ExecutionTime} = \frac{\#clock\ cycles\ per\ iteration \times \#iteration}{Clock\ Frequency} \quad (8)$$

where "clock cycles per iteration" refers to the number of clock cycles to complete one iteration, and the clock frequency is the frequency obtained from the hardware implementation results as shown in tables 1 and 3.

Table 3. Implementation results for the five cores model on Xilinx XC4VLX25-10SF363

Logic Utilization	Used	Available	Utilization (%)
Block RAMs	25	72	34
Occupied slices	10,750	10,752	99
Total slice registers	15,120	21,504	70
# of 4 input LUTs	14,705	21,504	68
# of Bounded IOBs	27	240	11
# used as BUFGs	16	32	50
Clock Frequency	124 MHz		

For the single core design, simulation results showed that it takes about 2971 clock cycles to complete one full iteration as was detailed in (7), and the datasets required 25 iterations to converge, thus hardware execution time is just 589 µs as computed from (8), given that the clock frequency is 126 MHz. Table 4 summarizes the speed-up that we have achieved when implementing our hardware model in FPGA, as compared to the Matlab software implemented in 3GHz Intel Core 2 Duo CPU.

Table 4. Timing performance of hardware and software implementations for a 2905x1 Microarray datasets of baker's yeast

Software (ms)	Hardware (ms)	Speed- up Single core	Speed-up Five cores
6.1	0.59	10.3x	51.7x

Our hardware achieved high timing performance, with speed-up of 10.3x for the case when implementing the single core, and 51.7x for the case when implementing the five cores approach. We can even get higher speed-up in larger FPGA devices which allow for more cores to be fitted into the device. Our multi-core approach is only limited by the logic resources available in the FPGA.

With regards to the power consumption of our hardware design, both of our implementations are energy efficient, since the floating point implementation consumes about 120 W when run in modern CPU's, while most FPGA boards consume no more than 30 W [23]. This means that an FPGA board is consuming 4 times less power than a CPU, while running 51.7 times faster in the case of the five cores implementation. Therefore, the total energy of our single core implementation is estimated to be 41.2 times less than the energy of the single core run on CPU, and for the five cores implementation the energy is estimated to be 206.8 times less than a CPU. These Energy figures are computed using (9) below:

$$Energy\ Effeciency = Power\ Effeciency \times Speed\ up \quad (9)$$

where power efficiency is the power consumed by the CPU divided by the power consumed by the FPGA, which in our case is 4x.

We have actually implemented the single core K-mean in FPGA using Xilinx ML 403 board which has Xilinx XC4VFX12 FPGA. The board runs at 100 MHz, thus the speed-up achieved was 8.2x compared to CPU, slightly less than the speed-up that can be achieved with the

FPGA mentioned earlier due to the difference in the clock frequencies. Then we have measured the power consumed by the board when running the single core K-mean in the FPGA and when running the same algorithm in CPU. We have found that the single core consumes only 10 W as compared to 90 W for the CPU. This shows that a single core K-mean consumes nine times less power than CPU, thus it is 73.8 times more energy efficient than the CPU.

6. Conclusion

In this paper, we have presented FPGA hardware design of the K-means algorithm for Microarray data mining, which proved to be power efficient and useful for server solution. Our results show promising speed-up potentials of FPGAs in Microarray data analysis. The speed-up we have achieved when processing 2905 gene expressions into eight clusters is 10.3 times more than the software implementation when using single core, and 51.7 times more when using five cores. This was due to applying concepts of pipelining, parallelism, and multi-core processing. Furthermore, our five cores implementation is 206.8 times more energy efficient than the CPU implementation. In summary, hardware implementation of K-means algorithm has the potential for speeding up data analysis of Microarray datasets. The work presented in this paper shows the early results of a project that aims to explore the computational performance of FPGAs in the area of Microarray data mining. Future work includes enhancing speed further, parameterizing the design to accept any number of clusters and multidimensional data, implement other Microarray data mining techniques in FPGA. In addition, compare with other hardware implementations such as GPU.

7. References

[1] B. Andreopoulos, A. An, X. Wang and M. Schroeder, "A roadmap of clustering algorithms: finding a match for a biomedical application", *J. Briefings in Bioinformatics*, vol. 10, no. 3, pp. 297-31, 2009.

[2] D. Stekel, *Microarray Bioinformatics*, 1st ed. Cambridge University Press, 2003.

[3] P.F. Macgregor and J.A. Squire, "Application of Microarrays to the Analysis of Gene Expression in Cancer", *Clinical Chemistry*, vol. 48, pp. 1170-1177, 2002.

[4] The colours of a Microarray, [image online] Available: http://www.ncbi.nlm.nih.gov/About/primer/microarrays.html.

[5] M. Akay, *Genomics and Proteomics Engineering in Medicine and Biology* (IEEE Press Series in Biomedical Engineering*),* Canada: A John Wiley & Sons, 2007.

[6] D. Lavenier, "FPGA implementation of the K-means clustering algorithm for hyperspectral images", Los Alamos National Laboratory, LAUR # 00-3079, pp. 1-18, 2000.

[7] M. Gokhale, J. Frigo, K. McCabe, J. Theiler, C. Wolinski, and D. Lavenier, "Experience with a hybrid processor: K-means clustering," *J. Supercomputing*, vol. 26, pp. 131-148, 2003.

[8] M. Estlick, M. Leeser, J. Theiler, and J.J. Szymanski, "Algorithmic Transformations in the Implementation of K-means Clustering on Reconfigurable Hardware," in *International Symposium on Field Programmable Gate Arrays,* pp. 103-110, 2001.

[9] M.D. Estlick, "An FPGA Implementation of the K-means Algorithm for Image Processing", M.S. thesis, Dept. Elect. Eng., Northeastern Univ., Boston, MA., 2002.

[10] J. Theiler, M.E. Leeser, M. Estlick, and J.J. Szymanski, "Design Issues for Hardware Implementation of an Algorithm for Segmenting Hyperspectral Imagery", *Imaging Spectrometry VI*, Volume 4132, pp. 99-106, Edited by Michael R. Descour and Sylvia S. Shen. Bellingham, WA: The International Society for Optical Engineering, 2000.

[11] P. Belanovic, "Library of Parametrized Hardware Modules For Floating-Point Arithmetic with an Example Application," M.S. thesis, Dept .Elect. Eng., Northeastern Univ., Boston, MA., 2002.

[12] 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", *Proc. SPIE* pp.4480, (2002).

[13] M. Leeser, P. Belanovic, M. Estlick, M. Gokhale, J. J. Szymanski, and J. Theiler, "Parameterized k-means clustering for rapid hardware development to accelerate analysis of satellite data," in *High Performance Embedded Computing (HPEC) Workshop*, MIT Lincoln Laboratory, 2001.

[14] V. Bhaskaran, "Parametrized Implementation of K-means Clustering on Reconfigurable Systems", M.S. thesis, Dept. Elect. Eng., Univ. of Tennessee, Knoxville, TN, 2003.

[15] A.Gda.S. Filho, A.C. Frery, C.C. de Araujo, H. Alice, J. Cerqueira, J.A. Loureiro, M.E. de Lima, Mdas.G.S. Oliveira, M.M. Horta, "Hyperspectral images clustering on reconfigurable hardware using the k-means algorithm", *Pros. of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI'03)*, Brazil, pp. 99-104, 2003.

[16] X. Wang, "Variable Precision Floating-Point Divide and Square Root for Efficient FPGA Implementation of Image and Signal Processing Algorithms", Ph.D. dissertation, Dept. Elect. and Comp. Eng., Northeastern Univ., Boston, MA, 2007.

[17] X. Wang and M. Leeser, "K-means clustering for multispectral images using floating-point divide", in *15th Annu. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*, pp. 151-159, 2007.

[18] G.A. Covington, L.G. Comstock, A.A. Levine, J.W. Lockwood, and Y.H. Cho, "High Speed Document Clustering In Reconfigurable Hardware", in *16th Annu. Conference on Field Programmable Logic and Applications (FPL)*, Madrid, Spain, pp.411-417, 2006.

[19] K. Labib and V.R. Vermuri, "A Hardware-Based Clustering Approach for Anomaly Detection", *International Journal of Network Security* (submitted Aug 2005).

[20] 14 hr Microarray Yeast datasets, [datasets online] Available: http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE28

[21] Matlab Bioinformatics Toolbox, [online] Available: http://www.mathworks.com/products/bioinfo/

[22] A. Nayak, M. Haldar, A. Choudhar, and P. Banerjee, "Precision and Error Analysis of MATLAB Applications during Automated Hardware Synthesis for FPGAS", in *Proc. of the Conf. on Design, Automation, and Test in Europe*, Grenoble, France, pp. 722-728, 2001.

[23] S. Kentaro, T. Nishikawa, T. Aoki, and S. Yamamoto, "Evaluating power and energy consumption of FPGA-Based custom computing machines for scientific floating-point computation", in *Inter. Conf. on Field-Programmable Technology (ICFPT'08)*, Taipei, Taiwan, pp.301-304, 2008.