A FAST AND SCALABLE HARDWARE ARCHITECTURE FOR K-MEANS CLUSTERING FOR BIG DATA ANALYSIS

By

RAMPRASAD RAGHAVAN

B.E., University of Mumbai, 2004

A Thesis submitted to the Graduate Faculty of the
University of Colorado at Colorado Springs

in partial fulfillment of the
requirements for the degree of

Masters of Science in Electrical Engineering

Department of Electrical Engineering

2016
This thesis for Master of Science Degree by
Ramprasad Raghavan
has been approved for the
Department of Electrical Engineering
by

___________________________
Dr. Darshika G. Perera, Chair

___________________________
Dr. T.S. Kalkur

___________________________
Dr. Charlie Wang

___________________________
Date
ABSTRACT

The exponential growth of complex, heterogeneous, dynamic, and unbounded data, generated by a variety of fields including health, genomics, physics, and climatology pose significant challenges in data processing and desired speed-performance. Existing processor-based (software-only) algorithms are incapable of analyzing and processing this enormous amount of data efficiently and effectively. Consequently, some kind of hardware support is desirable to overcome the challenges in analyzing big data. **Our objective is to provide hardware support for big data analysis to satisfy the associated constraints and requirements.**

Big data analytics involves many important data mining tasks including clustering, which categorizes data into meaningful groups based on the similarity or dissimilarity among objects. In this research work, we investigate and propose customized hardware architecture for K-means clustering, one of the most popular clustering algorithms. Our hardware design can execute multiple computations in parallel to significantly enhance the speed-performance of the algorithm, by exploiting the inherent parallelism and pipelining nature of the operations.

We design and develop our hardware architecture on a Field Programmable Gate Array (FPGA)–based development platform. Experiments are performed to evaluate the proposed hardware design with its software counterpart running on an embedded processor on the same development platform. Different hardware configurations (consisting of varying number of
parallel processing elements) are processed on varying data sizes. Our hardware configuration consisting of 32 parallel processing elements (PEs) is executed up to 150 times faster than the software-only solution that is executed by the processor. It is observed that the speed-performance further increases with the number of parallel PEs as well as with the size of the data.

These investigations demonstrate that hardware support for clustering algorithms is not only feasible but also crucial to meet the requirements and constraints associated with analyzing and processing big data. Our proposed hardware architecture is generic and parameterized. It is scalable to support larger and varying datasets as well as a varying number of clusters.
Dedicated to my family.
ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor and mentor, Dr. Darshika G. Perera, for giving me an opportunity to conduct this research work. Her immense knowledge in the field of Data Mining was a guiding light on many occasions. I am also very grateful to her for taking out a lot of time to thoroughly go through this thesis document and provide invaluable feedback.

I am also thankful to the thesis panel members, Dr. T.S. Kalkur and Dr. Charlie Wang, for setting aside time from their academic and professional commitments to review and evaluate my research work.

A special thanks to my fellow researchers, Navid, Mong and Anne, for providing excellent suggestions during early brainstorming sessions. I would also like to appreciate the efforts put by Ashley and Christina from the UCCS Writing Center to proof-read this document and help me polish it and perfect it.

Most importantly, I feel very blessed to have unconditional love and support of my family – my mother, Rajeshwari, my wife, Nisha and my daughter, Rashi, who endured my absence from home, especially during weekends. Their constant support and motivation inspired me to complete this research.
TABLE OF CONTENTS

1 INTRODUCTION AND MOTIVATION ............................................................................................................. 1
  1.1 Our Research Objectives ........................................................................................................................... 3
  1.2 Thesis Organization ....................................................................................................................................... 4
2 BACKGROUND ...................................................................................................................................................... 6
  2.1 Clustering and Classification ........................................................................................................................ 6
  2.2 Different Clustering Algorithms ................................................................................................................... 7
  2.3 K-Means Clustering Algorithm ..................................................................................................................... 8
  2.4 Related work on hardware architectures for K-Means Clustering ............................................................. 11
3 DESIGN APPROACH AND DEVELOPMENT PLATFORM .................................................................................. 15
  3.1 Experimental Platform ............................................................................................................................... 15
  3.2 Hierarchical Platform-based Design Approach ............................................................................................ 17
  3.3 System-Level Interface ................................................................................................................................ 19
    3.3.1 MicroBlaze Soft Processor Core .............................................................................................................. 20
    3.3.2 DDR3 Controller ....................................................................................................................................... 20
    3.3.3 DDR3-SDRAM External Memory ............................................................................................................. 20
    3.3.4 AXI Interconnect ..................................................................................................................................... 21
    3.3.5 UART Lite ................................................................................................................................................. 22
    3.3.6 IP Version and Memory Map .................................................................................................................. 22
  3.4 Benchmark Dataset ...................................................................................................................................... 22
4 EFFICIENT EMBEDDED ARCHITECTURE FOR K-MEANS .................................................................................. 24
  4.1 Embedded Hardware Architecture ................................................................................................................ 24
    4.1.1 K-Means Top Module ............................................................................................................................. 24
    4.1.2 Top Level State Machine ........................................................................................................................ 26
    4.1.3 Data Engine ........................................................................................................................................... 28
    4.1.4 Data Pre-Fetch Memory ......................................................................................................................... 30
    4.1.5 Cluster Memory .................................................................................................................................... 31
    4.1.6 Distance Measure ................................................................................................................................. 33
    4.1.7 Centroid Measure ................................................................................................................................. 36
    4.1.8 Xilinx LogiCORE CoreGen IPs ............................................................................................................ 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.9</td>
<td>Hardware Parameter List</td>
<td>41</td>
</tr>
<tr>
<td>4.1.10</td>
<td>K-Means Register Interface Module</td>
<td>42</td>
</tr>
<tr>
<td>4.1.11</td>
<td>Block-level verification methodology</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>Embedded Software Architecture</td>
<td>49</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Software Design flow</td>
<td>49</td>
</tr>
<tr>
<td>4.2.2</td>
<td>HW/SW Task Flow</td>
<td>51</td>
</tr>
<tr>
<td>5</td>
<td>RESULTS AND ANALYSIS</td>
<td>52</td>
</tr>
<tr>
<td>5.1</td>
<td>Execution time for Embedded HW versus Embedded SW</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>Impact of Parallel Hardware on speedup</td>
<td>56</td>
</tr>
<tr>
<td>5.3</td>
<td>Resource Utilization</td>
<td>57</td>
</tr>
<tr>
<td>6</td>
<td>CONCLUSIONS AND FUTURE WORK</td>
<td>61</td>
</tr>
<tr>
<td>6.1</td>
<td>Conclusions</td>
<td>61</td>
</tr>
<tr>
<td>6.2</td>
<td>Future Work</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
<td>64</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 3-1: Xilinx Platform Studio IP Cores and System Address 22
Table 4-1: Divider Configuration 39
Table 4-2: Multiplier IP Configurations 40
Table 4-3: Block Memory Configurations 40
Table 4-4: Xilinx ISE CoreGen IP Versions 41
Table 4-5: Design Parameters 42
Table 4-6: Hardware Register Map 44
Table 4-7: Status and Control field descriptions 46
Table 5-1: Execution time for Hardware and Software (8 Clusters) 53
Table 5-2: Execution time for Hardware and Software (16 Clusters) 54
Table 5-3: Execution time for Hardware and Software (32 Clusters) 54
Table 5-4: Overall Resource Utilization 58
Table 5-5: Resource utilization for K-Means Register Interface 59
Table 5-6: Resource utilization for the K-Means Hardware for three configurations 59
LIST OF FIGURES

Figure 3-1: Hierarchical Platform-Based Design Approach ............................................................ 18
Figure 3-2: Top Level Hardware Block Diagram ............................................................................. 19
Figure 3-3: DDR3-SDRAM Organization .......................................................................................... 21
Figure 4-1: Block Diagram for K-Means Top Module ................................................................. 25
Figure 4-2: Top level state machine ............................................................................................... 26
Figure 4-3: Data Engine state machines ......................................................................................... 29
Figure 4-4: Pre-Fetch memory data structures ............................................................................. 31
Figure 4-5: Cluster Memory Data structure .................................................................................. 32
Figure 4-6: Distance Measure Module ......................................................................................... 33
Figure 4-7: Distance Measure Flow Diagram ............................................................................... 34
Figure 4-8: Centroid Measure Module ....................................................................................... 36
Figure 4-9: Centroid State Machine .......................................................................................... 37
Figure 4-10: K-Means Register Interface Module ......................................................................... 43
Figure 4-11: Lcd Display Segment Map ...................................................................................... 44
Figure 4-12: NumK & Attr Register ............................................................................................ 45
Figure 4-13: Status and Control Register ................................................................................... 46
Figure 4-14: Block-level Test Environment ................................................................................. 48
Figure 4-15: Embedded Software Design Flow ........................................................................ 50
Figure 4-16: HW/SW Task Flow ................................................................................................. 51
Figure 5-1: HW Speedup with varying number of vectors ......................................................... 55
Figure 5-2: HW Speedup with varying number of clusters ......................................................... 56
Figure 5-3: HW Speedup with varying number of parallel PEs .................................................. 57
Figure 5-4: Increase in Resource utilization Vs. Increase in Speedup ....................................... 60
**LIST OF ACRONYMS**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPGPU</td>
<td>General Purpose Graphics Processing Unit</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Elements</td>
</tr>
<tr>
<td>DNA</td>
<td>Deoxyribonucleic Acid</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interconnect</td>
</tr>
<tr>
<td>PCIe</td>
<td>Express Peripheral Component Interconnect</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver/Transmitter</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Access Group</td>
</tr>
<tr>
<td>FMC</td>
<td>FPGA Mezzanine Connector</td>
</tr>
<tr>
<td>HPC</td>
<td>High Pin Count</td>
</tr>
<tr>
<td>LPC</td>
<td>Low Pin Count</td>
</tr>
<tr>
<td>AXI</td>
<td>Advanced Extensible Interface</td>
</tr>
<tr>
<td>XPS</td>
<td>Xilinx Platform Studio</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DIMM</td>
<td>Dual In-line Memory Module</td>
</tr>
<tr>
<td>IPIC</td>
<td>Intellectual Property Interconnect</td>
</tr>
<tr>
<td>BFM</td>
<td>Bus Functional Model</td>
</tr>
<tr>
<td>FF</td>
<td>Flip Flop</td>
</tr>
<tr>
<td>LUT</td>
<td>Look Up Table</td>
</tr>
<tr>
<td>BRAM</td>
<td>Block Random Access Memory</td>
</tr>
<tr>
<td>DDR3-SDRAM</td>
<td>Double Data Rate 3 – Synchronous Data Random Access Memory</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION AND MOTIVATION

Data acquisition techniques and data storage media have evolved rapidly from the late-2000s. This has resulted in an exponential growth of complex, heterogeneous, dynamic, and unbounded data being generated by a variety of fields including health, genomics, physics, climatology, social networks, etc. For instance, in genomics, the amount of sequence data generated doubled every seven months within the last decade [1]. Currently, dedicated sequencing centers typically produce several petabytes of data per year [1]. Analyzing and processing such an enormous amount of data pose a serious challenge.

Big data analytics involves many important data mining tasks. In this research work, we focus on the most widely used data mining tasks: clustering and classification. Clustering and classification categorize the data into meaningful groups based on the similarity or dissimilarity among objects. These tasks have a wide variety of applications: image segmentation in computer vision applications; pattern recognition in customers’ buying habits from purchase histories; genome classification based on the Deoxyribonucleic Acid (DNA) data, to name a few. Classification is a form of supervised learning, whereas clustering is unsupervised learning [2]. The former needs prior assumptions and directions from a human user, which includes assigning labels or classes to the sub-groups by sieving through the data set. In contrast, the latter makes its own assumptions and automatically categorizes the data into sub-groups based on the similarity/dissimilarity among the objects.
Existing clustering and classification algorithms are typically processor-based (software-only) algorithms. These algorithms are incapable of analyzing and processing enormous amounts of data efficiently and effectively. In this case, an algorithm is realized using a software program. The processor interprets the program instructions and executes them to perform an operation. These program instructions are stored in the memory. The processor has to fetch each instruction from the memory, decode, and then execute, incurring higher execution overhead. Furthermore, due to the large volume, the data must be stored in an external memory, and fetched for analyzing and processing, leading to increased execution overhead. In some cases, this data needs to be processed in real-time to reap the actual benefit. A survey [3] demonstrated that processor-based computing platforms, including multi-processor, multi-core, GPGPU (General Purpose Graphics Processing Unit) are simply not sufficient to handle this enormous amount of data. Consequently, new design techniques, architectures, and computing platforms are needed to overcome the challenges in analyzing big data.

In order to satisfy the constraints and requirements associated with big data analytics, it is imperative to provide some kind of hardware support. In this research work, we investigate special-purpose hardware for big data analysis. Special-purpose or customized hardware is optimized for a specific application and avoids the high execution overhead of fetch/decode/execute instructions as in processor-based software-only designs. As a result, customized hardware provides higher speed-performance, lower power consumption [4], and area-efficiency compared to equivalent software running on a general-purpose processor.

We focus on the K-means clustering, one of the most popular clustering algorithms. We investigate and propose an efficient and scalable hardware architecture for K-means clustering.
Our hardware design can execute multiple computations in parallel to significantly enhance the speed-performance of the algorithm, by exploiting the inherent parallelism and pipeline nature of the operations. Our hardware architecture is designed and developed on a Field Programmable Gate Array (FPGA)-based development platform. The state-of-the-art FPGAs make it feasible to design, develop, and implement such compute- and data-intensive algorithms in hardware. This investigation demonstrates that our FPGA-based hardware design drastically reduces the execution time of the K-means clustering algorithm compared to the equivalent processor-based software-only design, thus significantly enhancing the speed-performance.

1.1 Our Research Objectives

The main objective of our research work is to provide hardware support for big data analysis to satisfy the associated constraints and requirements.

In order to achieve the main goal, our intention is to design and develop a scalable, generic, and parameterized hardware architecture for the K-Means clustering algorithm on a reconfigurable hardware platform. Using the parameters as external stimuli, we control the size of the dataset as well as the performance attributes. The performance attributes control the number of parallel processing elements (PEs) and also provide a trade-off between performance versus area. The parameter values are specified by the target application and also based on the available resources of the hardware platform.

As a secondary goal, our intention is to design and develop the K-Means clustering in software on an embedded processor on the same development platform, for fair comparison
purposes. Experiments are performed to evaluate our hardware architecture with its software counterparts. Different hardware configurations (with varying number of parallel PEs) are processed on varying data sizes. Experimental results illustrate the effectiveness of our design and also provides guidelines to researchers developing configurations best suited for their application needs.

1.2 Thesis Organization

This thesis is organized as follows. In the second chapter, we provide a general background on Clustering and classification algorithms, and then focus on the K-Means clustering. We investigate the existing research work on hardware architectures for K-means clustering, and illustrate how our contributions address some of the limitations of the existing designs.

Our design approach and the FPGA-based reconfigurable hardware platform used to design and develop our hardware and software architectures are described in the third chapter. We also present a list of Xilinx Intellectual Property (IP) design components used for our hardware implementation. Benchmark dataset for performing the experiments are also presented.

In the fourth chapter, we present our proposed generic, parameterized, and parallel hardware architecture for the K-Means clustering algorithm. The different components in the hardware design and the data path and control path flows are also detailed. Furthermore, we discuss and present how different parameters affect the hardware, both in terms of size and speed.
In the fifth chapter, we analyze and present the experimental results for different hardware configurations, the execution times for varying data sizes, and the trade-off between area and performance. The hardware and software execution times are compared and speedups are measured against a variety of parameters. The analysis of trade-off between performance and area is helpful in deriving an optimal hardware configuration for a specific application as well as a specific hardware platform.

The final chapter concludes the thesis and discusses potential future directions to further improve hardware support for K-Means clustering.
CHAPTER 2

BACKGROUND

In this chapter, we discuss and present different clustering algorithms commonly used in data mining and focus on the K-Means clustering algorithm. We also investigate the existing research work on hardware architectures for K-means clustering and illustrate how our design addresses some of the limitations of the existing designs.

2.1 Clustering and Classification

Clustering and Classification are one of the main tasks in many data mining applications [5]. They help in grouping or categorizing the data sets into sub-groups where objects within a sub-group should be as similar as possible, and objects among the sub-groups should be as dissimilar as possible. Clustering groups based on inter-pattern similarities; whereas Classification creates labels or groups based on a smaller sample/training set, and then defines rules that map the data, which is not in the training set, to those labeled [6],[7],[8]. Both the tasks work on the principle of “Divide and Conquer” where the resultant sub-groups are often easier and faster to process. These tasks also highlight a pattern or a structure in the original data sets that may not be very apparent. The two tasks, however, address the grouping in different ways. In Clustering, the grouping is done by the algorithm on the whole dataset, while in Classification the grouping/labeling pre-exists based on a training dataset and the algorithm defines/finds rules that map new vectors to the labeled ones. In this research work, we focus on clustering algorithms.
A variety of clustering algorithms has been developed to process useful information from the seemingly random datasets. A single algorithm may not point to all the hidden patterns, and often multiple algorithms will run on the same dataset to yield optimal results. This task is usually done by processor-based (software-only) algorithms. The existing processor-based algorithms are incapable of analyzing this complex and enormous amount of data efficiently and effectively. Due to their complexity, these algorithms require significant processing power, affecting the speed-performance. Clustering algorithms exhibit a significant amount of functional parallelism as well as data parallelism, and typically amenable to pipelining. These features can be exploited to a great extent in a customized hardware, thus enhancing the speed-performance of these algorithms. In the next sub-sections, we present some of the existing clustering algorithms and detail on the K-means clustering.

2.2 Different Clustering Algorithms

Clustering algorithms are broadly divided into Hierarchical and Flat Clustering [8],[9]. In Flat clustering, the dataset is categorized into a set of clusters not related to each other. Hierarchical clustering takes the flat clustering a step further by creating a hierarchy of clusters. The additional information in the latter comes at the cost of speed and efficiency. Algorithms like K-Means and Model-based clustering belong to the first type, while algorithms like Agglomerative (bottom-up), Divisive (top-down), and Centroid Clustering belong to the second type [9]. Another parameter on which clustering can be distinguished is hard and soft clustering. In hard clustering, each data element belongs to only one cluster. Conversely, in soft clustering, each data element belongs to multiple clusters thus forming overlapping clusters in the overall dataset. Soft clustering is much more complex compared to hard clustering and is typically used
for indexing of a large dataset that helps in speedup the data retrieval requests. For example, a search term Jaguar could mean a brand of Car or type of an Animal and will be found under a cluster of Cars and a cluster of Animals. A detailed description of different Clustering and Classification algorithms can be found in chapters 8-10 of the book by J. Han et al. [10].

For this research work, we select the K-Means flat clustering algorithm and propose an efficient hardware architecture. Subsequent sub-sections describe the K-means in detail and present the existing research work for its hardware design.

2.3 K-Means Clustering Algorithm

The K-Means clustering, as many other clustering algorithms, groups the dataset into clusters, where vectors within a cluster are as similar as possible, and vectors among the clusters are as dissimilar as possible. The algorithm executes until the Mean Similarity Measure of all clusters reaches the lowest value. The basic steps of the algorithm are as follows:

Step 1: Initialize the K Cluster Centers
Step 2: Evaluate the Similarity Measures among all the vectors and all the cluster centers. Assign the vector to the closest cluster center
Step 3: Re-compute all the cluster centers as the centroid of all the vectors belonging to that cluster
Step 4: Repeat Steps 2 and 3 until the cluster centers stop moving or after a specified number of iterations

The first step, which is the initialization of the cluster centers, is critical as the results may vary based on the starting point. The most widely used method is to randomly select K vectors from the dataset as the initial cluster centers. Bradley and Fayyad in [11] showed that
initial random cluster centers can be further refined by applying the clustering algorithm to a small random sub-sample of the whole dataset. These researchers claimed that the refined cluster centers produce a better local minima when the clustering algorithm processes the whole dataset. D. Arthur and S. Vassilvitskii proposed a more sophisticated algorithm in [12] that iteratively improves the initial random selection for the cluster centers. For both these techniques, the entire dataset needs to be parsed several times to obtain a refined cluster. Also, the results from [11] showed that in many cases, random selection yields results that are very close to the global minima. After this investigation, we selected the random selection of vectors as the initial cluster centers for our designs.

The second step measures the similarity between the vector and the cluster center. The similarity measure is a numeric measure, which indicates how close the cluster center is to the vector. There are several ways to measure similarity and dissimilarity among the vectors. One way is to use the Minkowski Distance [13] given by equation (1).

\[
distance = \left( \sum_{k=1}^{n} |p_k - q_k|^r \right)^{\frac{1}{r}}
\]

where, \( r \) is a parameter, \( n \) is the total number of attributes and \( p_k \) and \( q_k \) are the \( k^{th} \) attribute of the two vectors being compared.

Depending on the value of \( r \), we get the following similarity/distance measures:

- Manhattan Distance \( (r = 1) \):
  
  This is the basic city block distance and is the sum of the absolute difference between the attributes.
- Euclidean Distance \((r = 2)\):

  This is the straight line distance between the two points.

- Supremum Distance \((r \to \infty)\):

  This is the maximum distance that is possible to compute between two points.

Other widely used similarity measures are Cosine Similarity and Correlation.

\[
\text{Cosine } (p, q) = \frac{(p \cdot q)}{||p|| ||q||} \quad \ldots \ (2)
\]

where \( \cdot \) indicates vector product and \( ||x|| \) indicates length of the vector \( x \)

\[
\text{Correlation } (p, q) = p' \cdot q' \quad \ldots \ (3)
\]

where \( x' \) is the standardized data object \( x \) given by \( x'_k = \frac{x_k - \text{mean}(x)}{\text{std}(x)} \)

In 2001, M. Estlick et al. [14] evaluated several types of Distance Measures and came to the following conclusion: although the Euclidean Distance is the most accurate for K-Means, the Manhattan Distance is better suited for hardware designs due to a smaller footprint and higher speed. Since then, the FPGA technology has matured. The current state-of-the-art FPGAs’ consist of advanced features including compact and pipelined DSP Slices which can be used for complex math implementations. In this research work, we demonstrate that multiplication-heavy Euclidean Distance measure is now feasible for hardware development by utilizing the pre-existing multiplier IPs (Intellectual Property). This significantly improves accuracy of the hardware design in processing the algorithm, with a penalty of relatively smaller area.
The third step of the algorithm is to find the new cluster centers for the clusters computed in the second step. The new cluster centers are computed according to the centroid equation (4).

$$Centroid \ C_i = \frac{\sum_{k=0}^{n} d_k}{n} \quad \ldots \ (4)$$

where \( n \) is the number of attributes in \( i^{th} \) cluster

From equations (1) and (4), we observe that steps 2 and 3 of the algorithm can be processed faster if the attributes are computed in parallel. This inherent parallelism in the algorithm makes it suitable for hardware designs. In the next section, we investigate and present existing research work on hardware architecture for K-Means Clustering.

### 2.4 Related work on hardware architectures for K-Means Clustering

Many researchers have proposed different hardware architectures for the K-Means clustering algorithm. Before 2001, only compute-intensive parts of the algorithm, such as Distance Measure [15], [16], were designed and implemented in hardware. This was mainly due to lack of high capacity reconfigurable hardware at that time. These designs would run the algorithm on a host processor and stream the data to the dedicated hardware when performing Distance Measure computation. A major disadvantage of this approach was the I/O streaming overhead, which affects the speed-performance of the design. The authors, Leeser et al. [17], claimed to obtain about 15% speed improvement with a similar design and suggested a future hybrid architecture, which could potentially improve speedup by a factor of 10. The early FPGA-based design for the K-Means was targeted to applications for spectral and hyperspectral image
analysis. In 2003, V. Bhaskaran [18] proposed a parameterized hardware architecture for the K-Means for an image processing application. In this case, the author used a Pilchard board that supported the Xilinx Virtex 1000E FPGA as the reconfigurable hardware and provided a Peripheral Component Interconnect (PCI)-based access to the host system for I/O operations. This was one of the first hardware architectures proposed for the K-Means, where all the iterative stages of the algorithm are implemented on an FPGA. This design used the Manhattan Distance for the Distance measure and a fixed-point division IP for the Mean measure. It was tested using a dataset with three clusters, and each vector having three attributes. The author claimed to achieve a speedup of 500 times than its software counter-part; however, the speedup comparison was with a Matlab simulation with floating-point arithmetic running on a different processor and a platform. In 2006, Takashi and Maruyama [19] proposed another hardware architecture based on a KD-Tree filtering algorithm proposed by Kanungo et al. [20]. This is a modified version of the K-Means algorithm, and requires a smaller data structure to be maintained in hardware. This reduces the hardware footprint compared to the designs for traditional K-Means algorithm. However, this algorithm has a better performance, especially when the clusters are distributed far apart. This design was used to process pixel data and achieved a processing speed of 20-30 fps (Frames per Second) for a 768x512 pixel image. In 2011, Hussain et al. [21] introduced another parameterized architecture for the K-Means algorithm and developed it on a Xilinx Virtex-4 FPGA. Similar to the previous designs, their Distance measure kernel was also implemented with the Manhattan distance. This hardware design achieved a speedup of approximately 300 times over a software design; however, the software design was executed on a completely different processor platform. Also this design
was built to process a fixed number of vectors and fixed number of clusters, which is another disadvantage. Any change in the dataset size requires redesigning the hardware. In 2013, J.S. Kutty, et al. [22] proposed a hardware architecture that simultaneously computed Manhattan Distance measure for all the cluster centers. Their proposed design achieved a speedup of 3, compared to the existing FPGA-based architecture for K-Means clustering at that time. However, this design was implemented on an advance Virtex-6 FPGA, whereas the other architectures (used for comparison purposes) were implemented on older Xilinx FPGA versions. Since the CMOS process technology of an FPGA has a significant impact on the frequency and the occupied area of a certain design, these comparisons are not necessarily fair. More genuine evaluation of speedup would be to measure the execution time in clock cycles. Furthermore, system-level architecture was not provided, which is essential when executing a large amount of data that typically exists in data mining techniques including clustering.

From the above investigation, we observed the following issues in most of the existing hardware architectures for the K-Means algorithm:

- Compromise on Distance measure accuracy for an area-efficient hardware by using Manhattan distance instead of Euclidean distance.
- Compromise on division for Mean computation by using truncated integer division.
- Speedup comparison based on software running on a completely different processor platform. The software designs in most cases utilized full floating-point arithmetic, while corresponding hardware designs used integer arithmetic. The Euclidian distance in the software designs was replaced with Manhattan distance in the corresponding hardware.
designs, which is an approximate distance measure that requires fewer resources. These trade-offs make runtime comparisons between the two designs unfair.

- Except for [18], most designs also lacked the use of an industry standard protocol for transferring data between the processor and the FPGA. This limits the design to work only in a targeted environment and for a targeted application.

- Most designs focused on spectral and hyperspectral image processing and the data structures were tuned accordingly. These designs cannot be scaled/adapted for other applications, which have data structures with different dimensions.

In the following chapters, we present our hardware architecture for the K-Means clustering algorithm. We also demonstrate how we addressed the above issues in our research.
CHAPTER 3

DESIGN APPROACH AND DEVELOPMENT PLATFORM

In this chapter, we present our design approach and the development platform we used to design, develop, and implement the K-Means clustering algorithm. A hierarchical platform-based design approach was employed over a stand-alone component-based design, since the former facilitates design re-usability and a smaller footprint. An FPGA-based development platform was used to implement both our hardware and software designs.

3.1 Experimental Platform

We used the Xilinx ML605 development platform [23] to carry out all our experiments, which includes development, implementation, and evaluation of our hardware and software designs for the K-Means clustering algorithm. This platform uses a Xilinx Virtex-6 XC6VLX240T-1FFG1156 FPGA [24], which consists of 240K logic gates, 768 XtremeDSP™ Slices, 15MB of Block RAMS, and 3.6Kb of distributed RAMs for developing any complex circuitry. The development board consists of 512MB DDR3-SDRAM external memory, which can be scaled up to 2GB, to hold large volume of data. The ML605 board also has various off-chip non-volatile memories including: 128MB of Platform Flash XL, 32MB BPI Linear Flash, and 2GB Compact Flash, to hold the configuration bitstreams, which is a binary file that sets all the FPGA’s programmable bit locations to configure the logic and routing resources appropriately [25]. Furthermore, the board provides a Universal Asynchronous Receiver/Transmitter (UART) and a Joint Test Access Group (JTAG) port to connect to a host processor via Universal Serial Bus (USB). The FPGA was
programmed via a host processor using the Xilinx Platform Studio (XPS) and the Xilinx Software Development Kit (SDK). Additional user desired features can be added to the ML605 board through daughter cards attached to the two onboard high-speed FPGA Mezzanine Connector (FMC) expansion connectors: a high pin count and a low pin count.

The development platform also consists of MicroBlaze soft processors, which are built using the FPGA’s general-purpose logic. Unlike the hard processors, the soft processor must be synthesized and fit into the available gate arrays. The MicroBlaze processor uses the Advanced Extensible Interface (AXI) bus to communicate with other peripherals in the system [26]. The user-defined peripherals integrated through AXI interface can be connected to any processor compatible with AXI infrastructure, thus allowing the peripherals to be re-used in similar systems [27]. In this case, the processor controls the peripherals as memory-mapped devices.

As depicted in Figure 3-2, using the Xilinx Platform Studio (XPS), we integrated the MicroBlaze processor, our custom-based design for K-Means clustering and other Xilinx IP cores for UART, DDR3 controller via AXI interface. XPS automatically assigns the address space for each peripheral in the system-level design and maps it to the processor addressable range.

Our customized hardware architecture is designed using Verilog, whereas the Xilinx IP cores are implemented using VHDL. Our hardware experiments are performed using the XPS. Multiple synchronous clock domains are utilized: 200 MHz clock for the DDR3-SDRAM and the memory controller logic; 100 MHz clock for the on-chip peripherals and the MicroBlaze processor.
Our software design is written in C and executed on the MicroBlaze processor (using the SDK) in order to make a fair comparison with our hardware design implemented on the same development platform. Software design on MicroBlaze is compiled with level 2 optimization. Level 2 was selected over other optimization levels as it activates nearly all optimizations that do not involve a speed-space tradeoff.

We used UART to send and store the benchmark dataset to the DDR3-SDRAM via MicroBlaze processor. In this case, the MicroBlaze collects the data from the UART and writes it to the external memory. Experiments are performed on the benchmark dataset to evaluate our hardware design over software designs for the K-Means clustering.

To evaluate the performance gain or speedup resulting from the use of hardware over software, we measured the execution time in processor clock cycles for each design for K-Means clustering, and computed the gain using the equation (5) below. The initial time taken to store the benchmark dataset in the DDR3-SDRAM is excluded from both the software and hardware time.

\[
\text{Performance Gain} = \frac{\text{Clock Cycles taken by Software Implementation}}{\text{Clock Cycles taken by Hardware Implementation}} \quad \ldots (5)
\]

### 3.2 Hierarchical Platform-based Design Approach

Both our hardware and software designs are implemented using a hierarchical platform-based design approach, which facilitates component reuse at various levels of abstraction. As shown in Figure 3-1, higher-level functions utilize lower-level sub-functions. This approach
greatly simplifies the hardware design and development, since it leads to a high degree of
design re-use [28] as well as modular-based designs.

The fundamental operators including Subtract, Multiply, Add, and Divide are the lowest
levels of the platform-based design hierarchy. The Distance Measure and the Centroid
Evaluation operations are implemented using these fundamental operators. The top-level K-
Means clustering is developed by iteratively using the Distance Measure and the Centroid
Evaluation operations. For the hardware design, the Multiplier [29] and Divider [30] are selected
from the Xilinx IP core library, while all the other functions and operators are designed using
Verilog and developed on the FPGA.

For the software design, the Microblaze processor is configured with an arithmetic unit
capable of performing the lowest level operations. The higher level functions are developed in C
and executed on the MicroBlaze.
3.3 System-Level Interface

Figure 3-2 depicts the system-level interfacing for our hardware and software designs for the K-Means clustering. Our hardware design is triggered by the MicroBlaze, whereas the software design is executed on the MicroBlaze Processor.

The system-level design is created using the XPS automated flow [31], which connects the necessary Xilinx as well as user-defined IP cores. The XPS generates wrappers for the user-defined IPs for seamless integration to the system.

Our hardware design is developed using Verilog HDL, and the rest of the system-level design is developed using VHDL using the XPS. Detailed descriptions of our hardware and software architectures are presented in Chapter 4. The following sections give a brief description of the Xilinx IP cores used in the system.
3.3.1 MicroBlaze Soft Processor Core

Microblaze acts as the primary host processor that controls the hardware designed for the K-Means clustering algorithm. In addition, the software design for the K-means clustering is executed on this processor. The processor is configured to run at 100 MHz clock. A basic floating-point unit is enabled to perform accurate division and multiplication operations for the K-Means clustering software design. The Instruction and Data caches for the processor are set at 128kB.

3.3.2 DDR3 Controller

The DDR3 Controller module [39] provides an address-mapped AXI-based interface to access the SODIMM-based DDR3-SDRAM external memory. The DDR3 controller has two AXI masters: Microblaze processor and the K-Means Top hardware module. The controller directly connects to the DDR3-SDRAM pins via the FPGA I/O pins. The DDR3 controller works at 200 MHz.

3.3.3 DDR3-SDRAM External Memory

The ML605 board provides a physical 512MB Dual In-line Memory Module (DIMM)-based DDR3-SDRAM for data storage. The DDR3-SDRAM works at 200MHz giving an effective memory access rate of 400 MHz. For our designs, we partitioned the memory into two regions: the vector (D) region and the Cluster center (K) region. The processor is capable of mapping these regions to non-overlapping memory locations via register programming. The processor initializes the cluster region by randomly selecting a required number of vectors from the data region. The final cluster centers are updated in the cluster center region after the completion of
the clustering operation. Figure 3-3 shows the organization of the DDR3-SDRAM for our experiments.

![Figure 3-3: DDR3-SDRAM Organization](image)

### 3.3.4 AXI Interconnect

AXI interface connects the AXI-based masters and slaves in the system. It consists of two instances [32] – the AXI and the AXI-Lite.

- The AXI-Lite protocol is a subset of the AXI protocol with reduced signals and single beat transfers [32]. The AXI-Lite connects the AXI-Lite master, in this case, the MicroBlaze processor, with AXI-Lite slaves, including the UART and the K-Means hardware register interface. The MicroBlaze communicates with the slaves via address regions dedicated for each slave.

- The AXI instance connects one slave, which is DDR3 Controller, to two masters – the MicroBlaze and the K-Means Top Custom IP. The AXI interconnect matches the DDR3 Controller’s frequency of 200 MHz with the Master’s frequency of 100 MHz.
3.3.5 UART Lite

Using RS232-based UART Lite [33], the MicroBlaze processor can communicate with the host computer via a Universal Serial Bus (USB) connection. The UART is configured for a baud rate of 9600 with 1 stop bit and no parity bit. For our experiments, the benchmark dataset is sent from the host computer to the DDR3-SDRAM via MicroBlaze using the UART. Furthermore, display messages for debugging purposes and the execution time details are sent from the software program running on MicroBlaze. The final K-Means clustering results can also be transmitted back to the host computer via this interface.

3.3.6 IP Version and Memory Map

Table 3-1 provides the Xilinx IP core versions and the corresponding memory-mapped addresses. The address map is generated automatically by the XPS.

<table>
<thead>
<tr>
<th>S.No.</th>
<th>IP Name</th>
<th>IP Type</th>
<th>Version</th>
<th>BaseAddr</th>
<th>HighAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>microblaze</td>
<td>MicroBlaze Processor soft-core</td>
<td>8.50.c</td>
<td>0x0000_0000</td>
<td>0x0000_1FFF</td>
</tr>
<tr>
<td>2</td>
<td>axi_v6_ddrx</td>
<td>DDR3 Controller</td>
<td>1.06.a</td>
<td>0xA400_0000</td>
<td>0xA7FF_FFFF</td>
</tr>
<tr>
<td>3</td>
<td>axi_uartlite</td>
<td>RS232 UART Controller</td>
<td>1.02.a</td>
<td>0x4060_0000</td>
<td>0x4060_FFFF</td>
</tr>
</tbody>
</table>

3.4 Benchmark Dataset

During the initial phase of our hardware and software designs, a small synthetic dataset was used to evaluate the correctness of the K-Means clustering algorithm. The synthetic dataset consisted of ten vectors, each having two attributes. We considered four clusters. The final
results were manually evaluated and matched with the experimental results of the hardware and software designs.

Since it is imperative to verify that a large number of real data can be processed efficiently, we decided to use a real benchmark dataset, particularly designed for clustering. After investigating several database archives, we decided on a benchmark dataset for Wholesale customer Data [34] from the UCI Machine Learning Repository, for all our experiments. This dataset has 440 records (or vectors) of integer data, each having 8 attributes. The dataset represents the clients of a wholesale distributor. It provides the annual customers’ spending on six different product categories from two different channels. We performed experiments on varying sizes of this dataset with a varying number of clusters. Our experimental results and analysis are presented in section 5.1.
In this chapter, we present our proposed hardware architecture for the K-Means clustering algorithm. Our hardware architecture is designed, developed, and implemented on the FPGA. In order to evaluate our hardware design, software architecture is also designed for the K-Means clustering and executed on the MicroBlaze processor. Execution time for both the hardware and software designs is measured and the speed-performance is evaluated. These experimental results and analysis are presented in Chapter 5.

4.1 Embedded Hardware Architecture

During the design and development, we partitioned our hardware architecture into two main modules – a K-Means Top module and a Register module. Both the modules are designed in Verilog using Xilinx ISE 14.7 [35]. We used Xilinx IP cores for the division and multiplication operators and for the Block Random Access Memories (BRAM).

4.1.1 K-Means Top Module

Figure 4-1 shows our hardware architecture for the K-Means Top module. A wrapper for the hardware design is generated through XPS for ease of integration at the system-level. Our hardware design interfaces to the AXI-4 interconnect. The wrapper consists of an AXI Master interface that accepts commands over an Intellectual Property Interconnect (IPIC) and converts them to AXI protocol. The hardware design reads/writes data/results to the DDR3-SDRAM via
the AXI interface. The addresses to access the DDR3-SDRAM and the amount of data transferred are provided by the Register interface using sideband control signals.

![Block Diagram for K-Means Top Module](image)

**Figure 4-1: Block Diagram for K-Means Top Module**

The Data Engine module (Figure 4-1) controls the IPIC interface and is responsible for all the data transfers between the hardware design and the DDR3-SDRAM. The data fetched from the external memory is aligned with the internal data structure as in Figure 4-4, and buffered to the internal pre-fetch memory.

Firstly, the initial cluster centers are fetched from the cluster region of the DDR3-SDRAM, and stored in the internal Cluster Memory. Secondly, the top-level state machine then triggers the Distance Measure state machine or the Centroid state machine to perform the various stages of the K-Means clustering. After each iteration, the interim cluster centers are evaluated and stored in the internal Cluster Memory. The iterations complete when one of the following conditions is met: when a maximum number of iterations is reached; or when the
vectors cease to move the clusters. Once the iterations complete, the cluster centers stored in
the internal Cluster Memory are written to the cluster region within the DDR3-SDRAM.

After the hardware finish executing the clustering process, each vector in the DDR3-
SDRAM, will have an offset pointer to the cluster center, which it belongs to. Similarly, each
cluster center in the memory has a field indicating the number of vectors associated with that
particular cluster center. The final clustering details can be used by the subsequent tasks of the
Data Mining operations.

In the next sections, we present the data path and control path for our hardware
architecture from top-down, based on our hierarchical platform-based design approach from
Section 3.2.

4.1.2 Top Level State Machine

As illustrated in Figure 4-2, our hardware design is controlled by a top-level state
machine that triggers the lower-level state machines based on the design flow.

Figure 4-2: Top level state machine
Each state performs the following operations:

1. **IDLE**
   In this state, our hardware design is idling and is waiting for a trigger signal from the processor in order to start the clustering operation.

2. **INIT_K**
   The transition to this state occurs once the processor triggers the hardware by setting the `StartClustering` bit in the Status Control register (as explained in page 46). In this state, the Data Engine module fetches the initial cluster centers from the Cluster center region of the DDR3-SDRAM (Figure 3-3). The Data Engine then aligns and stores all the cluster centers in the local Cluster memory.

3. **DIST_MEASURE**
   The next state transition occurs, when the internal Cluster Memory is initialized. In this state, the Distance Measure state machine computes the distance between each cluster center as well as the distance between each vector, and then associates the vectors with the nearest cluster center. This step is discussed in detail in section 4.1.6.

4. **EVAL_CENTROID**
   The transition to this state occurs after the Distance Measure operation is completed. In this state, the Centroid state machine computed the new cluster centers as a Mean of all the vectors belonging to a particular cluster. This step is detailed in section 4.1.7.

5. **UPDATE_K**
   The transition to this state occurs, when the exit parameters for the K-Means clustering are met (as mentioned in section 2.3). In this state, the Data Engine writes the final
computed cluster centers from the Cluster memory to the Cluster center region of the DDR3-SDRAM. Then it moves back to the IDLE state and flags the DoneClustering bit in the Status Control register. This indicates that our hardware design has completed clustering the dataset in the DDR3-SDRAM. The processor periodically polls this register bit and proceeds to subsequent operations when this bit is flagged.

4.1.3 Data Engine

The Data Engine state machine is responsible for all the data transfers to/from the DDR3-SDRAM, as requested the Distance Measure and the Centroid State Machines. Each vector occupies several 32-bit memory locations (one per each attribute/element) in the DDR3-SDRAM. The Data Engine collects all the attributes of a vector as a single multi-element array, and then stores it in the pre-fetch buffers. This enables the Distance Measure and Centroid state machines to processes an entire vector at a time, by exploiting the data parallelism using hardware. Also when transferring the results to the DDR3-SDRAM, the Data Engine receives the vectors as a single multi-element array from the pre-fetch buffers, and then extracts each attribute/element from the array and stores it in subsequent DDR3-SDRAM locations.

The DDR3-SDRAM is accessed through the IPIC interface. The AXI master in the top wrapper converts the read/write commands from IPIC to AXI-4 burst reads/writes. The burst transfers reduce the memory access latency between the FPGA and the DDR3-SDRAM compared to a single read/write. The number of vectors to be fetched in each transfer depends on the total number of vectors and the internal pre-fetch buffer size. For example, if the pre-fetch buffer can hold $n$ number of vectors, then the Data Engine will fetch $n$ number of vectors, one vector at a time. The Data Engine only starts fetching the data if there is enough space in the
pre-fetch buffer to hold the entire dataset, thus removing potential throttling of the AXI-bus and improving the speed-performance of the system.

![Data Engine state machines](image)

Figure 4-3: Data Engine state machines

As shown in Figure 4-3, Data Engine consists of three internal states machines: MULT, READ, and WRITE. The MULT computes the total size of data to fetch from the DDR3-SDRAM at the beginning of the clustering operation using equation (6).

\[
SizeD = NumD \times (NumAttr + 1) \times 4
\]

\[
SizeK = NumK \times (NumAttr + 1) \times 4
\] ... (6)

Where NumD is the number of vectors, NumK is the number of cluster centers, and NumAttr is the number of attributes/elements.

Both terms in equation (6) are computed using a single instance of the Xilinx multiplier IP. The individual read/write commands are further decomposed to match the internal pre-fetch buffer size.
The READ and the WRITE state machines, as the names imply, are used to trigger the reading/writing data/results to the DDR3-SDRAM respectively. These state machines are synchronized, thus only one state machine is allowed transferring data at a time via the IPIC bus due to certain limitations of this bus. The READ is given a higher priority than the WRITE. The READ sends read requests only when there is enough space in the pre-fetch buffer to store all the read data without any throttling, which ensures optimal use of the Data Engine hardware, leading to better throughput. The respective state machines go to idle states after processing all the vectors and cluster centers and also when the Data Engine flags a Done signal to the top-level state machine.

The WRITE state machine typically writes an entire vector to the DDR3-SDRAM, which is not necessary when writing results after the completion of the Distance Measure operation. In this operation, only the cluster center offset of a vector may change, and only that particular entry needs to be updated in the DDR3-SDRAM. This update can be done using multiple single-writes AXI transfers, however it introduces additional command overheads/latencies and also adds extra logic to compute the addresses for each command. These issues can be avoided, by writing an entire vector, although, it may impact the performance when the number of attributes in the dataset increases.

4.1.4 Data Pre-Fetch Memory

The pre-fetch memories are designed using Xilinx CoreGen as block memories (detailed in section 4.1.8.3). The Data Pre-Fetch Memory has two instances – one to buffer read data and another to buffer write data. Unlike the organization of the vectors/cluster centers in the DDR3-SDRAM, the pre-fetch memories are designed to be wide enough to hold an entire
vector/cluster center on the same row. Since the memory is on-chip, it is significantly faster than the off-chip DDR3-SDRAM. It can transfer an entire vector to the hardware modules at a time.

The width of each memory instance is determined by the total number of vectors, the total number of attributes per vector, and the number of bits of the attribute. The depth of each memory instance is determined by the number of vectors processed in parallel by the Distance Measure module. In this case, the pre-fetch memory can be completely off-loaded into the Distance Measure processing elements (PEs), allowing the Data Engine to fetch the next set of vectors while the Distance Measure state machine is processing the previous set of vectors. This further reduces the AXI/DDR3-SDRAM access latencies and speedup the hardware design.

Figure 4-4 shows the organization of a single row of data pre-fetch memory. When the number of vectors is higher than the number of clusters, the cluster center requires a larger data structure. Hence, the memory width is determined to hold the cluster center. From Table 4-5, the parameters for this memory are: width $K_W$ and depth $NUM_D_STAGES$.

### 4.1.5 Cluster Memory

The Cluster memory is designed using the Xilinx CoreGen block memory (detailed in section 4.1.8.3). The cluster centers of the dataset, which are accessed during both the distance
measure and the centroid computations, are stored in this memory. Memory access latency is significantly reduced by buffering all the cluster centers to an on-chip BRAM than storing these in the DDR3-SDRAM. Since the number of cluster centers are typically much less than the number of vectors, all the cluster centers can potentially be buffered to the hardware design.

Similar to the data pre-fetch memory, the width of this buffer also depends on the total number of vectors, the total number of attributes per vector, and the number bits of the attribute (Figure 4-5). However, the depth depends on the total number of cluster centers. The hardware design is initially customized for a maximum number of cluster centers using the NUM_K parameter (from Table 4-5). The hardware then performs clustering operation on a dataset for a number of clusters that is less than or equal to NUM_K.

![Cluster Memory Data structure](image)

**Figure 4-5: Cluster Memory Data structure**

The Cluster Memory data structure (Figure 4-5) requires higher number of bits than the cluster center data structure (Figure 4-4). This is mainly because the same memory is re-used to perform the Mean operation when computing the new cluster centers as centroids of the existing clusters. Therefore, the width of cluster memory needs to be wide enough to hold the intermediate data accumulated. From Table 4-5, the parameters for the cluster memory are: width K_MEM_W and depth NUM_K.
### 4.1.6 Distance Measure

The Distance Measure module performs the Distance computation between each cluster center and each vector. For each vector, this module finds the nearest cluster center by computing the Euclidian Distance between a vector and the existing cluster centers. It then associates the nearest cluster center with a corresponding vector by updating the cluster center offset of the vector field (as shown in Figure 4-4). This process continues until all the vectors are processed.

Figure 4-6 shows the block diagram of the Distance Measure module, and Figure 4-7 shows the flow diagram for the module. In order to find the nearest cluster center for a particular vector, the distance between that vector and all the cluster centers are measured. Multiple Distance computations are performed in parallel using multiple Distance Measure modules or PEs, thus increasing the speed-performance of this stage.

![Figure 4-6: Distance Measure Module](image-url)
A single Distance Measure PE (shown in Figure 4-6) performs the distance computation between a data vector and a cluster center. Once all the PEs are loaded with one vector each, the Distance Measure state machine sends all the cluster centers, one at a time, for distance measure computation. The number of Distance computations processed in parallel depends on the number of Distance Measure PEs implemented on chip. Since our hardware design is parameterized, the number of PEs can be determined during the implementation by configuring the NUM_D_STAGES parameter (Table 4-5). The Euclidian Distance is selected over other Distance Measures due to its accuracy [14].
The Euclidian Distance is computed using the following formula:

\[ d(p, q) = \sqrt{\sum_{k=1}^{n} |p_k - q_k|^2} \]

... (7)

Where \( n \) is the total number of attributes and \( p_k \) and \( q_k \) are the \( k^{th} \) attributes of the two vectors \( p \) and \( q \) being compared.

Our intention is to find the minimum distance; therefore resource intensive square-root function is replaced with a multiplication to minimize the on-chip hardware resources, yet retaining the accuracy of the computation. Accordingly the equation (7) is modified to:

\[ d(p, q)^2 = d(q, p)^2 = \sum_{k=1}^{n} |p_k - q_k|^2 \]

... (8)

From equation (8), each Distance Measure PE will require \( n \) multipliers to compute the square term of the difference in attributes of the vectors, if computed in parallel. However, with the increase in number of attributes, the number of multiplier instances increases beyond the available hardware resources on-chip. To overcome this issue, a single Distance Measure PE implements a single multiplier, computes the square term, one at a time, and accumulates the results. The final Distance Measure is the last squared term computed. Although this method impacts the execution time of the individual Distance Measure computation, it allows us to incorporate more parallel PEs, thus enhancing the overall speedup of the operation. The new final squared distance is compared with the former minimum squared distance. If the new squared distance is less than the former, the cluster center offset and the new minima are updated for that particular vector. This process continues until all the cluster centers are
processed. The vectors with the updated cluster center offset are then offloaded to the DDR3-SDRAM via the Data Engine as shown in Figure 4-6. Simultaneously, the next set of vectors is loaded into the Distance Measure PEs for the next round of computation. This operation continues until all the vectors are processed and assigned to the nearest cluster center.

### 4.1.7 Centroid Measure

The Centroid Measure module, in Figure 4-8, consists of a main control state machine, a summation unit, and multiple fixed-point divider units. At the beginning of the clustering operation, this module fetches the initial cluster centers from the DDR3-SDRAM and stores it in the on-chip cluster memory. During the clustering operation, it computes the new cluster centers as the centroid of the existing clusters. The cluster centers are then updated to the Cluster Memory. After completion, it writes the contents of the Cluster Memory to the cluster center region in the DDR3-SDRAM.

![Figure 4-8: Centroid Measure Module](image)

The Centroid State machine is shown in Figure 4-9. The states presented below detail each step of the Centroid Measure operation performed by the summation and divider units.
1. **IDLE**

   In this state, the Centroid Measure waits for a trigger from the top state machine.

2. **INIT_K**

   In this state, the Data Engine is triggered to fetch the initial cluster centers from the cluster center region of the DDR3-SDRAM and to store in the on-chip Cluster Memory. This state is executed only once at the beginning of the clustering operation.

3. **SUM_ATTR**

   In **SUM_ATTR**, the state machine parses all the vectors and sums the corresponding attributes for the vectors belonging to the same cluster center. Each vector has a field corresponding to the cluster center offset it belongs to (as in Figure 4-4). The state machine uses this information to fetch the corresponding cluster center from the Cluster Memory and sums the corresponding attributes. It also increments the number of vectors field and writes the results to the cluster memory. This operation continues until all the vectors are processed. The on-chip cluster memory is reused to hold the intermediate accumulation results, which saves space on chip to store all the accumulation results.
4. **MEAN_ATTR**

The transition to this state occurs when all the vectors are parsed and when the cluster memory holds the final accumulation results. In this state, the state machine computes the Mean of the summed attributes. Mean is computed by dividing all the attribute fields of the cluster memory data structure with the number of vectors field. This produces the revised cluster centers for the next iteration. This division operation is performed by the Xilinx Divider IP, which provides an AXI streaming interface [36]. The number of divider instances equals to the number of attributes, therefore the Mean computation for all the attributes belonging to a cluster center is processed in parallel. The AXI streaming interface ensures best throughput from the pipelined Divider IP. The final division result is in fixed-point format, which is rounded to the nearest integer value. The Divider IP is detailed in section 4.1.8.1.

4.1.8 **Xilinx LogiCORE CoreGen IPs**

In this section, we present three generic Xilinx IP cores used in our hardware architecture.

4.1.8.1 **Divider**

This module performs fixed-point division of two integers [30]. The final result is in the form of \(<\text{Integer}>,\text{Fraction}>. The core has a standard AXI-Stream interface that throttles the input data stream if the internal pipeline is full. The IP can be configured in terms of speed, area, and the division algorithm used. The configurations selected for our design are listed in Table 4-1.
We used the High-Radix division instead of Radix-2, since the former is much faster (offers more than 200% speedup) than the latter [30]. However, this divider module occupies more space on chip, although it uses Xilinx XtremeDSP™ slices to reduce the resource usage. The Dividend and the Divisor width depends on the individual attribute widths selected during the implementation. Since no native support is provided to round the results, we used the fractional part of the fixed-point result to implement rounding logic external to the IP. A range of pipelining options is provided to help the timing closure. Our hardware design requires a minimum of 12 pipeline stages for timing closure at 100 MHz on the FPGA platform.

### 4.1.8.2 Multiplier

This module performs integer multiplication [29]. As shown in Table 4-2, two different multiplier configurations are used in our hardware design. Parallel Multiplier type is used due to higher speed. The 16x24 multiplier is used in the Data Engine module, while the 16x16 multiplier is used in each PE of the Distance Measure module. This IP can also be configured to have varying number of pipelined stages for timing closure.
Table 4-2: Multiplier IP Configurations

<table>
<thead>
<tr>
<th>Configuration Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier Type</td>
<td>Parallel Multiplier</td>
</tr>
<tr>
<td>Multiplier Construction</td>
<td>Using MULTs, Speed Optimized</td>
</tr>
<tr>
<td>Port A Width</td>
<td>16</td>
</tr>
<tr>
<td>Port B Width</td>
<td>24, 16</td>
</tr>
<tr>
<td>Instance Location</td>
<td>Data Engine, Distance Measure</td>
</tr>
<tr>
<td>Pipeline Stages</td>
<td>1, 2</td>
</tr>
<tr>
<td>Latency</td>
<td>1 Cycle, 2 cycles</td>
</tr>
</tbody>
</table>

4.1.8.3 Block Memories

Table 4-3: Block Memory Configurations

<table>
<thead>
<tr>
<th>Configuration Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Type</td>
<td>Native</td>
</tr>
<tr>
<td>Memory Type</td>
<td>Simple Dual Port RAM, Common Clock</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Minimum Area</td>
</tr>
<tr>
<td>Width</td>
<td>K_W, K_MEM_W</td>
</tr>
<tr>
<td>Depth</td>
<td>NUM_D_STAGES, NUM_K</td>
</tr>
<tr>
<td>Instance Locations</td>
<td>Data Engine – Read Buffer, Data Engine – Write Buffer, Cluster Memory</td>
</tr>
<tr>
<td>Enables</td>
<td>Always Enabled</td>
</tr>
</tbody>
</table>

The Block Memory Generator core uses embedded Block Memory primitives in the Xilinx FPGAs to extend the functionality and capability of a single primitive to memories of arbitrary widths and depths [37]. Sophisticated techniques in the Block Memory Generator core produce optimized solutions to provide access to the memories for a wide range of
configurations. The block memories are used for the read and write pre-fetch memories and the Cluster Memory. The memory configurations are shown in Table 4-3. The width and depth of each memory are functions of hardware parameters (in Table 4-5).

4.1.8.4 Xilinx IP versions

Table 4-4 lists the versions of Xilinx IP cores used for our design.

<table>
<thead>
<tr>
<th>S.No.</th>
<th>IP Name</th>
<th>IP Type</th>
<th>Version</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>div</td>
<td>Divider</td>
<td>4.0</td>
<td>CoreGen</td>
</tr>
<tr>
<td>2</td>
<td>mult</td>
<td>Multiplier</td>
<td>11.2</td>
<td>CoreGen</td>
</tr>
<tr>
<td>3</td>
<td>block mem</td>
<td>Block Memories</td>
<td>7.3</td>
<td>CoreGen</td>
</tr>
</tbody>
</table>

4.1.9 Hardware Parameter List

Our hardware architecture is generic and can be applicable to any appropriate data mining application by changing various parameters listed in Table 4-5. Some of these parameters control the size of the dataset to be processed by specifying: the total number of vectors, the number of attributes per vector, and the total number of clusters. Speed-performance is also controlled by some parameters by defining the number of parallel PEs in the design.

These parameters also impact the memories used in the design. The dimensions of the two memories (i.e., Cluster Memory and Data Pre-Fetch Memory): width of K_MEM_W and depth of NUM_K; width of K_W and depth of NUM_D_STAGES respectively; are incorporated in to the memory wrappers of the corresponding memories.
Table 4-5: Design Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_ATTR</td>
<td>8</td>
<td>Maximum number of attributes per vector</td>
</tr>
<tr>
<td>ATTR_W</td>
<td>20</td>
<td>Size of each attribute in number of bits</td>
</tr>
<tr>
<td>NUM_K</td>
<td>64</td>
<td>Maximum number of clusters for the dataset</td>
</tr>
<tr>
<td>NUM_D</td>
<td>8192</td>
<td>Maximum number of vectors in the dataset</td>
</tr>
<tr>
<td>NUM_D_STAGES</td>
<td>8</td>
<td>Number of vectors computed in parallel for Distance Measure operation</td>
</tr>
<tr>
<td>K_W</td>
<td>Derived</td>
<td>Width of Pre-Fetch memory to hold the entire vector in one line of memory</td>
</tr>
<tr>
<td>K_MEM_W</td>
<td>Derived</td>
<td>Width of Cluster Memory to hold Mean value of all the vectors within a cluster</td>
</tr>
</tbody>
</table>

4.1.10 K-Means Register Interface Module

In this section, we present our K-Means Register Interface hardware module designed to control our K-Means Top hardware module presented in section 4.1.1.

This module consists of several registers, which can be used to provide an interface between the hardware and the MicroBlaze processor. As shown in Figure 4-10, this module has three major interfaces:

- AXI-Lite slave interface to program the registers
- K-Means Top status and control for the design shown in Figure 4-1
- LCD Display interface to control the LCD on the development platform
The AXI-Lite slave interface connects to the processor AXI-Lite bus, and the corresponding registers are mapped to the addressable range of the processor. The MicroBlaze processor can then trigger our hardware design through these registers. This module also interfaces with the off-chip LCD display, and has dedicated registers to hold the data to be displayed. The LCD display is used for debugging purposes in addition to the Terminal program via the UART.

As listed in Table 4-6, the module consists of 14 registers, with 32-bit entry per register. The Base Address for the register map starts at **0x4400_0000**, which is automatically assigned by the XPS during system integration. These registers are described in detail below.
Table 4-6: Hardware Register Map

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Lcd0</td>
<td>Holds character data for Lcd row0, word0</td>
</tr>
<tr>
<td>0x04</td>
<td>Lcd1</td>
<td>Holds character data for Lcd row0, word1</td>
</tr>
<tr>
<td>0x08</td>
<td>Lcd2</td>
<td>Holds character data for Lcd row0, word2</td>
</tr>
<tr>
<td>0x0C</td>
<td>Lcd3</td>
<td>Holds character data for Lcd row0, word2</td>
</tr>
<tr>
<td>0x10</td>
<td>Lcd4</td>
<td>Holds character data for Lcd row1, word0</td>
</tr>
<tr>
<td>0x14</td>
<td>Lcd5</td>
<td>Holds character data for Lcd row1, word1</td>
</tr>
<tr>
<td>0x18</td>
<td>Lcd6</td>
<td>Holds character data for Lcd row1, word2</td>
</tr>
<tr>
<td>0x1C</td>
<td>Lcd7</td>
<td>Holds character data for Lcd row1, word2</td>
</tr>
<tr>
<td>0x20</td>
<td>NumData</td>
<td>Number of vectors</td>
</tr>
<tr>
<td>0x24</td>
<td>NumK &amp; Attr</td>
<td>Number of cluster centers and attributes</td>
</tr>
<tr>
<td>0x28</td>
<td>StartAddrD</td>
<td>Starting address of Data region in DDR3-SDRAM</td>
</tr>
<tr>
<td>0x2C</td>
<td>StartAddrK</td>
<td>Starting address of Cluster region in DDR3-SDRAM</td>
</tr>
<tr>
<td>0x30</td>
<td>StatusControl</td>
<td>Bits of this register has control and status info</td>
</tr>
<tr>
<td>0x34</td>
<td>CycleCounter</td>
<td>Counts number of cycles after trigger</td>
</tr>
</tbody>
</table>

a. **Lcd0 – Lcd7, Offset 0x00 – 0x1C**

![Figure 4-11: Lcd Display Segment Map](image)

These registers hold the data to be written to the LCD display. Figure 4-11 shows a map of the LCD segment controlled by these registers. Each register can hold four ASCII characters. For our experiments, we used the LCD display to indicate when the board is ready to execute the clustering algorithm, and to verify whether the register interface is functional.
b. **NumData, Offset 0x20**

The total number of vectors is stored in this register. The programmed value of this register should be less than or equal to the NUM_D value used to implement our hardware design.

c. **NumK & Attr, Offset 0x24**

This register holds the total number of cluster centers (numK) and the number of attributes (numAttr). In this case also, the programmed values of this register should be less than or equal to the NUM_K and NUM_ATTR values respectively, that are used during the implementation.

```
+-------+-------+-------+
| numK  | Reserved | numAttr |
+-------+-------+-------+
```

Figure 4-12: NumK & Attr Register

d. **StartAddrD, Offset 0x28**

The starting address of the data region of the DDR3-SDRAM is stored in this register.

e. **StartAddrK, Offset 0x2C**

The starting address of the cluster region of the DDR3-SDRAM is stored in this register. The utilization of StartAddrD and StartAddrK are depicted in Figure 3-3.
f.  **StatusControl, Offset 0x30**

As shown in Figure 4-13, the status and control register consists of various signals to trigger our hardware design.

![Figure 4-13: Status and Control Register](image)

These status and control signals/bits are detailed in Table 4-7. This register is accessed by the processor via a read-modified-write operation. Also, after the clustering operation is triggered by setting the StartClustering bit, the processor should not modify the register contents until the DoneClustering bit is flagged.

**Table 4-7: Status and Control field descriptions**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Access</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:23</td>
<td>num_iter</td>
<td>RO</td>
<td>0x000</td>
<td>This field specifies the number of iterations taken for the last clustering operation. Valid when DoneClustering is set.</td>
</tr>
<tr>
<td>22:13</td>
<td>max_iter</td>
<td>RW</td>
<td>0x3FF</td>
<td>This field specifies the maximum iteration for the current clustering.</td>
</tr>
<tr>
<td>12:7</td>
<td>Reserved</td>
<td>RW</td>
<td>-</td>
<td>Unused</td>
</tr>
<tr>
<td>6</td>
<td>WrError</td>
<td>RO</td>
<td>0x0</td>
<td>This field indicates any DDR3-SDRAM write errors. Valid when DoneClustering is set.</td>
</tr>
<tr>
<td>5</td>
<td>RdError</td>
<td>RO</td>
<td>0x0</td>
<td>This field indicates any DDR3-SDRAM read errors.</td>
</tr>
<tr>
<td>Bits</td>
<td>Name</td>
<td>Access</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>--------------------</td>
<td>--------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>errors. Valid when DoneClustering is set.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CounterReset</td>
<td>RW</td>
<td>0x0</td>
<td>This field, when set, resets the counter. SW must clear this bit to enable the counter again.</td>
</tr>
<tr>
<td>3</td>
<td>CounterStop</td>
<td>RW</td>
<td>0x0</td>
<td>This field, when set, stops the counter. The HW automatically sets this bit when the DoneClustering is set.</td>
</tr>
<tr>
<td>2</td>
<td>CounterStart</td>
<td>RW</td>
<td>0x0</td>
<td>This field, when set, starts the counter. Once the counter starts, this field will have no impact on the counter.</td>
</tr>
<tr>
<td>1</td>
<td>DoneClustering</td>
<td>RO</td>
<td>0x0</td>
<td>This field is set by Hardware when it finishes the clustering operation.</td>
</tr>
<tr>
<td>0</td>
<td>StartClustering</td>
<td>RW</td>
<td>0x0</td>
<td>This field when set starts the Clustering operation. This field is cleared by HW as soon as the clustering operation starts.</td>
</tr>
</tbody>
</table>

**g. CycleCounter, Offset 0x34**

This register provides the number of clock cycles between the start and the stop triggers. We used a counter in both our hardware and software designs to measure the execution time (in number of clock cycles) for K-Means clustering operation. The counter starts if one of the StatusControl.CounterStart bit or the StatusControl.StartClustering bit is set. We stop the counter by setting one of the StatusControl.CounterStop bit or the StatusControl.DoneClustering bit. Before starting the clustering operation, the processor must reset the counter by initially setting and then resetting the StatusControl.CounterReset bit.
4.1.11 Block-level verification methodology

Figure 4-14 shows the simulation test environment of our design.

The verification environment is designed in Verilog and simulated on a mixed mode VCS simulator. A mixed mode simulator is used since most of the Xilinx IPs are designed in VHDL. The Scenario Generator module provides a technique to plug-and-play different combinations of number of vectors, number of attributes, and number of clusters. This module receives this information from an external dataset file, which also includes the benchmark data. The AXI-4 Slave Memory Bus Functional Model (BFM) mimics the DDR3-SDRAM accesses, whereas the AXI-Lite Master BFM mimics the processor interface. The Scenario Generator initializes the cluster centers in the cluster region of the memory with randomly selected vectors from the dataset. Then it programs the K-Means Register module with: the starting address of the vectors; the starting address of cluster center region; and the the number of vectors, attributes and, clusters
via the AXI Lite Master BFM. After setting up the memory and the registers, the Scenario Generator triggers the clustering operation by flagging the `StartClustering` bit. Next, it polls the `DoneClustering` bit until it is set.

The Verilog Model for K-Means performs the same clustering operation in parallel. This model receives a copy of the initial memory structure with all the vectors, and stores the final results in the same copy. Once our hardware design flags the `DoneClustering` bit, the memory from the AXI4 Slave BFM is compared with memory from the Verilog Model to determine PASS/FAIL criteria. Using this test environment, we can also perform memory comparison on a per iteration basis to pinpoint the exact iteration where an error occurred, which will help in debugging the errors.

Furthermore, we built a C-Model for the Clustering algorithm as an additional checkpoint to test our Verilog Model.

### 4.2 Embedded Software Architecture

In order to evaluate our hardware architecture, we designed and implemented embedded software architecture for the K-Means Clustering algorithm. Software K-Means is designed in C, compiled on Xilinx Software Development Kit (SDK) [38], and executed on the MicroBlaze processor. Similar to the hardware design, the software design also triggers the hardware counter to measure the execution time (in number of clock cycles) for the K-Means clustering operation.

#### 4.2.1 Software Design flow

The software design flow is illustrated in Figure 4-15.
For our experiments, the UART is used to communicate between the host processor/computer and the development platform. Initially, the number of vectors, the number of attributes, and the number of clusters, are sent from the host processor to the development board. The experiments are performed using various data sizes, various number of clusters, etc. The starting address of the data regions in the DDR3-SDRAM is pre-determined and fixed for all experiments. The starting address for the cluster region is estimated by the software, and is based on the number of vectors being processed during an experiment.

In real world scenario, the hardware designs for a data mining applications could be executed on an FPGA as a stand-alone unit. In this case, the data mining task prior to the clustering is responsible for populating the DDR3-SDRAM with the initial dataset.
4.2.2 HW/SW Task Flow

Figure 4-16 distinguishes the task flow between the hardware and software designs.

The hardware (HW) flow is used to initialize, trigger, and monitor the hardware design implemented in section 4.1, while the software (SW) flow is used to execute the software design on the processor as detailed in section 2.3. All the hardware and software modules, including the sub-modules/functions are designed using integer arithmetic, except for the Mean computation. A rounding logic is required to increase the accuracy of the Mean. Hence, the Mean computation is designed using: a fixed-point division for hardware, and a floating-point division for software. Experiments are performed on both our hardware and software designs and the execution time to process the K-Means Clustering for each design is measured using the hardware counter. These experimental results and analysis are discussed and presented in the following chapter.
CHAPTER 5

RESULTS AND ANALYSIS

In this chapter, we present our experimental results and analysis performed to evaluate the efficiency of our hardware architecture. The experiments were performed on different hardware configurations (consisting of varying number of parallel processing elements) and with varying data sizes. Each experiment consisted of two runs of K-means clustering: one using the hardware flow, and another using the software-only flow. The software design was compiled using level 2 optimization and executed on the MicroBlaze processor. Level 2 was selected over other optimization levels, since it is the standard optimization level, and activates nearly all optimizations that do not involve a speed-space tradeoff. The experiments were performed on the benchmark dataset presented in section 3.4. Our software designs were executed on MicroBlaze processor running at 100MHz, and our hardware designs were executed on the FPGA at the same clock frequency.

5.1 Execution time for Embedded HW versus Embedded SW

The hardware and software execution time obtained from our experiments are used to evaluate the speedup using the equation (5) from section 3.1. The experiments are performed on three different hardware configurations, which consist of 8, 16, and 32 parallel processing elements (PEs) respectively, for the Distance measure computation. The number of vectors, in
the Distance measure computation, is evaluated in parallel. In this case, the number of vectors is controlled by the NUM_D_STAGES parameter in Table 4-5.

The hardware and software execution time for varying cluster configurations is presented in Table 5-1, Table 5-2 and Table 5-3. We present the execution time in terms of the number of clock cycles, which is a standard unit, thus, can be used to estimate the speedup/time for any target platform.

Table 5-1: Execution time for Hardware and Software (8 Clusters)

<table>
<thead>
<tr>
<th>NumD</th>
<th>NumAttr</th>
<th>SW Time (Clk Cycles)</th>
<th>HW Time (Clk Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>8-PE</td>
</tr>
<tr>
<td>50</td>
<td>4</td>
<td>423652</td>
<td>11636</td>
</tr>
<tr>
<td>50</td>
<td>8</td>
<td>750739</td>
<td>19784</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>1876743</td>
<td>36383</td>
</tr>
<tr>
<td>100</td>
<td>8</td>
<td>2966573</td>
<td>51830</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
<td>2031558</td>
<td>46328</td>
</tr>
<tr>
<td>200</td>
<td>8</td>
<td>6329389</td>
<td>138920</td>
</tr>
<tr>
<td>300</td>
<td>4</td>
<td>10636029</td>
<td>268299</td>
</tr>
<tr>
<td>300</td>
<td>8</td>
<td>7912345</td>
<td>653155</td>
</tr>
<tr>
<td>400</td>
<td>4</td>
<td>8697183</td>
<td>228149</td>
</tr>
<tr>
<td>400</td>
<td>8</td>
<td>18274180</td>
<td>665986</td>
</tr>
<tr>
<td>440</td>
<td>4</td>
<td>9891726</td>
<td>263346</td>
</tr>
<tr>
<td>440</td>
<td>8</td>
<td>21398906</td>
<td>559347</td>
</tr>
</tbody>
</table>
Table 5-2: Execution time for Hardware and Software (16 Clusters)

<table>
<thead>
<tr>
<th>NumD</th>
<th>NumAttr</th>
<th>SW Time (Clk Cycles)</th>
<th>HW Time (Clk Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>8-PE</td>
</tr>
<tr>
<td>50</td>
<td>4</td>
<td>667054</td>
<td>15968</td>
</tr>
<tr>
<td>50</td>
<td>8</td>
<td>1398128</td>
<td>33105</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>1823361</td>
<td>29115</td>
</tr>
<tr>
<td>100</td>
<td>8</td>
<td>5592004</td>
<td>86596</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
<td>13382442</td>
<td>267130</td>
</tr>
<tr>
<td>200</td>
<td>8</td>
<td>28522720</td>
<td>562584</td>
</tr>
<tr>
<td>300</td>
<td>4</td>
<td>8814649</td>
<td>194038</td>
</tr>
<tr>
<td>300</td>
<td>8</td>
<td>23637511</td>
<td>523256</td>
</tr>
<tr>
<td>400</td>
<td>4</td>
<td>15032280</td>
<td>346461</td>
</tr>
<tr>
<td>400</td>
<td>8</td>
<td>33725938</td>
<td>787765</td>
</tr>
<tr>
<td>440</td>
<td>4</td>
<td>19695866</td>
<td>461076</td>
</tr>
<tr>
<td>440</td>
<td>8</td>
<td>33384583</td>
<td>793628</td>
</tr>
</tbody>
</table>

Table 5-3: Execution time for Hardware and Software (32 Clusters)

<table>
<thead>
<tr>
<th>NumD</th>
<th>NumAttr</th>
<th>SW Time (Clk Cycles)</th>
<th>HW Time (Clk Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>8-PE</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
<td>10530246</td>
<td>192999</td>
</tr>
<tr>
<td>200</td>
<td>8</td>
<td>23745893</td>
<td>447387</td>
</tr>
<tr>
<td>300</td>
<td>4</td>
<td>14294466</td>
<td>291106</td>
</tr>
<tr>
<td>300</td>
<td>8</td>
<td>30084476</td>
<td>630838</td>
</tr>
<tr>
<td>400</td>
<td>4</td>
<td>36087909</td>
<td>765448</td>
</tr>
<tr>
<td>400</td>
<td>8</td>
<td>50973219</td>
<td>1126699</td>
</tr>
<tr>
<td>440</td>
<td>4</td>
<td>51687009</td>
<td>1121743</td>
</tr>
<tr>
<td>440</td>
<td>8</td>
<td>95190178</td>
<td>2150308</td>
</tr>
</tbody>
</table>
Figure 5-1 shows hardware speedup for 16 clusters and 8 attributes. The average speedup is 48 for the smallest hardware configuration that evaluates 8 vectors in parallel in the Distance measure computation, whereas average speedup increases to 115 with 32-PEs. It is observed that increasing the number of PEs does not proportionately reduce the execution time, hence the speedup. This is because only the Distance measure iteration of the algorithm is parallelized.

We expected an increase in speedup as the data size increases. However, this depends on the type of data, the initial cluster centers, and the number of iterations taken for the clustering. It should be noted that our software version of K-Means did not converge for datasets above 100 vectors. In this case, we only ran the software design for the same number of iterations as the hardware design (which converged), and obtained execution time accordingly, in order to expedite our experiments. As a result, there is a slight decrease in the speedup for a higher number of vectors as illustrated in Figure 5-1.
The Figure 5-2 depicts the hardware speedups for fixed number of vectors with varying number of clusters. This shows that our hardware design became more efficient in terms of speed-performance, as the number of clusters in a given dataset increases. For the smallest configuration with 8 PEs, we observed approximately a 300% increase in the speedup, when the number of clusters increased from 8 to 32. With 32 PEs, we saw approximately 658% increase in the speedup, for the same increment in the number of clusters.

5.2 Impact of Parallel Hardware on speedup

From the hardware design parameters given in Table 4-5, the NUM_D_STAGES controls the number of parallel PEs, while the others control the size of the dataset to process. Both the Figure 5-1 and Figure 5-2 show that by increasing the number of PEs to perform the distance measure computation in parallel significantly enhanced the speed-performance. This indeed demonstrates that parallelism in computation can be exploited to a great extent in hardware. Both graphs show that by using a higher number of PEs, the graph linearly shifts upwards.
showing a uniform scaling. For example, from the Figure 5-1, the average speedup increases from 48 (for 8-PEs design) to 115 (for 32-PEs design), leading to an improvement in a speedup of 140% compared to the 8-PEs. In addition, Figure 5-3 also shows that with the increase in the size of the dataset, the incremental rate of speedup also improves. Therefore, the parallel hardware becomes more efficient as the dataset size increases. This improvement of speedup comes at a cost of additional hardware resources. The trade-off between speed-performance and space (resource utilization) are discussed in the next section.

![Speedup Impact for different dataset sizes](image)

**Figure 5-3: HW Speedup with varying number of parallel PEs**

### 5.3 Resource Utilization

The main idea of parameterizing our hardware architecture is to provide an option to the designers to either build a faster hardware with higher resource utilization, or a slower hardware that has a comparatively smaller footprint. As depicted in Figure 5-3, by adding more parallel PEs, the speedup improves significantly. The number of PEs that fit into the hardware depends on the available gate count of the FPGA. The designers should carefully consider this
speed-space trade-off. In addition, incorporating parallel hardware depends on the target application. For example, image processing applications typically have to process a large volume of data, and in most cases the data needs to be processed in real-time. These applications will benefit from a design configured with the highest number of parallel PEs that will fit into the FPGA. For applications where speed is not critical, the designer can select a configuration with a lower number of parallel PEs. This will lead to a smaller footprint, which then will reduce the overall NRE cost.

In this section, resource utilizations for different hardware configurations are illustrated. The speedup versus resource utilization can be used to estimate a near optimum configuration for a specific application as well as for a specific reconfigurable hardware platform.

<table>
<thead>
<tr>
<th></th>
<th>Slice</th>
<th>Slice Utilization</th>
<th>Slice Registers</th>
<th>Slice LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-PE</td>
<td>8140</td>
<td>21%</td>
<td>16171</td>
<td>19366</td>
</tr>
<tr>
<td>16-PE</td>
<td>9820</td>
<td>26%</td>
<td>18650</td>
<td>23741</td>
</tr>
<tr>
<td>32-PE</td>
<td>12377</td>
<td>32%</td>
<td>23412</td>
<td>33963</td>
</tr>
</tbody>
</table>

Table 5-4 shows the overall resource utilization for the whole system-level design in Figure 3-2. The resource utilization is according to the Xilinx XC6VLX240T FPGA that has a total of 301440 flip-flops and 150720 LUTs distributed across 37680 Slices [24]. Our hardware configuration with highest number of PEs is 32, which occupies only 32% of the entire FPGA. This shows that there is still space to add more PE to our FPGA. Since the increase in resource utilization is due the number of parallel PEs, it only affects the footprint of our hardware design
for K-Means clustering, whereas the footprint for the rest of the system remains constant for all three configurations.

Table 5-5 presents the resource utilization for the K-Means Register Interface module in Figure 3-2, detailed in section 4.1.10. This module provides a mechanism for MicroBlaze processor to control our hardware architecture for K-Means Clustering. The resource utilization for this module is also fixed for all three hardware configurations.

Table 5-5: Resource utilization for K-Means Register Interface

<table>
<thead>
<tr>
<th>Slice Flip Flops</th>
<th>Slice LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>835</td>
<td>663</td>
</tr>
</tbody>
</table>

Table 5-6 presents the resource utilization for just our hardware design as in Figure 3-2 detailed in section 4.1.1. The three different configurations are based on the NUM_D_STAGES parameter, whereas other parameters were kept with default values as in Table 4-5.

Table 5-6: Resource utilization for the K-Means Hardware for three configurations

<table>
<thead>
<tr>
<th></th>
<th>Slice Flip Flops</th>
<th>Slice LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>8-PE</strong></td>
<td>5533</td>
<td>9534</td>
</tr>
<tr>
<td><strong>16-PE</strong></td>
<td>7876</td>
<td>15078</td>
</tr>
<tr>
<td><strong>32-PE</strong></td>
<td>12598</td>
<td>26719</td>
</tr>
</tbody>
</table>

It was observed that there was a significant increase in the LUT utilization compared to Flip-Flop utilization when the number of PEs was increased. This is because the on-chip memories were being implemented with LUTs. From Table 5-6, we can predict the hardware requirements for a design with varying number of parallel PEs. From Figure 5-1 and Figure 5-3,
we can predict the speedup for different hardware configurations with varying number of PEs. These details can be used to derive the optimal configuration for a given system.

Figure 5-4: Increase in Resource utilization Vs. Increase in Speedup

Figure 5-4 shows the percentage increase in speedup and percentage increase in resource utilization for different hardware configurations with varying number of PEs. We observed that our hardware design has a higher impact on speedup compared to the impact on resource utilization indicating that the design becomes more efficient as we add more PEs.
CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this concluding chapter, we summarize our contributions and discuss the directions for our future work.

6.1 Conclusions

From chapter 5, our experimental results clearly showed that our hardware design is significantly faster compared to the equivalent software design for the K-Means clustering algorithm. These experiments also illustrated that existing processor-based, software-only algorithms such as K-Means can be designed and implemented with higher precision in a relatively compact hardware. In [18], Manhattan Distance was used for Distance measure, and integer division was used for Mean computation, for the hardware designs. The Manhattan Distance does not provide an accurate Distance measure [14], and the integer division results in truncation of the Mean. These approximations lead to errors when forming the clusters. We addressed these issues by using the Euclidian Distance and a fixed-point division respectively for our hardware architecture. This also distinguishes our work from the existing research work on hardware support for K-Means clustering. In addition, the existing hardware architectures were designed for a specific application in mind, typically targeted to a specific development platform, and implemented with a proprietary interface. In contrast, our proposed hardware architecture is generic, can be designed and implemented on any development platform, and supports the industry standard AXI Interface. Our hardware design is parameterized as well as
scalable to process varying data sizes for different applications. By increasing the number of parallel processing elements, the speed-performance of our hardware design is further enhanced. Our hardware architecture can easily be configured for different applications with different performance constraints. Throughout this research work, we achieved all our research objectives stated in section 1.1.

6.2 Future Work

Our investigation made a substantial contribution by addressing some of the issues in big data analysis as well as some limitations of the existing research work. However, there is still scope for further improvements and enhancements as given below:

- Design the “initialization of Cluster centers” in hardware. This process involves random selection of vectors as initial cluster centers and takes up a significant amount of processor time.

- One of the major limitations of K-Means clustering is that it evaluates local minima based on the initial cluster centers. One way to resolve this is to run the clustering operation multiple times with different initial values, which is time-consuming if done in software and executed on a processor. We can modify our hardware design to automatically perform the clustering operation multiple times to derive the global minima.

- An alternative to the above two improvements is to design the K-Means++ [12] algorithm for cluster center initialization. As the algorithm involves multiple passes of the dataset, it is highly suited for hardware.
In order to reduce design-cycle time, we used the Xilinx IPIC interface which is then translated to AXI protocol by Xilinx IPs. Modifying our design to directly communicate via AXI can further reduce AXI and DDR3-SDRAM access latency by using the Read and Write channels simultaneously. Currently, IPIC can only perform one read/write operation at a time.

- Implement other clustering algorithms in hardware and provide a mechanism for dynamic reconfiguration of the FPGA to run the selected algorithm [28]. This will give more flexibility to the target application by allowing it to select the best hardware for a given dataset.

- We used UART to read data from the host computer and store the data in DDR3-SDRAM to test our architecture. A real system would have a much larger dataset stored in datacenter servers. Adding an Express Peripheral Component Interconnect (PCIe) interface to the design to store the data in the DDR3-SDRAM will allow our hardware to be integrated with the server compute farms as a PCIe daughter card. Data mining applications running on the servers will then be able to fully utilize our hardware via PCIe to perform necessary clustering operations.


[21] H. M. Hussain, K. Benkrid, H. Seker and A. T. Erdogan, "Highly Parameterized K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs," in Reconfigurable Computing and FPGAs (ReConFig), Cancum, 2011.


[34] M. Cardoso, Wholesale customer Data Set, University of California, Irvine.


