

FPGA implementation of a Multi-processor for Cluster Analysis

José Pedro André Canilho

Thesis to obtain the Master of Science Degree in
Electrical and Computer Engineering

Supervisor: Professor Doutor Horácio Cláudio de Campos Neto

Examination Committee

Chairperson: Professor Doutor Nuno Cavaco Gomes Horta
Supervisor: Professor Doutor Horácio Cláudio de Campos Neto
Member of the Committee: Professor Doutor Mário Pereira Véstias

November 2015

Acknowledgments

I would like to thank my supervisor, Professor Horácio Neto, for all the helpful guidance and insight throughout the development of this thesis.

I would also like to thank my fellow colleagues that I met throughout the course.

A huge thank you goes to my closest friends, some of which I had the fortune of meeting in IST, who were always there for me even when things got tough.

Lastly, a special thank you goes to my family, who gave me everything needed for me to succeed in these 5 years of college.

Abstract

Clustering remains one of the most fundamental tasks in exploratory data mining, and is applied in several scientific fields. With the emergent relevance of Big Data analysis and real-time clustering, the time taken by the conventional clustering algorithms becomes a main concern, as results are not computed in an acceptable amount of time. To overcome this issue, the scientific community is addressing the most widely used clustering algorithms and devising new ways to accelerate them. Approaches range from parallel and distributed computing, to GPU computing and custom hardware solutions using FPGA devices. The recent SoC devices integrate FPGA resources with GPPs and are very promising solutions for hardware/software co-design architectures.

In this dissertation, a hardware/software architecture is proposed to efficiently execute the widely known and commonly used K-means clustering algorithm. The architecture splits the algorithm's computational tasks by both hardware and software, with custom built hardware accelerators performing the most demanding computations. By doing so acceleration is achieved not only by performing the computationally demanding tasks faster but also by parallelizing different independent steps of the algorithm through both hardware and software domains. A prototype was designed and implemented on a Xilinx Zynq-7000 All Programmable SoC. The solution was evaluated using several relevant benchmarks and speed-ups over a software-only solution were measured. A maximum speed-up of 10.1 was observed using only 3 hardware processing elements. However, the system is fully scalable and therefore capable of achieving much higher speed-ups simply by increasing the number of processing elements.

Keywords

Clustering, K-means, Hardware/Software Co-design, Hardware Acceleration, Systems on Chip, Custom Hardware Design

Resumo

Clustering continua a ser uma das técnicas fundamentais em *Data Mining* exploratório, e é aplicada numa vasta gama de campos científicos. Com a relevância crescente da análise de *Big Data* e clustering em tempo real, o tempo de execução dos algoritmos convencionais começam a tornar-se uma preocupação, uma vez que os resultados não são conseguidos em tempo útil. Para ultrapassar este obstáculo, a comunidade científica está a abordar os algoritmos mais utilizados e a idealizar métodos para os acelerar. Técnicas abordadas passam por computação paralela e distribuída, aceleração por GPU e soluções em hardware personalizado, utilizando FPGAs. Os mais recentes SoC integram recursos FPGA e software, tornando-se soluções promissoras em arquitecturas hardware/software.

Nesta dissertação, uma arquitectura hardware/software, concebida para o algoritmo K-means, foi projectada e concretizada. A arquitectura divide as tarefas computacionais entre software e hardware, com as tarefas mais exigentes sendo tratadas por aceleradores hardware personalizados. Aceleração é conseguida não só por executar as tarefas exigentes mais rapidamente, mas também por paralelizar tarefas independentes entre os domínios hardware e software. A arquitectura foi implementada num dispositivo Xilinx Zynq-7000 All Programmable SoC. A solução foi testada utilizando vários *benchmarks* e speed-ups relativos a uma solução em software foram medidos. O máximo observado foi 10.1, usando 3 aceleradores. No entanto, o sistema é escalável e, portanto, capaz de conseguir speed-ups superiores com o aumento do número de aceleradores.

Palavras Chave

Clustering, K-means, Co-projecto Hardware/Software, Aceleração por Hardware, Systems on Chip, Projecto de Hardware Personalizado

Contents

1	Introduction	1
1.1	Clustering background	2
1.2	Motivation	5
1.3	Objectives	5
1.4	Main contributions	6
1.5	Dissertation outline	6
2	State-of-the-art	8
2.1	Parallel and Distributed computing approaches	10
2.2	GPU Computing	11
2.3	Custom Hardware Designs	13
2.4	Summary	20
3	Multi-Processor Architecture for Cluster Analysis	22
3.1	Target Device	23
3.2	AXI Protocol	25
3.2.1	Read transaction	26
3.2.2	Write transaction	27
3.2.3	AXI4 variants: Full, Lite and Stream	27
3.2.4	AXI Interconnect	28
3.3	Hardware/Software Architecture	29
3.4	Summary	33
4	Hardware & Software Components	34
4.1	DMA Block	35
4.2	AXI4-Stream Broadcaster	36
4.3	AXI4-Stream Interconnect	37
4.4	Hardware Accelerators	38
4.4.1	Memory and BRAM Controller	40
4.4.2	Invalidate Block	40
4.4.3	Floating-Point Cores	40
4.4.4	Minimum Distance Block	41

4.5	Tree Reduction Block	42
4.6	Software Component	43
4.7	Summary	45
5	Analysis and Experimental Results	49
5.1	Hardware Throughput	50
5.2	PS/PL Bandwidth	51
5.3	Hardware Resources	52
5.4	Execution time and Speed-ups	54
5.4.1	Analysis and predictions	54
5.4.2	Experimental Results	58
5.4.2.A	Varying number of centers	58
5.4.2.B	Varying data dimensionality	61
5.4.2.C	Varying number of datapoints	62
5.4.2.D	Peak Speed-up	64
5.5	Floating-Point Performance	64
5.6	Summary	67
6	Conclusions	69
6.1	Future work	71

List of Figures

1.1	Different cluster analysis results on the "mouse" dataset	5
2.1	Lavenier's SPA architecture for the assignment step	14
2.2	Gokhale's hardware/software final architecture	15
2.3	Continuous K-means computation time of both steps, for different subsets; datapoints = 100.000, centers = 4	16
2.4	KACU architecture	16
2.5	Architecture using floating-point division	17
2.6	Architecture for bioinformatic applications, featuring parallel center evaluation	18
2.7	Kutty's high speed configurable architecture	18
2.8	Asano's comparison between CPU, GPU and FPGA implementations	19
3.1	Zynq-7000 SoC main block diagram [Xild]	23
3.2	AXI read transaction [Xilb]	26
3.3	time diagram for AXI read transaction [Xilb]	26
3.4	AXI write transaction [Xilb]	27
3.5	time diagram for AXI write transaction [Xilb]	27
3.6	AXI4-Stream transaction [Xilb]	28
3.7	Top Level design architecture	30
4.1	Xilinx's AXI DMA core [Xila]	35
4.2	Block Diagram of a 3-master AXI4-Stream Broadcaster	37
4.3	Block Diagram of a 3-master AXI4-Stream Interconnect	38
4.4	Block Diagram of the Hardware Accelerator	39
4.5	Block Diagram of the Minimum Distance block	42
4.6	Block Diagram of the Tree Reduction block	44
5.1	Case 1 - Time diagram for a faster ARM execution	56
5.2	Case 2 - Time diagram for a faster accelerator execution	56
5.3	Execution time and speed-up results for the 1 st experiment	59
5.4	Execution time and speed-up results for the 2 nd experiment	60
5.5	Execution time and speed-up results for the 3 rd experiment	61
5.6	Execution time and speed-up results for the 4 th experiment	63

5.7	Performance values for the 1 st and 2 nd experiments	65
5.8	Performance values for the 3 rd experiment	66
5.9	Performance values for the 4 th experiment	66

List of Tables

3.1	Zynq-7000 Programmable logic resources	24
3.2	Task delegation between Hardware and Software domains	30
5.1	Resource usage for the DMA blocks	52
5.2	Resource usage for the AXI interconnect layer	52
5.3	Resource usage for the accelerator block	53
5.4	Resource usage for the Tree Reduction component	53
5.5	Total resource usage	54
5.6	Expressions for the computational time of each step	55

List of Algorithms

1.1	K-means pseudo-code.	4
4.1	master ARM pseudo-code	47
4.2	slave ARM pseudo-code	48

List of Acronyms

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APU	Application Processor Unit
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DBSCAN	Density-based Spatial Clustering of Applications with Noise
DDR	Double Data Rate
DMA	Direct Memory Access
DSP	Digital Signal Processing
EM	Expectation-Maximization
FIFO	First In First Out
FLOPS	Floating-Point Operations per Second
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
GIC	General Interrupt Controller
GPU	Graphics Processing Unit
IP	Intellectual Property
LUT	Look-up Table
MPI	Message Parsing Interface
OCM	On-Chip Memory

PL	Programmable Logic
PS	Processing System
SoC	System on Chip
SPA	Systolic Process Array
VHDL	VHSIC Hardware Description Language

1

Introduction

Contents

1.1	Clustering background	2
1.2	Motivation	5
1.3	Objectives	5
1.4	Main contributions	6
1.5	Dissertation outline	6

Cluster analysis, or clustering, consists in grouping a set of objects in such a way that objects which are similar to one another according to some metric belong in the same group (called a cluster). It is one of the most useful and used task of exploratory data mining, and it can be applied in a wide variety of fields. A few examples are machine learning, pattern recognition, image processing, information retrieval and bioinformatics.

Clustering is an unsupervised learning method. Unlike supervised learning, where a training set containing samples of the dataset and their correct classification is used to train a system, only the main unclassified dataset is used. After performing the clustering task over the dataset, the classification of each datapoint will correspond to the cluster it got placed into.

The process of clustering does not refer to one specific algorithm, but to a general task able to be solved by a variety of different algorithms with possibly different metrics and capable of producing completely different classifications. One of the main reasons for the large variety of clustering algorithms was the lack of a precise notion of a "cluster". A "cluster" simply stands for a group of data objects. This somewhat vague definition is prone to several interpretations and allows researchers to employ different cluster models. Some of the most frequently used models are:

- centroid models, where each cluster is represented by a mean vector
- distribution models, where clusters are modelled using statistical distributions
- density models, where clusters are defined as dense regions in space

Other known models, although less frequently used, are connectivity models (used, for example, in hierarchical clustering) and subspace models. Each different model may have features that affect the classification results of the dataset and help to choose which model is appropriate to each context.

Clustering algorithms rely on one of the several existent models. In particular, the K-means algorithm (which will be discussed thoroughly throughout this dissertation) uses a centroid model, which is a suitable model to a wide range of research fields. Any improvement made to the algorithm can, hence, be meaningful in many applications.

1.1 Clustering background

Herein we will unequivocally define the target clustering algorithm, in order to introduce how clustering will be performed and which tasks will be performed by each component of the architecture.

The chosen target algorithm was the K-means algorithm, which is one of the most simple algorithms capable of performing the clustering task. Despite its simplicity, it is still one of the most widely used clustering algorithms, due to its easy implementation and fast execution time. The algorithm uses a centroid model. It separates the data into a set of clusters, each one represented by the mean vector of all the datapoints within the class.

Each datapoint is classified into the cluster whose center is closest to it. The distance is usually judged using the euclidean distance as a metric, although some other types of metrics can be applied [ELTS01].

After an initial position is attributed to each center, the algorithm starts updating the position of each center in an iterative fashion. Each iteration is divided in two main steps:

- **Assignment step:** each datapoint is assigned to the nearest center, given the chosen distance metric
- **Update step:** after all the datapoints are assigned, the centers are re-calculated. The new positions correspond to the mean of all the datapoints within each cluster

Mathematically, the algorithm can be described as an optimization problem. Given the euclidean distance as a metric, and given a dataset $S = (x_1, x_2, \dots, x_N)$ and a number of clusters K previously defined ((c_1, c_2, \dots, c_K)), the following cost function C can be defined:

$$C = \sum_{i=1}^N \|x_i - c_{l(i)}\|^2 \quad (1.1)$$

where $l(i)$ stands for a function which returns the index of the cluster associated to datapoint i .

The purpose of the minimization problem is to find the centers (c_1, c_2, \dots, c_K) that minimize the cost function.

The algorithm converges when the positions of all the centers stay the same between iterations. As both computation steps minimize the sum of the square of the distances between each cluster, and the number of possible partitions of the dataset is finite, it is guaranteed that the algorithm will converge to a local minimum. It can't be assured that the global minimum will be found, though. The initialization of each center is the important factor that determines the overall quality of the classifications (given by the cost function) and the algorithm is often repeated with different initializations in order to find better cost minimums. The center initialization usually follows one of three methods:

- the values for each center are chosen randomly, using some pseudo-random generation function
- random datapoints from the dataset are chosen to be the initial centers (known as the *Forgy method*)
- each datapoint is classified into a random cluster and the mean for each cluster corresponds to the initial center (known as the *Random Partition method*)

The algorithm can be defined as an iterative process, given by the pseudo-code 1.1.

The overall complexity given to the standard K-means algorithm is $O(nkdi)$, with n being the number of datapoints, k being the number of clusters, d being the dimensionality of each datapoint and i being the number of iterations needed for convergence. Lines 10-22 represent the assignment step, where each datapoint gets classified and the class accumulators and counters get updated in each classification (so that the update step can be performed later). Lines 23-25 represent the update step, in which the mean of all the clusters are calculated and assigned as the new centers. The iteration gets repeated until the centers stop changing.

Clustering algorithms are often distinguished as being either *hard-bound* or *soft-bound*. In a *hard-bound* algorithm, each datapoint either belongs to a cluster or not. In a *soft-bound*, each datapoint belongs to each cluster to a certain degree. Although being a simple algorithm to conceive and implement, the K-means algorithm has a few limitations which make it incapable of classifying some types of datasets correctly. The K-means algorithm is an example of a *hard-bound* algorithm. The

Algorithm 1.1 K-means pseudo-code.

Input: dataset, dataset size N , number of clusters K , data dimensionality D **Output:** classifications, center coordinates

```
1: centerInitialization();
2: repeat
3:   for each center  $k$  do
4:     classAccumulator[k] = 0;
5:     classCounter[k] = 0;
6:   end for
7:   for each datapoint  $d$  do
8:     classifications[d] = -1;
9:   end for
10:  for each datapoint  $d$  do
11:    minDistance = Infinity;
12:    for each center  $k$  do
13:      currentDistance = distance( $d, k$ );
14:      if  $currentDistance < minDistance$  then
15:        minDistance = currentDistance;
16:        closestCenter =  $k$ ;
17:      end if
18:    end for
19:    classifications[d] = closestCenter;
20:    classAccumulator[closestCenter] +=  $d$ ;
21:    classCounter[closestCenter]++;
22:  end for
23:  for each center  $k$  do
24:    newCenter[k] = classAccumulator[k] / classCounter[k];
25:  end for
26: until (centers don't change)
```

Expectation-Maximization (EM) algorithm and the Density-based Spatial Clustering of Applications with Noise (DBSCAN) algorithm [EpKSX96] are examples of *soft-bound* algorithms. *Soft-bound* algorithms are usually more complex but have the advantage of being able to correctly classify datapoints belonging to overlapping clusters. One famous clustering example which illustrates the two types of boundaries is the "mouse" distribution (figure 1.1). In the "mouse" distribution, the dataset consists of points taken from three gaussian distributions: 2 distributions with small variance, and 1 distribution with large variance. A correct classification is achieved when the datapoints are matched to their corresponding distribution. A *hard-bound* like K-means will determine the center of each distribution correctly. However, since it relies exclusively on the distance between datapoints and centers, it won't be able to correctly classify the points from the distribution with larger variance that happen to be closer to the center of the other gaussians. *Soft-bound* algorithms such as the EM algorithm solve this issue by implying statistical distributions in the classification process.

Hard-bound algorithms such as K-means are also not resistant to *outliers*, which is the designation given to isolated datapoints, distant from the remaining ones. *Outliers* can be present in any dataset, most often due to measurement errors. Since any point is considered in the update step, any noise present in the dataset will influence the quality of the classifications. Algorithms like DBSCAN [EpKSX96] are able to ignore *outliers* that are far from the actual datapoints, by discarding them from the dataset.



Figure 1.1: Different cluster analysis results on the "mouse" dataset

1.2 Motivation

Clustering can be a rather lengthy process, when several iterations through the dataset are needed and when the dataset itself is large in both number of points and dimensionality. The increasing scientific interest in what is nowadays referred as *Big Data* analysis (which is a broad term to define very large datasets) emphasizes the need of producing clustering results of large datasets in an acceptable amount of time. Even the most simple and straightforward algorithms may sometimes take too long. This problem can be attenuated by the use of acceleration techniques, in order to speed up clustering algorithms.

The acceleration of algorithms has been a hot topic in the scientific community for quite some time and its importance in clustering algorithms is increasing dramatically with the equally increasing demand of classifying large datasets. Several acceleration techniques have been applied to clustering algorithms, from parallel and distributed computing to the use of Graphics Processing Units (GPUs).

Another different route is the use of custom hardware components capable of executing an algorithm (or part of it) very efficiently. The latest Field-Programmable Gate Array (FPGA) devices offer an incredibly large amount of programmable logic blocks, which allow the design of larger and more complex hardware blocks. Furthermore, FPGA resources are also being used in System on Chip (SoC) devices, incorporating both the flexibility of custom hardware design and the power of *hard-core* processors in one single chip. The SoC devices make it possible to produce hardware/software co-design architectures without the use of FPGA logic resources in *soft-core* processors.

1.3 Objectives

The main objective of this thesis was the research and development of a scalable hardware/software co-design architecture, capable of performing the K-means algorithm over any dataset, given its size, dimensionality and number of clusters. Once the design of the architecture was completed, the next goal was to implement the design on a Xilinx Zynq-7000 All Programmable SoC, taking advantage of some of the device's built-in components. Finally, in order to evaluate the solution's performance, extensive testing was performed, using several *benchmarks* and measuring several important metrics, such as execution time results.

1.4 Main contributions

In this dissertation a new efficient hardware/software co-design architecture for clustering datasets using the K-means algorithm is proposed. The architecture expands upon the previously devised solutions. It was designed with the Zynq-7000 SoC devices in mind and the complete architecture fits within the chip, thus removing the need of any hosts. In order to accelerate the execution of the K-means algorithm, the architecture makes use of hardware accelerators, as well as providing parallelism in two ways. Firstly, several hardware accelerators can be used to parallelize the most demanding computational task of the algorithm. And lastly, the architecture delegates computational tasks to both hardware and software domains, further exploiting the potential parallelism between tasks.

In addition to these features, the proposed architecture also offers the following advantages:

- it can handle datasets of different sizes and dimensions, without changing and re-arranging hardware components
- it is highly scalable: several hardware acceleration blocks can work in parallel with no impact on the hardware/software data bandwidth
- it is able to achieve good speed-ups, when compared to embedded ARM Central Processing Unit (CPU) implementations

1.5 Dissertation outline

This dissertation is organized as follows. Chapter 2 presents a summary of the current state-of-the-art related with the study of the K-means algorithm, with emphasis in previous solutions for the algorithm's acceleration. It starts by presenting several possible modifications to the standard K-means algorithm, that can both lead to smaller execution times and help to map the algorithm onto hardware architectures. Several variants of the K-means algorithm are mentioned at this stage. This chapter also presents several design solutions for accelerating K-means. These are divided into one of three different target platforms: parallel and distributed computing, GPU programming and custom hardware designs.

In chapter 3, the developed architecture is presented and explained. It starts by describing the target device used and mentioning its overall characteristics relevant to the architecture, followed by an overview of the main communication protocol between hardware/software instances and the specification of each hardware block designed.

In chapter 4, the developed architecture is then thoroughly explained, with each of the most meaningful components being detailed separately. In the end, a mention is made to the developed software programs that run alongside the hardware architecture, which further detail how the architecture functions as a complete hardware/software co-design solution.

In chapter 5, a theoretical analysis as well as the experimental results are reported. Firstly, a theoretical reasoning supported with both expected and real measurements is done, including some prediction of what should be the expected results in function of the number of processing elements in the system. The experimental results feature both execution time and performance analysis for several datasets of

different sizes and dimensions, as well as different number of centers. An analysis of the area occupied by the hardware architecture and other important features such as hardware throughput and bandwidth are also featured.

Chapter 6 presents the conclusions taken from this work, and possibilities for future evolutions for the work are mentioned.

2

State-of-the-art

Contents

2.1	Parallel and Distributed computing approaches	10
2.2	GPU Computing	11
2.3	Custom Hardware Designs	13
2.4	Summary	20

This chapter provides an overview of the most relevant work done regarding the acceleration of the K-means algorithm. The review of the evolution of the state-of-the-art revealed several different architectures that fit in one of three types of target platforms: acceleration using parallel and distributed computing, acceleration using GPU and acceleration using custom hardware (mainly through the use of FPGA devices).

Part of the research done was also focused on algorithmic transformations which can speed up the algorithm's execution and provide a better and more efficient mapping onto the hardware domain. The work published by V. Faber [Fab94] provides a slightly different approach to the standard K-means algorithm (called the *Continuous K-means algorithm*), claiming to achieve a much faster convergence, specially for large datasets. The modified algorithm starts by using the *Forgy method* for the initialization of the centers. If the dataset is large enough, the distribution of the initial centers should reflect the distribution of the entire dataset accurately. The second main difference lies in the length of each iteration. In the standard K-means algorithm, all the datapoints need to be evaluated. The *Continuous K-means algorithm* only evaluates a random sample of datapoints in each iteration. Once again, if the dataset is large enough and the random sample is enough to be representative of the entire dataset, the algorithm should converge more quickly than the standard K-means algorithm. Further testing was performed and it was stated that a sample of 10 to 15 percent of the entire dataset is enough for the algorithm to converge and the modified algorithm can be up to ten times faster than the standard K-means algorithm. However, there is no mention of comparisons between the quality of the results between the two algorithms, which should be an important measurement when reducing the sampling size of each iteration.

Some other algorithmic transformations are presented by Ordonez, C. [Ord03], for clustering binary streams of data. Three variants of the K-means algorithm are presented: *On-line K-means*, *Scalable K-means* and *Incremental K-means*. Although these approaches may be less suited for general purpose clustering, significant speed-ups can be observed, specially for the on-line and incremental variants (up to 6 times faster than the standard algorithm).

Another variant of the standard K-means algorithm is the *Progressive Greedy K-means algorithm* [WH07]. However, this algorithm sacrifices computation time in order to try and achieve lower cost function values, by doing some pre-processing of each point on whether or not it should move to another center.

Some other algorithmic transformations also facilitate the mapping of the algorithm onto custom hardware, by the use of different distance metrics [ELTS01]. Since the standard euclidean distance contains multiplications, which are harder to implement in hardware (specially when dealing with floating-point arithmetic), some architectures use the L_1 norm instead, also known as the *Manhattan norm*, which consists on the sum of the absolute value of the difference between the datapoint and the center (2.1).

$$\sum_{i=1}^D |x_i - c_{l(i)}| \quad (2.1)$$

The use of the L_1 norm eliminates the need for multiplications, whilst producing a small impact on

the quality results, when comparing to the results obtained through the use of the euclidean distance. However, for higher dimensionality data (10 dimensions and above), this impact is practically negligible [ELTS01]. The same can't be said for the L_∞ norm, which produces the worst results of all the comparisons made.

In sum, acceleration can be achieved by small and simple changes to the algorithm itself. In fact, some changes combined with developed solutions for any target platform can help to further improve eventual speed-up results. The use of the *Manhattan norm*, is particularly common in several hardware solutions targeting the K-means algorithm.

2.1 Parallel and Distributed computing approaches

Parallel and Distributed computing has provided many design solutions to speed up an extensive number of computer applications and is still nowadays an important and relevant approach worth considering. Parallel computing generally consists on several CPUs responsible for performing partitions of a large task, having some sort of shared memory available among all units. It is the core design of all multi-processing systems implemented today.

Each computational step of the K-means algorithm offers a high degree of parallelism, since the evaluation of one datapoint (or one center) doesn't depend on the evaluation of the remaining datapoints. The work done by T. Kucukyilmaz [Kuc14] provides a parallel implementation of the standard K-means algorithm for shared memory multi-processors, using the standard euclidean distance as the chosen metric. The developed implementation starts by parallelizing the initial center selection. Although this may have a low impact on the final speed-up, the overhead of doing so is practically inexistent, since the centers should be globally available to all processors in the first place. The assignment step, which is the most demanding step computation-wise, is parallelized between the processors: each processor is responsible for performing the assignment step on a subset of the dataset and storing the partial mean square error results. The results for all the datapoints are kept in the shared memory, so that the update step can be also parallelized between the processors. With a parallelized update step, each processor computes the new center of a subset of clusters. In the end, the master processor collects and accumulates the partial mean squared errors from all the slave processors, using a simple synchronization mechanism, such as a *mutex*.

The both the sequential implementation and the parallel solution were tested in a Intel Nehalem-EX Xeon 7550, with 8 cores running at 2.0 Ghz. The experimental results covered the evolution of the computation time for different dataset sizes, different number of clusters and different dimensionalities. Increases in the dataset size resulted in proportional increases on the computation time of both sequential and parallel implementations, although the parallel algorithm's execution time scales with 1/8 of the sequential version. However, for a small number of clusters and a small dimensionality, the sequential implementation is up to 4 times faster than the parallel implementation, due to the significant overhead of the creation of each thread.

As in parallel computing, distributed computing relies on several processing elements capable of work-

ing independently from one another. However, distributed computing covers architectures that may have each processing unit in separate machines. Unlike parallel architectures, distributed architectures feature slower inter-process communication, making it even more important to ensure decent load balancing and as few inter-process communication as possible. Distributed computing has also been applied to the K-means algorithm. Zhang, J. *et. al* [ZWH⁺11] implemented a distribution version of K-means (entitled *MK-means*), which uses Message Parsing Interface (MPI) as a programming model. The approach divides the dataset in several subsets and assigns them to each processor. In order to avoid inter-communication process in each iteration, all processors run the entire algorithm until convergence is achieved for their respective subset. When all processors finish, a merge of all the results is done and the final results are obtained. The results obtained in [ZWH⁺11] don't exactly reveal faster computation times, when compared to a sequential implementation, due to a still large communication overhead. However, the analysis of the computation time over increasingly dataset sizes predicts that *MK-means* enables improving of the computation time over large scale datasets.

Besides parallelizing several computation tasks over different processing units, another aspect to consider is the use of more efficient data structures, such as *KD-trees* [PDF09]. *KD-trees* can be used to perform an efficient search of the nearest neighbour in the assignment step. In the case of K-means algorithm, the KD-Tree is used to optimise the identification of the closest centroid for each datapoint. The experiments conducted by Pettinger, D. *et. al* [PDF09] reveal slightly better execution times for a small number of processors, but much worse execution time for a larger number of processors (32 and above).

2.2 GPU Computing

GPUs have recently been the subject of attention in research as an efficient co-processor for implementing many classes of highly parallel applications. The GPU's design is engineered for graphics applications, where many independent SIMD (*Single Instruction Multiple Data*) workloads are simultaneously dispatched to several processing elements. While parallelism has been explored in the context of traditional threads and SIMD processing elements, the principles involved in dividing the steps of a parallel algorithm for execution on GPU architectures remains a significant challenge.

Typically, most GPU-based implementations are unable to achieve their peak performance. The most common cause is the time spent in memory accesses, which can account for a significant portion of the execution time. Implementations limited mostly by memory accesses are referred to as memory-bound and, depending on the target application, the achieved performance values can be far inferior than the peak values.

Revealing to be a promising approach to significantly speed up lengthy and parallelizable algorithms, graphics processors are now widely used in several computer applications, including the K-means algorithm. [CMSS07] reports the solution designs for three known computer applications, one of which being the K-means algorithm. Although the bulk of the solution design isn't mentioned, the main bottleneck of GPU approaches is emphasized: memory accesses. The latest graphics processors are capable of achieving incredibly high peak performances (up to several TFLOPS). As a result, the actual peak performance

achieved when performing the algorithms drops as more memory accesses are needed. One important variable in estimating the memory access overhead is the available bandwidth. Recent implementations can achieve bandwidths of several GB per second [ZG09].

Che, S. *et. al* [CMSS07] adopt the Compute Unified Device Architecture (CUDA) programming model in their K-means implementation. CUDA provides a high-level approach to GPU programming with a seamless integration with the C programming language. Unfortunately, the CUDA version used at the time only allowed *synchronous Direct Memory Access (DMA)*. This means a *kernel* can't begin execution until the DMA transfer completes, which causes a large memory access overhead, specially for larger transfers. Similarly to the parallel and distributed approaches, the implemented parallel algorithm divides the dataset through the GPU thread blocks, where each thread is responsible for finding the nearest center of one data object. This way, the GPU becomes responsible for the assignment step of the algorithm, leaving the CPU only in charge of the center updates, after the use of a reduction function in order to gather the partial results from the GPU.

The solution was implemented in an NVIDIA GeForce 8800 GTX and speed-ups were measured against a CPU-only implementation running in a Pentium 4. The results reported only feature a comparison between the execution time of the distance calculation for all datapoints, which was accelerated up to 8 times, when compared to the CPU-only implementation. Datasets of different sizes were evaluated, but no mentions were made about the number of clusters used and about the data dimensionality.

In 2008, Farivar, R. *et. al* [FRCC08] presented a GPU based solution, also designed using CUDA. The solution parallelizes the algorithm in a similar way as in [CMSS07]: several threads are responsible for labelling part of the dataset, given the centers' positions. The CPU acts as the master thread and it is firstly responsible for uploading the dataset into the GPU. This transaction happens only once, since the datapoints remain the same throughout the algorithm's execution. The dataset is split among several threads (both the CPU and GPU threads), which perform the assignment step until convergence is achieved, followed by a reduction step and the center update done by the CPU, in order to obtain the final results. The communication cost introduced by the reduction step can be neglected for large datasets.

The experiments performed over this solution only featured one dataset: one million 1-D datapoints classified over 4 clusters. The target GPU was an NVIDIA GeForce 8600 GT, which has the peak performance values of 113 GFLOP/s and 22.4 GB/s of memory bandwidth. The maximum speed-up observed was roughly 13, when compared to a CPU-only implementation using a 2 Ghz host machine. However, the authors claimed that the current top of the line NVIDIA GeForce 8800 GT (with peak values of 336 GFLOP/s and 57.6 GB/s) could potentially achieve a speed-up of 68.

Following Farivar's work a year later, [ZG09] reports the accelerating approach followed by Zechner, M. *et. al*, which also relies on the CUDA programming model. The architecture once again delegates the distance calculation step to the GPU while the centroid updates are sequentially performed by the CPU.

Extensive testing was performed, using datasets of various sizes and dimensionalities, covering a wide range of clustering situations. The GPU used was an NVIDIA GeForce 9600 GT, which has a peak performance of 208 GFLOP/s and peak memory transfer bandwidth of 57.6 GB/s. However, the im-

plementation was memory-bounded, and the measured performance values are inferior. As a matter of fact, the observed bandwidth values ranged from 23 GB/s to 44 GB/s, and the computation performance suffered a significant drop: only 26 GFLOP/s to 36 GFLOP/s could be achieved. Nonetheless, a considerable speed-up could be observed, when compared to a CPU only implementation. For large dataset sizes and dimensionalities, the maximum speed-up observed in the experiments was 14, as the time spent labelling the datapoints grows significantly. These results end up contradicting Farivar’s expectations, however, for the use of more powerful GPUs and instead emphasize the communication bottleneck.

More recently, Kakooei, M. *et. al* [KS14] proposed a GPU implementation that parallelizes the initial center selection and applies a dynamic center correction throughout the algorithm’s execution, along with the usual assignment step parallelization. The results were measured in terms of both execution time and accuracy. Although the accuracy achieved with the dynamic center correction procedure is far superior (above 90%, when the standard K-means algorithm achieves roughly 78%), the extra processing time takes its toll on the execution time. As a result, only a speed-up of roughly 3.4 was observed, when compared to the base CPU-only implementation.

2.3 Custom Hardware Designs

Custom hardware design solutions are probably the most relatable and comparable solutions with the architecture designed in the scope of this dissertation. The prior solutions developed rely on the use of FPGAs, which provide a unique combination of highly parallel custom computation, relatively low manufacturing/engineering costs, and low power consumption requirements. Over the last two decades, these characteristics have proved to be a powerful combination for many scientific applications, including clustering algorithms.

Most of the solutions found rely on pure hardware implementations of the algorithm, where both steps of the K-means algorithm are mapped into distinguishable blocks of hardware, each one responsible for only one task. The recent introduction of SoC FPGAs allow the implementation of efficient hardware/-software co-design architectures within a single chip. With embedded *hard-core* processors, among other useful components, and efficient bridging between hardware and software domains, high performances can be achieved.

In 2000, Lavenier, D. [Lav00] presented an FPGA implementation of the algorithm, suited for the analysis of hyperspectral images, fed to the hardware via a stream of pixels. The architecture focuses on the assignment step only and assumes the existence of a processor responsible for uploading the data stream into the hardware on each iteration. The hardware consists of a Systolic Process Array (SPA) with a number of processing elements equal to the number of centers (2.1). Each processing element is responsible for computing the distance to each center, using the Manhattan norm as the distance metric. If the computed distance is less than the current minimum, then the information of the closest center is updated and then passed along the array. Otherwise, the information is kept untouched when it is passed along. Each processing element contains a logic block for the distance evaluation and a memory block which stores the current center.

The data transfer protocol between the host and the device wasn’t completely specified. Memory-

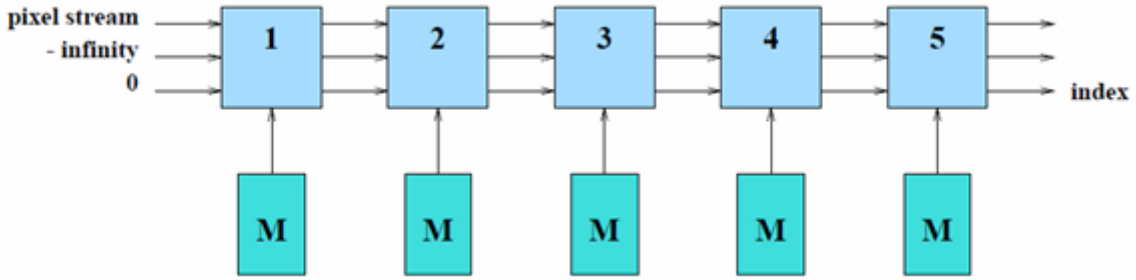


Figure 2.1: Lavenier’s SPA architecture for the assignment step

mapped and DMA transfers were compared: the DMA solution produced better speed-up results, as the transfer rate achieved was roughly 20 times higher than the transfer rate of the memory-mapped solution. The architecture was also tested in different boards, in which the largest one could hold up to 256 processing elements. The high degree of parallelism offered by the SPA along with the pipelined nature of the array resulted in increasing speed-ups for an increasing number of centers. The highest speed-up registered was of 336, for 256 centers, although there is no mention of the processor used for the CPU-only implementation.

Later, in 2003, Gokhale, M. *et. al* [GFM⁺03] presented a hardware architecture which computed the distance calculation of the datapoints to the center and delegated the center updates to a processor. The target platform featured an Altera [Alt] Excalibur FPGA device with two different processors: the NIOS 1.1 RISC *soft-core* and the ARM *hard-core*. Gokhale’s development followed an incremental methodology: firstly, only the plain hardware block responsible for the distances was considered. A simple hardware block that only computed the distance to the centers was designed and mapped onto the FPGA. The rest of the algorithm executed on software and the former distance calculation instructions are replaced by simple reads and writes onto the programmable logic, through the use of parallel I/O connections. This revealed to be an important step in the development, since it illustrated one of the main aspects when analysing the bottleneck of hardware/software designs: communication time. This solution was in fact slower than a full software implementation, because the numerous hardware reads and writes caused a very large communication overhead, that overwhelmed the actual hardware computation time.

Secondly, an SPA similar to the architecture proposed by Lavenier [Lav00] was designed. The processor would write the data stream onto the first element of the array and read the final result from the final element. This solution allowed the hardware not only to compute the distance of the datapoint to each center, but also to label each datapoint with the closest center. This solution reported a speed-up of 25 for the distance calculation task itself, and a total speed-up of 6 for the complete algorithm. Both *soft-core* and *hard-core* CPUs were tested. Although the *hard-core* CPU was able to operate at a higher frequency, the speed-up achieved using it was a bit inferior: a maximum speed-up of 3 was measured. This was due to the fact that the CPU-only implementation ends up being also faster.

Finally, dual port Blocks Random Access Memory (BRAMs) were included to do the communication between the processor and the user logic. A 32-bit wide BRAM allowed for the use of several SPAs, after

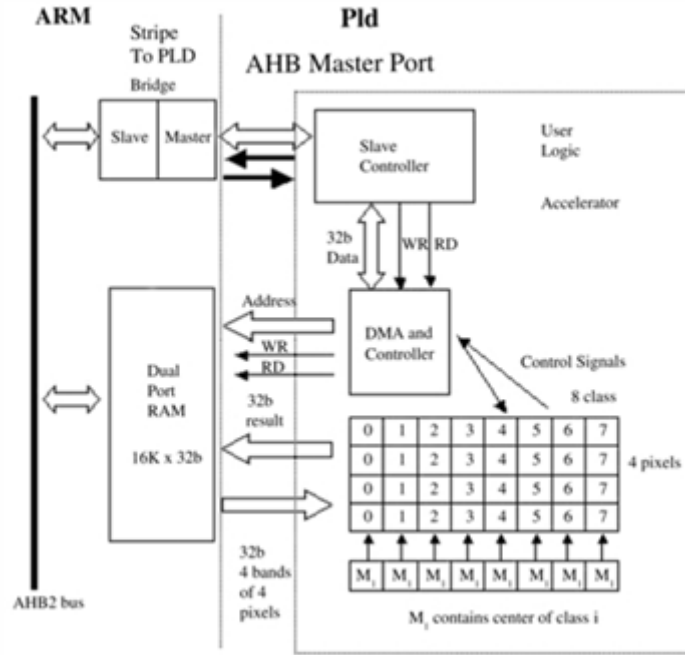


Figure 2.2: Gokhale's hardware/software final architecture

some initial configurations regarding the base addresses for the data in the memory. This architecture was only implemented using the ARM processor, which was connected to the programmable logic via an Advanced High-performance Bus (AHB) bus (2.2). As before, the hardware portion of the architecture performed both the distance calculation and the labelling. The total computation time was compared with a CPU-only implementation running at 1Ghz, and the speed-up achieved was 11.8.

Two years later, Wei-Chuan Liu *et. al* [LHC05] proposed a hardware architecture capable of performing the algorithm in its entirety. The framework was named *KACU*: K-means with hArdware Centroid Updating. The architecture and the overall work performed gave emphasis to the center updating step and the algorithm applied was the continuous K-means [Fab94] instead of the standard algorithm, since it provides more update steps, given the number of the datapoints evaluated per iteration. The center updating step may in fact start to be computationally more demanding, for small subsets, since they need to be performed more often, as shown in figure 2.3. The subset evaluated in each iteration was always $B = \sqrt{N}$, with N being the number of datapoints, as suggested in [Ord03]. The main architecture once again consisted of an SPA, with the addition of the accumulators needed to perform the mean of each center and the divider block, along with some extra control blocks, see figure 2.4. The new centers only replaced the old ones after the control unit signalled so.

The architecture was implemented in an Altera FPGA, using a NIOS *soft-core* processor executing at 50 Mhz for the control unit. Better speed-ups were achieved for small subsets, since the execution time of the update step becomes less negligible. A maximum of 5.6 was observed. One important aspect that was not mentioned, though, was the type of numerical representation used. This is an important factor, as it strongly affects complexity and performance of the hardware design.

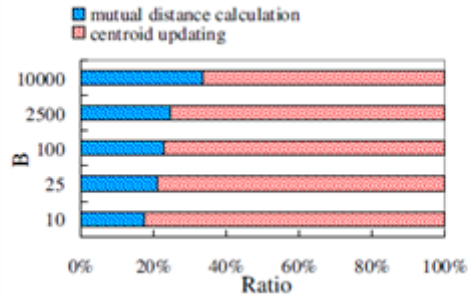


Figure 2.3: Continuous K-means computation time of both steps, for different subsets; datapoints = 100.000, centers = 4

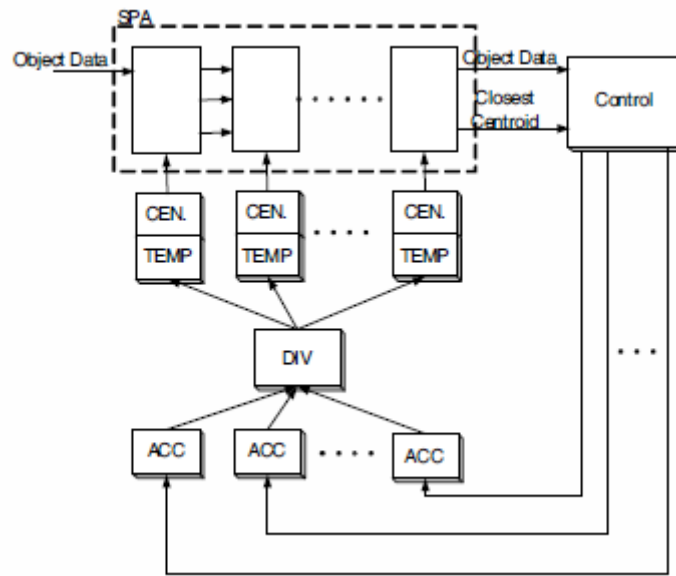


Figure 2.4: KACU architecture

Following the same line of thought, Wang, X. *et. al* [WL07] developed another custom hardware solution, targeted for the standard K-means algorithm applied in the analysis of multi-spectral images. This solution also delegated all the computational tasks to the custom hardware. The host processor was only used for the initial center selection and to upload the datapoints and the centers to the memory within the FPGA. Freeing the CPU completely from the computational tasks may be useful, specially in areas such as image processing where different analysis can be done in parallel. Fixed-point arithmetic was used for all the calculations, except for the division in the center updating step, where *fixed-to-float* converters were used at the inputs of the dividing block and a *float-to-fixed* converter was used at the output. The remainder of the datapath unit computed the Manhattan norm. The rest of the architecture included a synchronization block in order to sync the evaluated pixels with the accumulators' entries (see figure 2.5) and some residual control tasks regarding memory transfers. Since the architecture was conceived to target the analysis of multi-spectral images, the datapath unit may be replicated by the different channels of the multi-spectral image.

The architecture was tested on a Xilinx XC2V6000 FPGA. The operating clock frequency and the

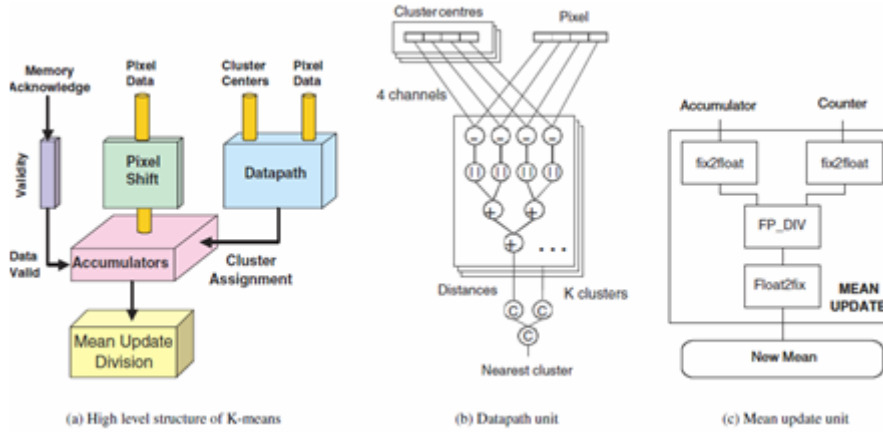


Figure 2.5: Architecture using floating-point division

number of clusters used, were not mentioned. However, some conclusions were presented on the observed bottlenecks. Once again, the data transfer of the target image onto the FPGA overwhelmed the computation time of the algorithm, specially when few iterations were needed to achieve convergence. For 50 iterations, a speed-up of 11 was observed, when comparing to a full CPU implementation running on an Intel Pentium 4 operating at 3.2 Ghz. Most of the clustering performed by K-means usually converge after a few dozen iterations. In order for the computation time on the hardware solution to be similar to the data transfer overhead, a large number of iterations would be needed. A speed-up of 173 was observed, for 1000 iterations, but this is a very untypical scenario.

Hussain, H. *et. al* [HBSE11] proposed a specific hardware implementation of the K-means algorithm targeting bioinformatic applications. The architecture used fixed-point arithmetic and some analysis was presented regarding the appropriate bit length of both the integer and fractional part. Although re-arranging the hardware for the optimal bit lengths for each dataset is not practical, since the hardware implementation is supposed to be used in a specific area of analysis, it is conceivable to choose a bit length appropriate for most cases. In order to find an appropriate length, the authors analysed several datasets and made their choice considering the datasets' range and precision.

The parallelization strategy followed in this work was different than the one used in the previous implementations discussed, not only from the ones using custom hardware design but also from the ones using GPUs and parallel/distributed computing. Instead of dividing the dataset through several blocks in the hardware and each block being responsible for computing the minimum distance of a subset of the datapoints from all centers, this architecture computed the distance of one datapoint to all centers in parallel. The distance calculation blocks were followed by a tree comparison of all of the centers, which delivered the minimum distance and the center ID to the appropriate accumulator. In the end, a dividing block computed the new centers, based on the accumulators and the counters for each center. The complete architecture is portrayed in figure 2.6 . The developed architecture was only capable of clustering over 8 centers, although it would be relatively easy to expand it for a larger number of centers.

The architecture was implemented in a Xilinx XC4VLX25-10SF363 FPGA and was able to execute

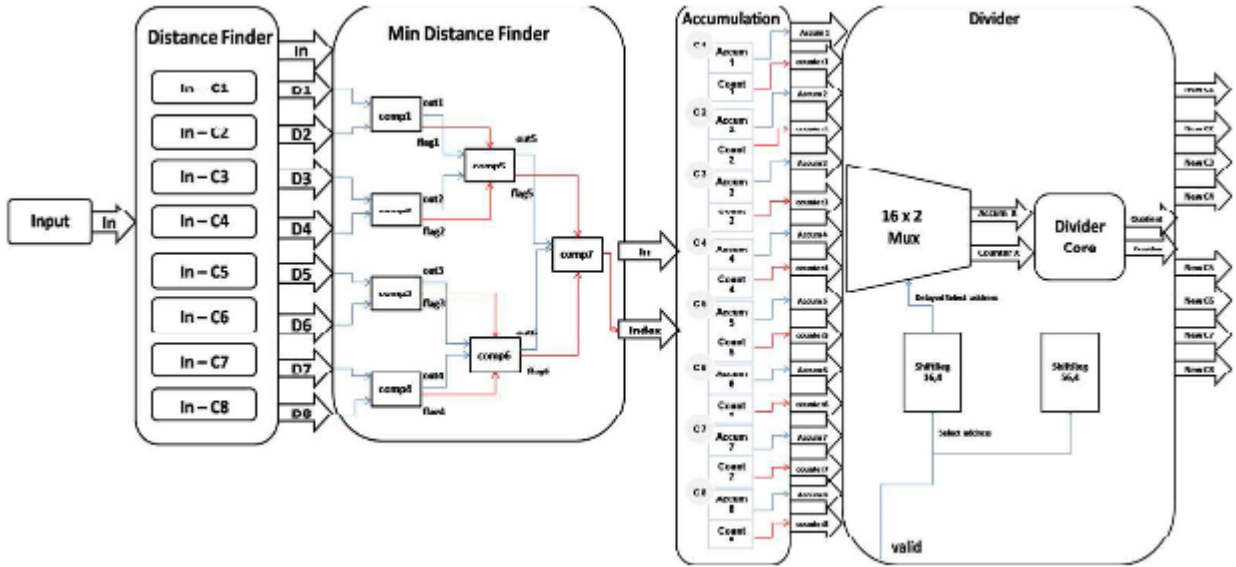


Figure 2.6: Architecture for bioinformatic applications, featuring parallel center evaluation

at 126 Mhz. A speed-up of 10.3 was observed, compared to a CPU-only implementation on an Intel Core2 Duo 3.0 GHz. One processing element only occupied 20% of the FPGA resources, which allowed to replicate it 5 times to evaluate 5 datapoints in parallel. With 5 processing elements, the speed-up observed was 51.7. This, once again, reflected the high degree of parallelism of the K-means algorithm, due to the independent analysis of the datapoints.

More recently, in 2013, Kutty, J. *et. al* [KBA13] presented a high speed configurable FPGA architecture. Similarly to the previous approach, the architecture split the distance calculation of a single datapoint through several distance calculation blocks, one per each center. These blocks computed the Manhattan distance and delivered the results to a tree comparison block. The main difference to Wang’s architecture is that a dividing block is used for each center. The dataset and the initial centers are stored *a priori* in the BRAMs within the FPGA. Figure 2.7 shows a block diagram of the architecture.

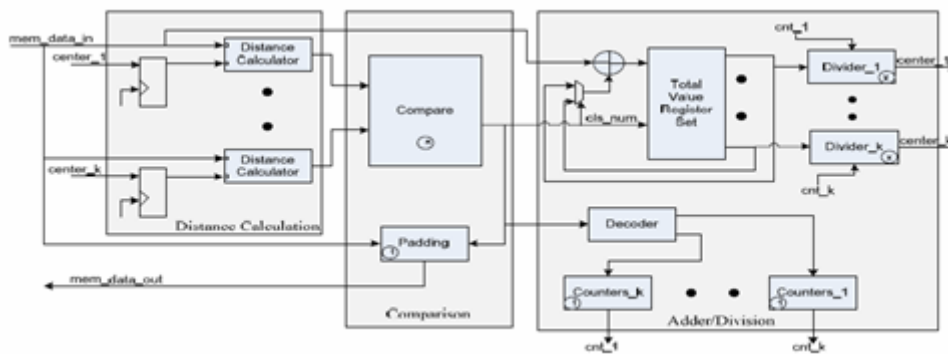


Figure 2.7: Kutty’s high speed configurable architecture

The architecture is highly pipelined and is able to produce a throughput of 1 datapoint classification per clock cycle. The architecture can be parametrized for a given word length of each datapoint, number of clusters and number of datapoints in the dataset. However, these parametrizations are given as VHIC

Hardware Description Language (VHDL) generics, which means that a different configuration requires a re-implementation of the architecture, with more or less logic hardware resources being used. Moreover, the architecture seems to be fixed to 2-dimensional datapoints and no mention is made to any variable dimensionality parameter. The numerical representation was not mentioned specifically, although it may be assumed that fixed-point arithmetic was used, given the variable word length parameter.

The architecture was implemented in a Xilinx Virtex-6 FPGA. A operating clock frequency of 400 Mhz was indicated for up to 9 clusters, being, to the best of my knowledge, the highest clock frequency achieved in any of the designs proposed thus far. The clock frequency degrades with the increase of number of clusters, although a frequency of 300 Mhz was still indicated for 32 clusters. The results focused on the occupied area of the architecture, rather than on the actual computation time, and no speed-ups were mentioned for the complete algorithm’s execution, unfortunately. The only inference that can be made regarding the computation time is the link between the clock frequency and the architecture’s throughput, which was claimed to be of 1 classification per clock cycle.

To conclude, we refer two studies that present comparative results of K-means algorithm implementations on FPGAs, CPUs and GPUs [AMY09, HBES11]. In 2009, Asano, S. *et. al* presented a comparison of FPGA, GPU and CPU implementations of image processing applications. Several important image processing tasks were studied, one of which being K-means clustering. The FPGA implementation followed the standard parallel approach used in all GPU implementations covered earlier, where several datapoints are evaluated in parallel. The standard euclidean distance was used instead of the Manhattan distance.

The CPU used was an Intel Core 2 Extreme QX6850 3GHz, the GPU used was a XFX GeForce 280 GTX with 933.12 peak GFLOPS (using CUDA version 2.1), and the FPGA design was implemented on Xilinx XC4VLX160. The observed results are portrayed in figure 2.8 (the implementation performance was measured in *frames per second*). The FPGA solution clearly outperformed the other 2 solutions, more markedly for the tests with a lower number of clusters.

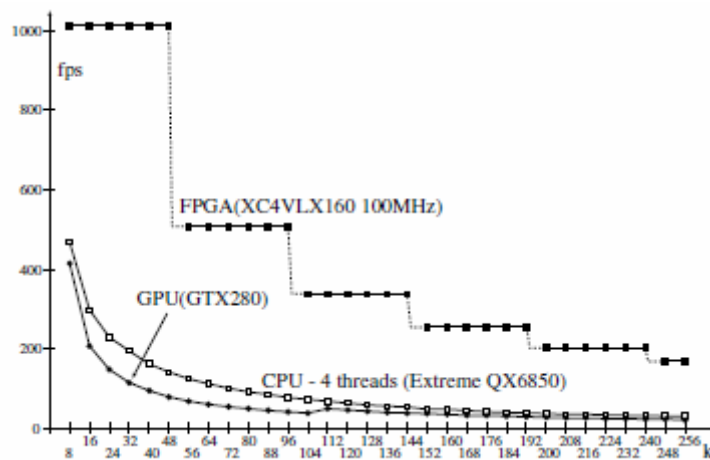


Figure 2.8: Asano’s comparison between CPU, GPU and FPGA implementations

In 2011, Hussain, H. *et. al* conducted the second study comparing CPU, GPU and FPGA imple-

mentations. The FPGA design follows Kutty’s architecture (2.7): every computational step is performed in the hardware (with one divider available for each cluster) as well as offering the same parametrizable variables. In fact, Hussain’s architecture also allows different dimensionalities, in contrast with Kutty’s architecture. A host is only needed to upload the dataset and the initial centers into the BRAMs available in the FPGA. The hardware implementation was compared with a GPU implementation running on an NVIDIA GeForce 9600M GT graphics card (with peak performance of 208 GFLOPS) and a CPU implementation running on Intel Core 2 Duo at 2.2 Ghz. The tests were performed using 3 and 4 clusters and datasets of 4 and 9 dimensions. The FPGA results using 4-dimensional datasets showed a speed-up of 26 when comparing with the CPU implementation and a speed-up of 3.2 when comparing with the GPU implementation. For 9-dimensional datasets, speed-ups of 54 and 6.7 were observed, over the CPU and GPU, respectively. A brief comparison regarding the energy efficiency of the 3 implementations, showed that the FPGA solution was 615 times more energy efficient than the CPU solution and 31 times more efficient than the GPU solution.

2.4 Summary

This chapter presented an overview of the main scientific contributions that covered the acceleration of the K-means clustering algorithm. It started by focusing on some algorithmic transformations that can potentially lead to a faster convergence and thus a reduced computation time. One important variant of the K-means algorithm, called the *Continuous K-means algorithm*, only analyses a portion of the dataset per iteration, leading to faster iterations while maintaining good quality results, if the dataset is large enough. Another very important transformation is the distance metric: many solutions implement the L_1 norm (also known as the Manhattan norm) instead of the regular euclidean norm, when evaluating the distances of each datapoint to the centers. By using the L_1 norm, the need of performing multiplications is eliminated. This proved to be particularly useful in the custom hardware implementations.

Following these algorithmic transformations, a few mentions were made to some parallel and distributed implementations of the K-means algorithm. [Kuc14] and [ZWH⁺11] presented solutions using parallel and distributed approaches, respectively, with relative success in terms of computation time and speed-ups compared to single threaded implementations. In these cases, and specially in the distributed computing approach, the communication overhead proved to be significant, despite the high degree of parallelism inherent to the algorithm itself.

Next, a number of solutions using GPU programming were mentioned. In all cases, the dataset was divided among the available threads, so that several datapoints could be compared to all centers in parallel. These implementations also offered significant improvements over CPU-only implementations. However, the communication overhead and the memory accesses once again proved to significantly affect the results. The GPUs are unable of performing at their peak performance levels due to the multiple memory accesses, specially for earlier versions of the CUDA programming model, which didn’t allow computation to start while DMA transactions were still occurring. The GPU’s memory bandwidth is the most important analysed factor that limits the performance of the implementations. Nonetheless, GPU solutions proved to be viable, since significant speed-ups could still be achieved, despite the performance

drops.

Finally, some custom hardware solutions were presented. The first architectures [Lav00, GFM⁺03] delegated the distance computation to the hardware components while a host processor was in charge of the center updates. Even without mapping the entire algorithm to hardware, high speed-ups could still be achieved. Once again, the memory accesses were a big factor in the overall acceleration. Then, several implementations were discussed that implemented in hardware both the assignment and the update steps. These designs still need the presence of a host, but only for the initial dataset upload and for minimal control tasks. The KACU architecture [LHC05] was the first fully hardware mapped solution to be presented, providing a single dividing block for all center updates. [WL07] expanded upon the KACU architecture, using a floating-point divider instead. Most solutions, including the ones used in the parallel and distributed approaches, and in the GPU implementations, divided the dataset among the several processing elements. [HBSE11] and [KBA13] instead divided the evaluation of the same datapoint over several centers through the available processing elements. This is a slightly different paradigm, in the sense that, although the datapoints are evaluated sequentially, the distance to each center is done in parallel. [AMY09] and [HBES11] presented comparison studies between CPU, GPU and FPGA implementations which clearly indicated that FPGA implementations are viable and able to achieve even higher speed-ups than GPU implementations, while being much more energy efficient.

The state-of-the-art review showed that FPGA implementations are viable solutions for the acceleration of the K-means algorithm and are even able to outperform solutions in other target platforms. The study of the many different FPGA implementations, along with some of the algorithmic transformations mentioned, provides some early foundations for the architecture detailed in the following chapters. The proposed architecture will feature the *Manhattan norm* as the chosen distance metric, since it is the most hardware efficient metric and does not greatly impact the classification results. It will also follow the paradigm used in [HBSE11] and [KBA13], as the centers will be distributed among several processing elements. This way the architecture will become highly scalable and a higher degree of parallelism among the existing processing elements can be achieved.

3

Multi-Processor Architecture for Cluster Analysis

Contents

3.1	Target Device	23
3.2	AXI Protocol	25
3.2.1	Read transaction	26
3.2.2	Write transaction	27
3.2.3	AXI4 variants: Full, Lite and Stream	27
3.2.4	AXI Interconnect	28
3.3	Hardware/Software Architecture	29
3.4	Summary	33

This chapter presents a detailed explanation of the architecture proposed and developed for the computation of the K-means algorithm. The presentation will start with an overview of the target device chosen to implement the architecture and its more relevant features. Then, the proposed architecture will be presented following a *top-down* approach.

3.1 Target Device

The designed architecture makes use of several built-in components and therefore it is important to discuss the main features and specifications of the target device from the beginning.

The whole system was implemented in a Xilinx Zynq-7000 All Programmable SoC device [Xilc]. The Zynq-7000 All Programmable SoC device series enables extensive system level differentiation, integration, and flexibility through hardware, software, and I/O programmability. These allow the design of systems with tightly coupled software based control and analytics with hardware-based processing and optimized system interfaces. The Zynq-7000 family consists of 7 different devices, all of them with roughly the same configuration. The higher tier devices offer a greater amount of programmable logic blocks as well as faster operating frequencies.

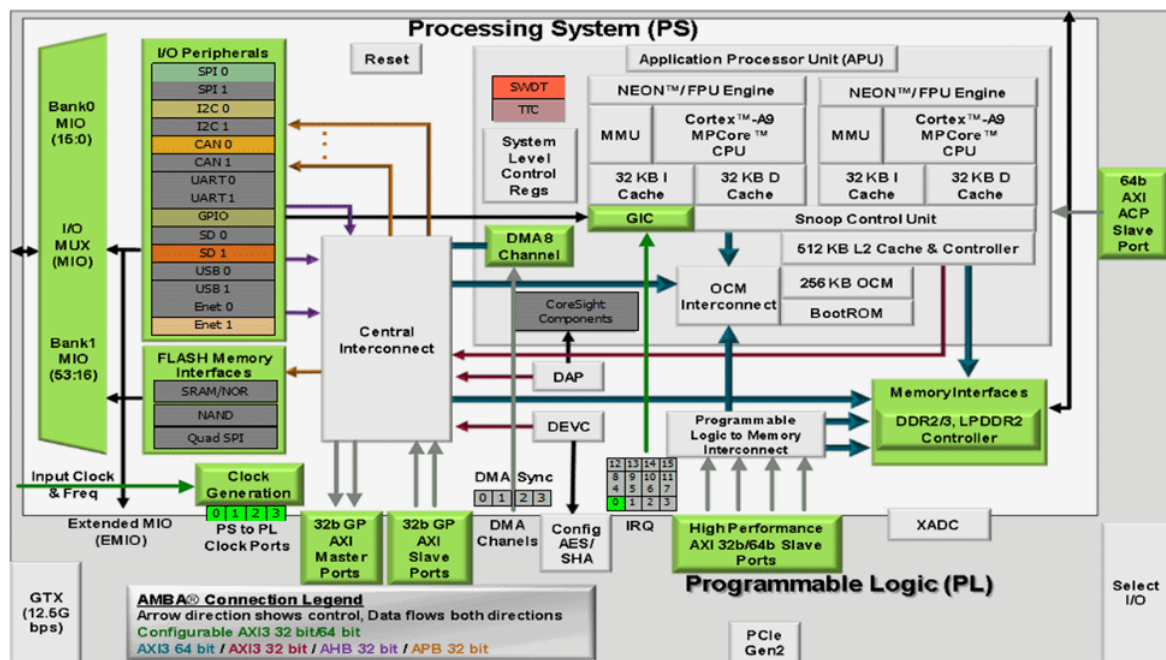


Figure 3.1: Zynq-7000 SoC main block diagram [Xild]

Figure 3.1 illustrates the main components of each Zynq device. Two main blocks are distinguished: the Processing System (PS) and the Programmable Logic (PL). The PS features an Application Processor Unit (APU) including two ARM cortex A9 *hard-cores*, which can operate at up to 866 Mhz in the low-range devices and up to 1 Ghz in the mid-range devices. Each core has its own level 1 instruction and data caches with 32 KB of space. There’s also a 512 KB level 2 cache, shared among the two processors. The PS also provides 256 KB of On-Chip Memory (OCM). Each processor features its own Floating Point Unit (FPU) engine. A general interrupt controller is offered, allowing the designer to program up to 16 interrupt routines. A built-in Double Data Rate (DDR) memory controller enables an easy

interaction with an external DDR memory. The size of the DDR memory may vary with each device. As examples, the ZYBO development board is based on Zynq Z-7010 device and offers 512 MB of DDR memory whereas the ZedBoard development board is based on Zynq Z-7020 and has 1 GB of DDR memory. Still inside the APU, 8 DMA channels are offered, of which 4 can be used for data transfers between the PS and the PL. Outside the APU, several I/O peripherals are available, e.g. SPI, I2C, CAN, USB and Ethernet interfaces, 2 UARTs. All peripherals are connected to an I/O multiplexer. The central interconnect right beside the I/O peripherals offers connectivity between the peripherals the DDR memory, the level 2 cache in the APU and the PL.

The connectivity between the PS and the PL is done through several Advanced eXtensible Interface (AXI) ports. There are:

- 2 general-purpose (GP) AXI 32-bit slave ports
- 2 general-purpose (GP) AXI 32-bit master ports
- 4 high-performance (HP) AXI slave ports (programmed to be either 32 or 64 bits wide)
- 1 accelerator coherency port (ACP) 64-bit slave port

Each port can operate at up to 150 Mhz. The designation of each port is attributed given its maximum bandwidth. Each GP port offers up to 600 MB/s of bandwidth and are commonly used for control functions and peripheral accesses. The HP ports offer up to 1200 MB/s of bandwidth and are suited for large DMA transfers. The ACP port features an optional cache coherency but it shares the CPU interconnect bandwidth.

The AXI protocol as well as the most relevant differences between the mentioned ports will be further detailed in the section (3.2).

The PL itself is based on the Xilinx Artix-7 and Kintex-7 FPGA series. The designer can make use of the available PL resources and produce his/her own custom hardware blocks, using standard hardware description languages such as VHDL or Verilog, the same way as in any FPGA. The low-range Zynq devices (up to Z-7020) are based on the Artix-7 FPGA and the high-range devices are based on the Kintex-7 FPGA. Even within the low-range devices, the Zynq-7000 PL offers up to 53.200 Look-up Tables (LUTs) and 106.400 flip-flops, 560 KB of extensible BRAM memory and 220 programmable Digital Signal Processing (DSP) slices. Table 3.1 summarizes the available PL resources in all devices in the Zynq-7000 family.

Table 3.1: Zynq-7000 Programmable logic resources

Device	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
LUTs	17.600	46.200	53.200	78.600	171.900	218.600	277.400
FFs	35.200	92.400	106.400	157.200	343.800	437.200	554.800
BRAM	240 KB	380 KB	560 KB	1.060 KB	2.000 KB	2.180 KB	3.020 KB
DSPs	80	160	220	400	900	900	2.020

3.2 AXI Protocol

This section lays down some introductory aspects and general operating procedures of the AXI protocol, which rules all the data transfers between the device's PS and the PL. As shown in the previous chapter, data transfer is an important factor in the architecture that deeply affects the overall performance of the system. More specifically, in the K-means algorithm, the communication overhead due to data transfers can have even a larger impact in the total execution time than the actual computation tasks. Therefore and in order to maximize the performance of the data transfers in the designed system, it is fundamental to take advantage of the performance oriented features of the AXI protocol.

The AXI protocol is part of the Advanced Microcontroller Bus Architecture (AMBA) specification [Xilb] and it was first presented in the third generation of the AMBA interfaces, under the version name of **AXI3**. The AMBA AXI protocol is targeted at high-performance and high-frequency system designs, including a number of features that makes it suitable for high-speed interconnections. This makes the protocol appropriate for high-bandwidth and low-latency designs. The capability of operating at high frequencies is complemented by the fact that no complex bridges are needed to connect components, allowing the protocol to meet timing requirements of a wide range of designs. Furthermore, it is backward-compatible with the previous AMBA generations' interfaces. Later, in the 4th generation of AMBA, **AXI4** was released, featuring some improvements on the AXI3 protocol, as well as introducing protocol variants in order to cater to more specific needs. These variants will be presented later on in this chapter.

In summary, the AXI protocol features:

- separate control and data phases
- support for unaligned data transfers
- ability to perform burst-based transactions
- separate read and write data channels, allowing for simultaneous reads and writes
- ability to handle multiple outstanding addresses
- ability to perform out-of-order transactions

All AXI channels operate on a basic *valid/ready* handshake. Besides the data signal, there are two other important signals: the *valid* and the *ready* signal. The source asserts and holds the valid signal when valid data is available for transfer. The destination asserts the ready signal if it is able to receive incoming data. Data is transferred when both valid and ready signals are asserted high. Once there is no more data for the source to send, the valid signal is deasserted. Similarly, if the destination is, for some reason, unable to receive any more data, it deasserts the ready signal. Another commonly used and useful signal is the *last* signal, or usually referred to as “*last* signal”. When used in a data channel, *last* indicates that the current transferred value is the last value of a given burst of packet of data. The usage of *last* is extremely useful when dealing with transfers that imply the use of data packets, the same way as it happens with data transfers over Ethernet, for instance.

Basic AXI transactions make use of 5 different channels:

- read address channel
- read data channel
- write address channel
- write data channel
- write response channel

The protocol operates on a mandatory master-slave paradigm. Each end of the connection has to be either a master or a slave and both ends can't swap designations throughout the performed transfers. This doesn't mean that the data transfer occurs in only one way, though. The master and slave classification will dictate in which direction a read transfer or a write transfer is performed. A transfer from the master to the slave is considered a write whereas a transfer from the slave to the master is considered a read. Both read and write transfers are done in a pretty similar way to one another.

3.2.1 Read transaction

A read transaction starts with a request from the master. The master specifies the data's initial address along with other control information in the read address channel. The slave replies with the data in the read data channel. Figures 3.2 and 3.3 illustrate an example of a burst read transaction. The signals in the time diagram with prefix *AR* refer to the read address channel and the signals with prefix *R* refer to the read data channel.

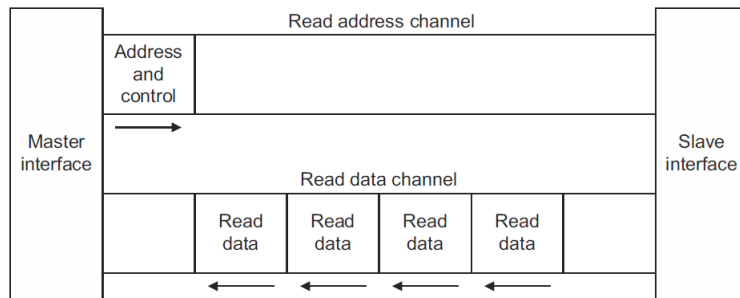


Figure 3.2: AXI read transaction [Xilb]

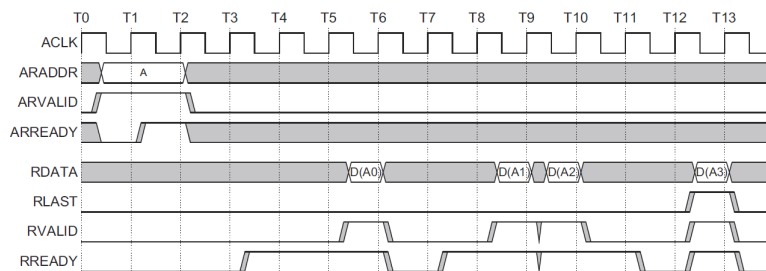


Figure 3.3: time diagram for AXI read transaction [Xilb]

3.2.2 Write transaction

A write transaction also starts with an issue from the master, this time in the write address channel. The master transmits the destination address for the transmitting data, along with extra control information. After the request is validated by the slave in the write address channel, the actual data transfer can commence, in the write data channel. Once the transfer is complete, the slave needs to issue a confirmation, stating that either the transaction completed successfully or some error occurred. This is done in the write response channel. Figures 3.4 and 3.5 illustrate an example of a burst write transaction. Signals with the prefix *AW* refer to the write address channel, signals with the prefix *W* refer to the write data channel and signals with the prefix *B* refer to the write response channel.

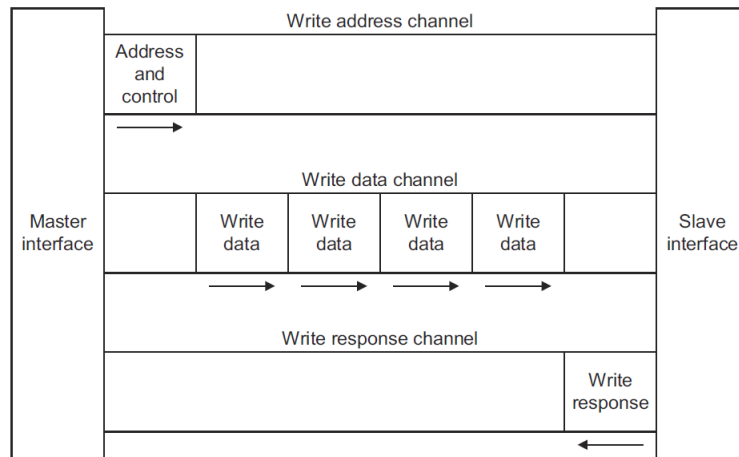


Figure 3.4: AXI write transaction [Xilb]

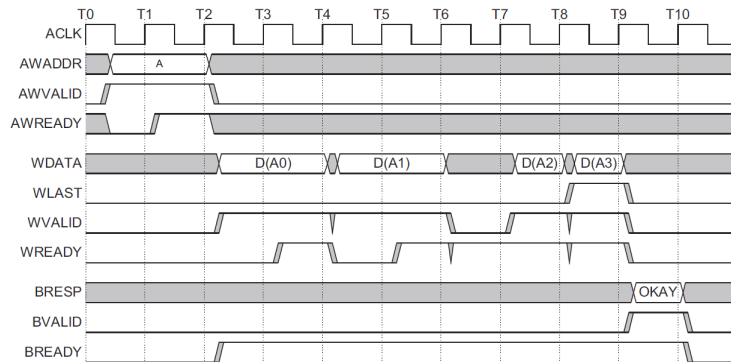


Figure 3.5: time diagram for AXI write transaction [Xilb]

3.2.3 AXI4 variants: Full, Lite and Stream

The AXI4 protocol has some differences compared to the older AXI3 protocol. AXI4 allows bursts of up to 256 beats, while AXI3 had a maximum burst length of only 16 beats. As a result, the setup overhead for each burst is diminished, for large data transfers. However, the most significant contribution of the AXI4 version was the introduction of protocol variants that differ slightly to the AXI3 protocol and are more suitable for specific scenarios. The AXI4 protocol distinguishes 3 main variants. These are:

- *AXI4-Full*, or *AXI4 Memory-Mapped*
- *AXI4-Lite*
- *AXI4-Stream*

The first variant is the closest to the original AXI protocol. It maintains the use of the 5 different channels, allows bursts of up to 256 beats, has parametrizable data widths and operates the same way as the read and write examples described earlier.

The *AXI4-Lite* variant, as the name suggests, is a lighter implementation of the *AXI4-Full* protocol. This variant also uses the 5 channels but it doesn't allow the use of bursts (only single values are transferred) and the data widths are limited to either 32 or 64 bits. However, the simpler implementation comes with a smaller footprint and less logic resources are needed. Moreover, the bridging back to *AXI4-Full* is easily attainable (usually achieved by the AXI interconnect blocks (3.2.4)). The *AXI4-Lite* protocol is, thus, suitable for momentary transfers that don't require large amounts of data nor have critical bandwidth requirements. One typical example of usage of this protocol is in initial hardware configurations.

The *AXI4-Stream* variant is the most different from the earlier AXI3 protocol. Differently from the previous variants, the *AXI4-Stream* protocol only uses one data channel, and the data only flows in one direction: from the master to the slave. In other words, the *AXI4-Stream* only uses the write data channel. This means that there is no need of specify any addresses at the start of the transaction. If both master and slave are fit to send and receive data, respectively, then the transfer starts. The *AXI4-Stream* protocol, besides also having a small footprint and being extremely simple to implement, has another major upside: it can handle unlimited burst lengths. This makes it appropriate for data connections that need to handle large data transfers with low latency and high bandwidth requirements (or, as the name of the protocol itself implies, it is ideal for streaming data). Along with the unlimited burst advantage, the *AXI4-Stream* protocol has a parametrizable data width, allows data packing and also supports unaligned transfers.

For the sake of clarification, figure 3.6 illustrates a simple *AXI4-Stream* burst transaction, using *tlast* to signalize the end of a given packet of data.

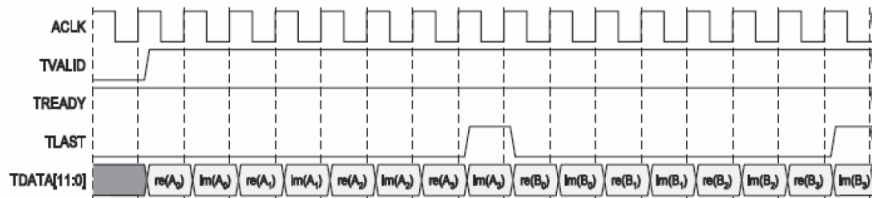


Figure 3.6: AXI4-Stream transaction [Xilb]

3.2.4 AXI Interconnect

One important block that features most designs using AXI protocols to connect several elements, and that will feature several times in the implemented architecture, is the Interconnect block. This block

allows the same interface (either master or slave) to be connected to several blocks at once, as well as performing protocol conversions if necessary. A typical AXI interconnect block features the peripheral interfaces for master and slave components, attached to appropriate clock conversion mechanisms, width converters and data FIFOs as the designer sees fit, and a central routing block called the *crossbar*. The way the crossbar block controls the output of each transaction can be configured. The routing rules can be as simple as round-robin solutions or can be based on the address of each component in the register space. The latter is frequently used, since the routing issue gets taken care of without the designer's input: all the designer has to ensure is to correctly specify the address in each AXI transaction.

The interconnect can assume many different known configurations. These can be as simple as to directly connect a single master to a single slave (called the *Pass Through* configuration). Other configurations include a simple protocol conversion between a single master and a single slave (*Conversion Only* configuration), several masters being connected to the same slave (*N-to-1 Interconnect* configuration), a single master connected to multiple slaves (*1-to-N Interconnect* configuration) and, more generally, multiple masters connected to multiple slaves (*N-to-M Interconnect* configuration).

3.3 Hardware/Software Architecture

The developed architecture utilizes resources from both the PS and the PL segments of a Zynq-7000 SoC device. As mentioned earlier, the design follows a hardware/software co-design methodology, meaning that both custom hardware blocks and software applications were developed concurrently. The Zynq-7000 SoC already provides a clear separation between the software and hardware domains, given that the PS already contains the ARM cores capable of executing software applications and the PL offers the resources needed for the implementation of the hardware blocks. The architecture was developed using both custom made hardware blocks and Xilinx's pre-built cores in the LogiCORE Intellectual Property (IP) catalog. The design was implemented using Xilinx's Vivado 2014.4 tool and the software was developed using Xilinx's SDK 2014.4.

As referred, the two main steps of the K-means algorithm are the assignment step (classify the datapoints and change the appropriate accumulators and counters) and the update step (update the centers' locations). The first step can still be subdivided into the two elemental tasks: classifying the datapoint and dealing with the accumulators and counters. In the proposed architecture, the datapoint classification is done by the custom hardware components. The accumulations and increments will be performed by the two ARM cores. The final center updating step will be performed by one of the ARMs, since the computational overhead of updating the centers is underwhelmed by the remaining computation and the actual time gain in parallelizing the update step would be minimal. All operations are performed using single precision floating-point representation. Table 3.2 summarizes the task assignment information in both hardware and software domains.

The devised solution features parallelism in both hardware and software domains. The datapoint classification task is parallelized through the several existing accelerators, by splitting the centers among them. This solution offers a more scalable hardware architecture, with the number of accelerators being limited only by the number of centers considered for the clustering problem and the device's resources.

Table 3.2: Task delegation between Hardware and Software domains

Task		Performed by
Assignment Step	Datapoint Classification	HW Components
	Accumulations/Counting	ARM
Update Step		ARM

The accumulations/counting task is also parallelized, by dividing the task through both ARM cores. Parallelism is also achieved between hardware and software domains, as both tasks can be performed simultaneously, in some extent. This is achieved through the use of interrupts between the PS and the PL. A block diagram of the top level design is shown in figure 3.7.

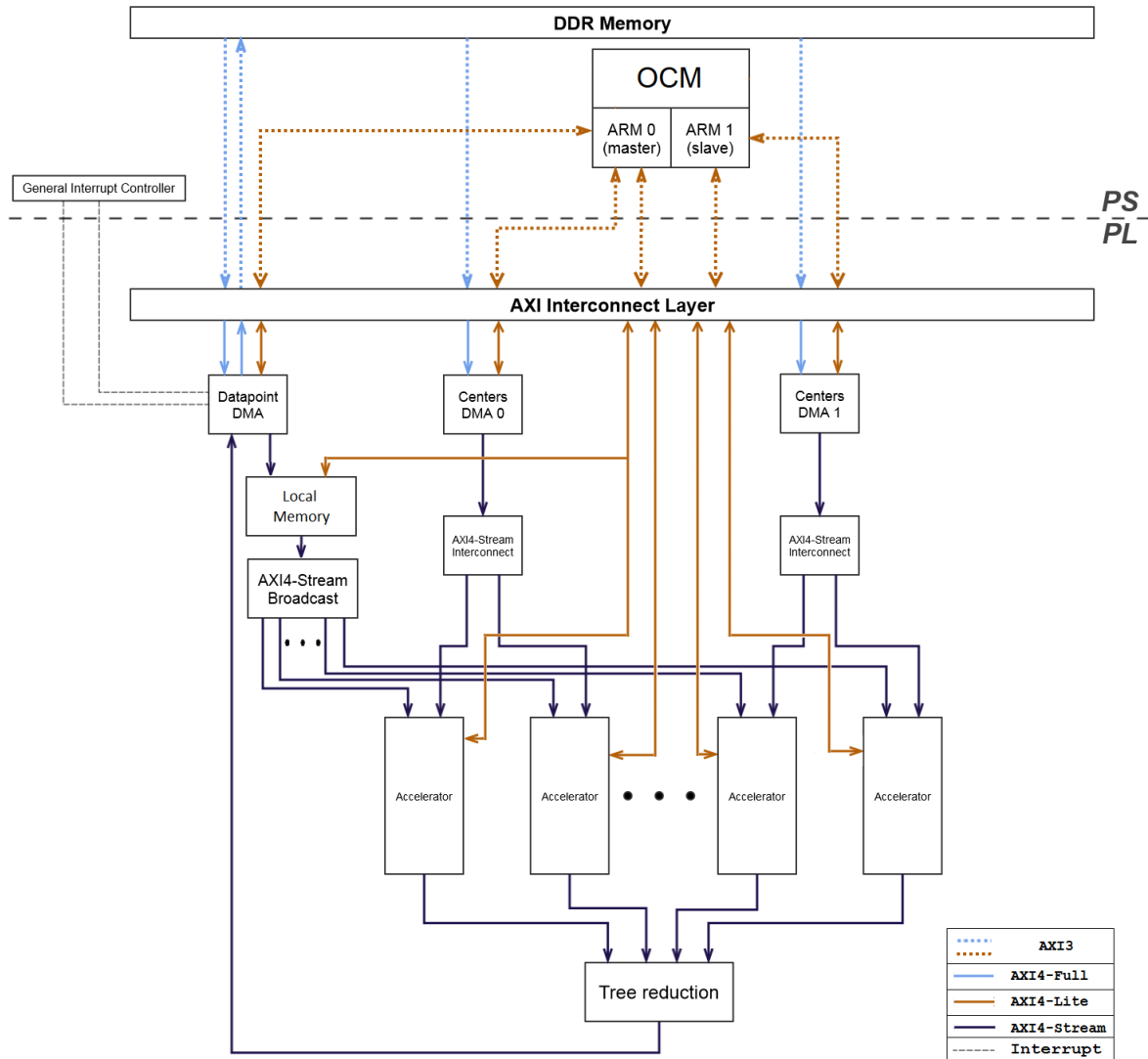


Figure 3.7: Top Level design architecture

The main components on the PS part are the two ARM Cortex-A9 processors. Both processors operate simultaneously during the algorithm’s execution, one acts as a master, the other as a slave. The OCM memory holds the binary executables for the software application running on both cores.

The dataset to be classified in the clustering process is stored in the external DDR memory. The

dataset can be accessed by the software components (ARM cores) and, through the DMA blocks resident in the PL, by the hardware components. Since the datapoints don't change throughout the algorithm's execution, there is no need to perform any type of synchronization between these accesses. The DDR memory also contains the centers being used in each iteration of the algorithm. These values can change during the algorithm's execution. However, since the two main steps of the algorithm occur sequentially, the synchronization needed to access the centers is already enforced by the algorithm: the DMA blocks read the centers during the assignment step while the master ARM core updates the centers in the update step. The memory will also hold the classifications of each datapoint for the iteration being processed, which are updated through the Datapoint DMA block.

The General Interrupt Controller (GIC) is used to handle two different interrupt signals coming from one of the DMA blocks. The purpose of these two interrupts will be clarified later on. The interrupt signals from the DMA block are connected to the GIC through 2 of the 16 existent IRQ ports, portrayed in figure 3.1. These are plain simple connections with no protocol involved. Both orange and blue dashed lines represent AXI3 connections. The use of two different colours helps to denote the different types of AXI variants each connection will get, once they get through the AXI interconnect layer, as well as the different type of AXI interface each connection uses. The end-points of each connection on the PS side aren't directly the ARM cores and the DDR memory themselves, but the AXI interfaces already provided in the PS. By visually connecting the PS end-points to the ARMs and the memory, a more intuitive visualisation of what entity is in charge of each connection is offered. All of the dashed connections are bi-directional: the use of uni-directional lines in the blue dashed connections serves to emphasize the directions in which data flows throughout the execution.

The blue connections are meant for data transfer and will be converted to AXI4-Full once they go through the AXI interconnect layer. These use, therefore, the HP ports. The orange lines are meant for initial component configurations and will be converted to AXI4-Lite, therefore will use the GP ports.

The AXI Interconnect Layer basically consists of several AXI Interconnect blocks responsible for performing AXI3-to-AXI4 conversions and vice-versa, as well as connecting the AXI interface ports of the PS to the appropriate blocks in the PL (namely, the DMA and hardware components).

The data communication to/from the hardware components is processed through 3 DMA blocks. The *Datapoint DMA* is firstly responsible for reading the datapoints from the DDR memory and delivering them to the accelerators. Each DMA read transaction is issued by the master ARM core, through the AXI4-Lite interface. Once each transaction is completed, the ARM core gets notified through one of the interrupt signals connected to the GIC. This DMA block is also responsible for receiving the classifications from the *Tree Reduction* block and writing them in the DDR memory. The interrupt signal associated to the memory writes also gets asserted when each transaction ends, notifying the ARM core that a new burst of results is available.

The two Centers DMAs communicate the updated center values from the main memory to the hardware accelerators, which are stored in local memories within each accelerator. As they only read values from the memory, they only need a read channel, and therefore are much simpler than the main DMA block. Each ARM core is responsible for configuring its own center DMA block, through the AXI4-Lite

interface. Each DMA block is responsible for reading half of the centers from the DDR memory via the AXI4-Full interface and delivering them to the accelerators. Unlike the datapoint DMA, the centers DMA blocks don't rely on interrupts: a polling mechanism is used instead, since there is no need to alert the processors when each centers' transfer is completed.

Data that goes through the DMA blocks is subjected to a protocol change, automatically guaranteed by the DMA blocks' functionalities. These are capable of performing AXI4-Full to AXI4-Stream conversions, and vice-versa. The AXI4-Stream is more appropriate for the transfer of both datapoints and centers, since the number of values needed to transfer all datapoints and centers can be pretty large.

Before handing the values to the accelerator blocks, both datapoints and centers still need to go through a couple more hardware blocks. The datapoints coming from the DMA block get stored in a local memory in the PL. Since the accelerators need to compute the distance of a single datapoint to all of the centers, the same datapoint needs to be re-used several times. Multiple DMA transactions of the same datapoint would increase the communication time significantly, and must be minimized. The same reasoning can be made for the centers, which also get stored in local memories within each accelerator. For these reasons, the use of local memory is imperative, in order to achieve any kind of speed-up. If the system had to rely on several DMA transactions of the same data during the same iteration, the execution time would increase dramatically, specially when the data is being initially held in the device's DDR memory.

The *Local Memory* block consists of a dual-port 8KB BRAM and a custom designed BRAM controller. The controller provides an AXI4-Stream interface to the BRAM and handles the BRAM signals (such as *dataIn*, *dataOut* and address signals) accordingly. The controller makes use of specific configuration values used for the first accelerator, namely the number of centers and the data dimensionality, in order to figure out how many floating-point values make up a complete datapoint and for how much time should a datapoint be kept in memory.

The controller uses the memory's port A to store incoming values from the DMA. It allows the memory to hold up to two datapoints at the same time: the datapoint being analysed, and the datapoint that follows next. This way, the following blocks will always perceive the datapoints as being in the local memory and the minimum latency access is ensured throughout the whole execution. The controller uses port B to send the datapoints to the following hardware blocks. Depending on the number of centers per accelerator, it may have to send the same datapoint several times.

As the top level design implies, the computation is parallelized in the hardware domain by splitting the distance calculation among the existing accelerators. Each accelerator is responsible to compute the distance of all datapoints to a subset of all the centers. In other words, computation is split by dividing the centers among the accelerators, instead of splitting the dataset. In order to achieve this, all datapoints need to be broadcasted to all the accelerators. This is done through the *AXI4-Stream Broadcast* block. As for the centers, each half runs past an AXI4-Stream interconnect block. Each interconnect block splits the incoming centers among half of the accelerators.

The custom hardware accelerators receive the datapoints and their respective centers from the broadcast block and the interconnect blocks, respectively, and are responsible for computing the *Manhattan*

distance between each datapoint and the centers. If more than one center is assigned to the accelerators, these are also capable of figuring out which center index produced the minimum distance. All accelerators output both the minimum distance found and its respective center index. These are received by the *Tree Reduction* block, which performs a tree comparison of all the results and outputs the center index associated to the lowest distance. This final result is sent back to the Datapoint DMA, which writes the result in the DDR memory. All the transfers beginning in the DMA blocks until the Tree reduction block are done using the AXI4-Stream protocol.

3.4 Summary

This chapter provided a detailed explanation of the hardware/software co-design architecture proposed to compute the K-means algorithm. The target device family chosen for this architecture was the Xilinx's Zynq 7000 All Programmable SoC, which provides the hardware resources of Xilinx's Artix-7 and Kintex-7 FPGA series, as well as the computational power of two ARM Cortex-A9 *hard-cores* in a single chip.

An overview of the architecture was firstly presented following a top-down approach. This included a brief explanation of what each component accomplishes in the algorithm as well as presenting how the different components are interconnected.

The solution described in this chapter applies a parallel computation scheme in both hardware and software domains. Multiple hardware accelerators compute the datapoint classifications in parallel, with the centers being split among them and the datapoints being broadcasted efficiently in hardware. This approach produces a highly scalable hardware architecture, independent from the hardware/software bandwidth. The accumulations/countings from are also parallelized, by the 2 ARMs. In addition, further parallelism is achieved by performing both the datapoint classification task and the accumulations/countings simultaneously.

The following chapter describes, in a more detailed manner, the hardware blocks residing in the PL, as well as the developed software.

4

Hardware & Software Components

Contents

4.1	DMA Block	35
4.2	AXI4-Stream Broadcaster	36
4.3	AXI4-Stream Interconnect	37
4.4	Hardware Accelerators	38
4.4.1	Memory and BRAM Controller	40
4.4.2	Invalidate Block	40
4.4.3	Floating-Point Cores	40
4.4.4	Minimum Distance Block	41
4.5	Tree Reduction Block	42
4.6	Software Component	43
4.7	Summary	45

This chapter expands upon the architecture description given in the previous chapter, by presenting the main components of the architecture in more detail. Firstly, the main hardware blocks residing in the PL will be addressed, followed by the software component developed for both ARM cores.

4.1 DMA Block

All the DMA cores used were instantiated from the Xilinx LogiCORE IP Catalog [Xila]. These cores were used instead of the built-in DMA channels in the PS for two main reasons. Firstly, the built-in DMA channels are obliged to use the GP ports, which offer a much lower bandwidth and therefore are not appropriate for large data transfers. Lastly, the cores provided by Xilinx already perform the AXI4-Full to AXI4-Stream conversions needed, since all the subsequent components use AXI4-Stream interfaces. If using the PS DMA channels, the same conversion could be done, with an AXI4-Stream First In First Out (FIFO). However, this solution is a lot harder to implement and the potential gain in area occupied in the PL is minimal.

Each core contains a set of registers accessible via the AXI4-Lite interface. These registers are intended for initialization, status and management purposes, such as initializing the channels in either polling or interrupt mode, issue data transfers and check for errors in a given transaction. The core can be configured to have either a read channel, a write channel or both channels simultaneously. The read and write channels operate independently from one another. Each channel has its own high-performance *DataMover* block, which is also responsible for performing the AXI4-Full to AXI4-Stream and AXI4-Stream to AXI4-Full conversions. The *DataMover* also contains buffers, capable of storing a certain amount of data until the next block needs it.

Although not used in the developed architecture, an optional Scatter/Gather engine may be enabled, which allows the user to issue several data transfers at once, with each transfer starting at different base addresses. This is useful for transferring data that doesn't reside in a contiguous block of memory.

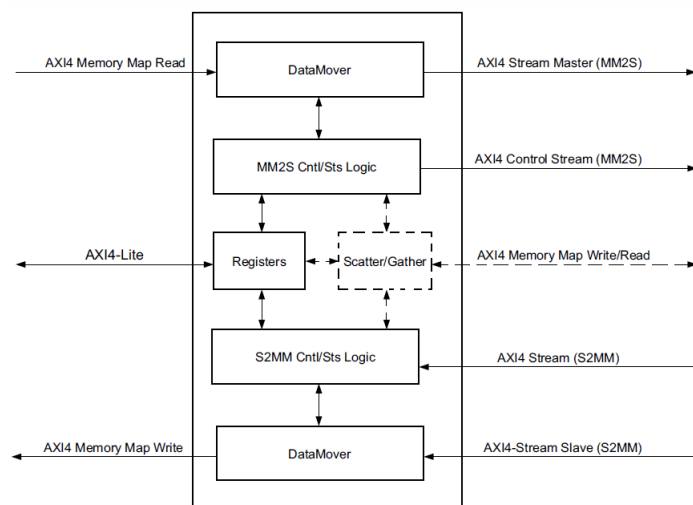


Figure 4.1: Xilinx's AXI DMA core [Xila]

Figure 4.1 illustrates the Xilinx's AXI DMA core through a block diagram. The read channel performs

an AXI4 Memory Map to AXI4-Stream conversion (MM2S). The write channel performs an AXI4-Stream to AXI4 Memory Map conversion (S2MM).

4.2 AXI4-Stream Broadcaster

The AXI4-Stream broadcaster is a custom hardware block specifically designed for sending the datapoints to all the accelerators, therefore avoiding the far less efficient solution of issuing several DMA transfers of the same data and assigning each transfer to one accelerator. The broadcaster block was designed as a generic IP core that can be used in any environment where broadcasting incoming data via the AXI4-Stream protocol is needed. By performing the broadcast in the hardware domain, the PS/PL bandwidth is spared, as an increase in the number of accelerators used does not imply the need of more data transfers.

Each AXI4-Stream interface of the broadcaster block contains the following AXI4-Stream signals:

- the data signal (*tdata*). The width needs to be concerted beforehand
- the ready signal (*tready*)
- the valid signal (*tready*)
- the “last” signal (*tlast*)

Since the purpose of the block is to broadcast from a single source onto several destinations, the block contains a single slave interface and several master interfaces. As mentioned earlier, an AXI4-Stream transfer is performed when the master’s *tvalid* signal and the slave’s *tready* signal are both asserted. Both *tdata* and *tlast* can be routed directly since these won’t impact whether a transfer gets performed or not. However, these signals need to stay stationary until a transfer can be performed to all outputs simultaneously. This can be done through some simple logic functions.

The slave interface needs to be aware when all the master interfaces are ready to receive data. This is performed by computing the AND logic function of all the masters’ *tready* signals and connecting the result to the master’s *tready* port.

The logic for the *tvalid* signals can be a bit more tricky. Each master interface needs to know when the slave interface has a valid value to broadcast. However, knowing this on itself isn’t enough, since the slave will only start the broadcast when all masters are ready. In order for each master to receive the same correct broadcast value, the receiving *tvalid* signal of each master needs to take into account both the slave’s *tvalid* and the other masters’ *tready* signal. When all of these are asserted, then a valid broadcast can start. This is performed, once again, by a simple AND function of the slave’s *tvalid* and the remaining masters’ *tready* signals.

Figure 4.2 illustrates the logic behind the AXI4-Stream broadcaster, using 3 masters as an example. Each AND gate used needs to have the width equal to the number of master interfaces and the number of AND gates used in the *tvalid* logic also needs to be equal to the number of master interfaces.

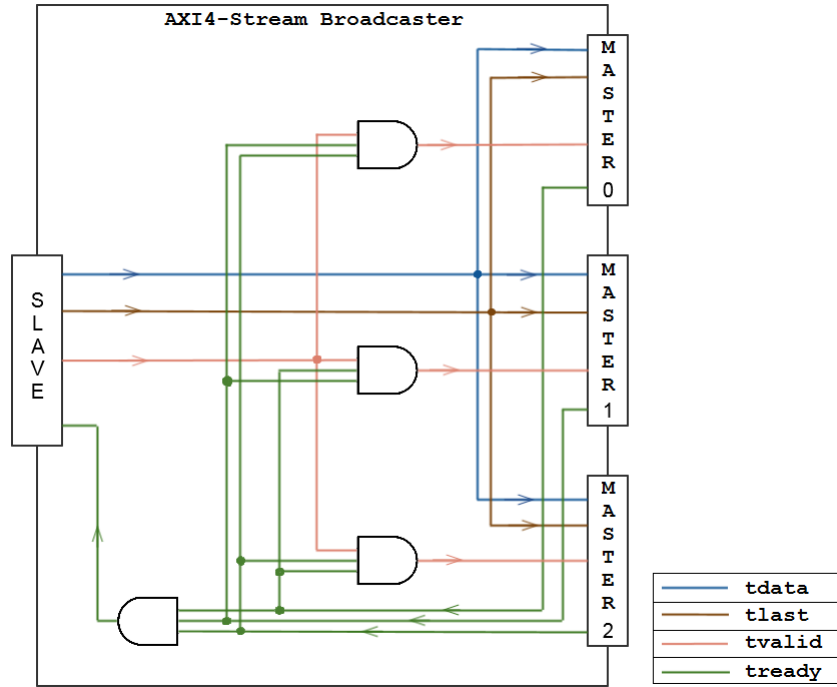


Figure 4.2: Block Diagram of a 3-master AXI4-Stream Broadcaster

4.3 AXI4-Stream Interconnect

The AXI4-Stream Interconnect block used is another custom made hardware block designed specifically to split the incoming values (the centers, in this case) as equally as possible among the connected accelerators. Since AXI4-Stream interfaces don't provide an identifiable base address, the write transfers are performed using a pre-determined order.

As illustrated in the top level diagram shown in figure 3.7, each interconnect block receives half of the centers from the respective DMA block. The centers arrive to the interconnect block ordered by their respective ID, meaning that the center of cluster 0 arrives first, followed by the center of cluster 1 and so on. For an arbitrary number of clusters, C , the DMA block controlled by the master ARM delivers the first half of the centers in the ascending order of cluster ID, from cluster 0 to cluster $C/2 - 1$. Similarly, the DMA block controlled by the slave ARM delivers the second half of the centers, also in ascending order of cluster ID, from cluster $C/2$ to cluster $C - 1$.

The interconnect block distributes each half of the centers to the respective accelerators, in such way that the best load balancing is achieved (*i.e.* the centers are split as evenly as possible). In order to perform the correct comparisons later on in the tree reduction block, the same ascending order by cluster ID needs to be maintained throughout all the accelerators. More specifically, the accelerator connected to the first branch of the reduction tree needs to handle the subset of clusters with the lowest ID. The accelerator connected to the second branch needs to handle the subset of clusters with the second lowest ID, and so on. If, for example, a round-robin rule were applied instead, the ascending ordering would be lost throughout the accelerators and the final results would be incorrect.

In order to preserve the correct ordering, the interconnect blocks, along with the DMA blocks, make

use of the *tlast* signal in the AXI4-Stream protocol. When sending the centers, the ARM cores issue the *tlast* signal to be asserted when the last center of each subset is being sent. This way the interconnect block knows when to redirect the output to a different accelerator.

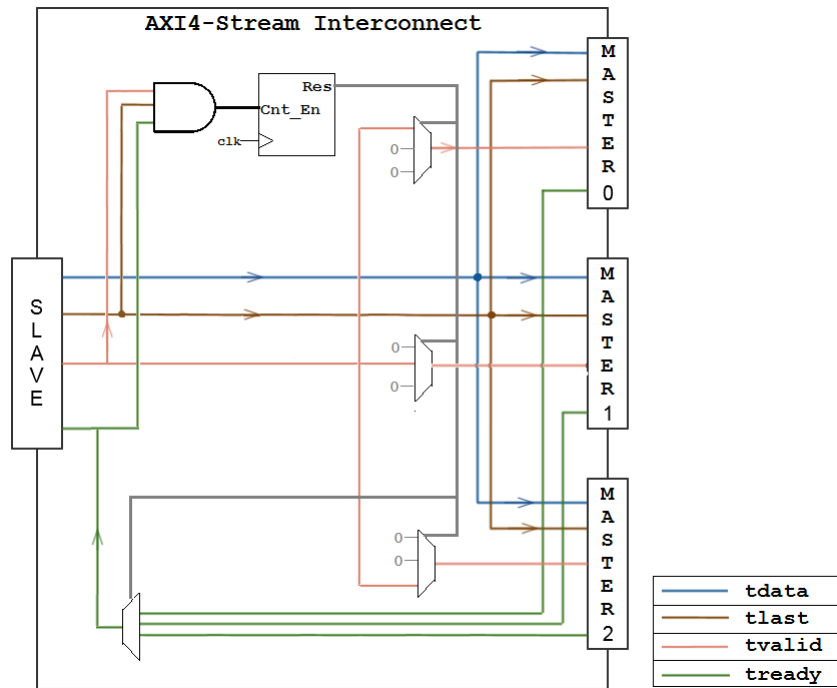


Figure 4.3: Block Diagram of a 3-master AXI4-Stream Interconnect

Figure 4.3 illustrates the block diagram of the AXI4-Stream interconnect, in an example using 3 masters. The block uses a counter which counts from zero to number of masters - 1 and is incremented each time the output needs to be redirected. This happens when the last value is sent successfully to the current active master. Hardware-wise, this is equivalent to both *tvalid* and *tlast* of the slave interface being asserted, as well as the *tready* signal of the current selected master. The slave interface makes use of a multiplexer, in order to evaluate the *tready* signal of the current selected master. Each master also makes use of a multiplexer, in this case to select the appropriate *tvalid* signal. If the master is the one selected for output, then the *tvalid* signal coming from the slave goes through the multiplexer. Otherwise, the master always gets zero.

4.4 Hardware Accelerators

The hardware accelerators are the most important custom blocks in the PL side of the architecture, computation-wise. These will be responsible for computing the *Manhattan* distance of all the datapoints to all the centers and finding the closest center (of the subset of centers assigned to that specific accelerator) to each datapoint. Several hardware accelerators can be used in parallel, while the centers are split as evenly as possible throughout all the accelerators. Each accelerator receives the same datapoint at the same time, coming from the broadcaster block via AXI4-Stream. It also receives one or more centers from one of the AXI4-Stream interconnect blocks.

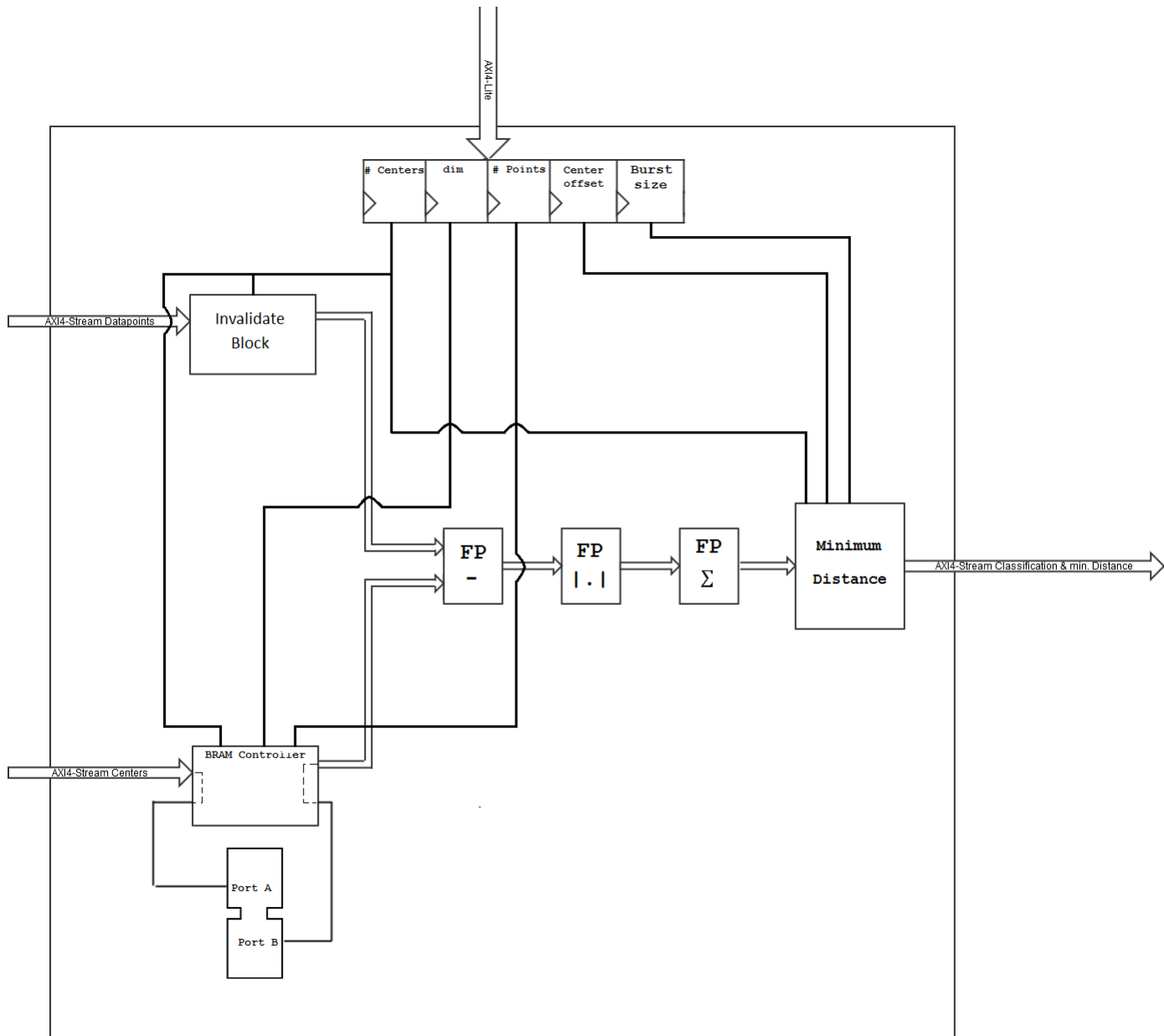


Figure 4.4: Block Diagram of the Hardware Accelerator

A block diagram for the accelerator is portrayed in figure 4.4. The BRAM stores the centers received from the DMA block. Since the number of centers is far inferior to the number of datapoints, it is conceivable to store all the centers in this local memory and share them with all the computation blocks.

The centers come via AXI4-Stream, so a simple AXI4-Stream BRAM controller was designed. These are explained in more detail in section 4.4.1.

There are 5 registers that hold the initialization values of the accelerator, given through the AXI4-Lite interface. These are used by the BRAM controller, the *Invalidate block* and the final *Minimum Distance* block. The controller needs these parameters in order to correctly cycle through the intended values in the memory. The *Invalidate block* needs the parameters to invalidate broadcasted datapoint values, in case the accelerator has less centers to evaluate than other accelerators. The *Minimum Distance* block needs these to correctly keep track of the current ID of the closest cluster and to assert *last* appropriately,

so that the end of each burst of results is properly signalled.

Finally, the remaining 4 computational blocks are the floating-point subtracter, the *absolute value* block, the floating-point accumulator and the *Minimum Distance* block. All of these blocks, as well as the BRAM controllers, are connected to one another via AXI4-Stream, with each interface having the same 4 signals as the broadcaster and the interconnect blocks.

4.4.1 Memory and BRAM Controller

The local memory in each accelerator consists of a dual-port BRAM that holds the centers for that specific accelerator. During the implementation phase, a default local memory size of 16KB was used. This value can, however, be changed if the lack of memory in the PL starts to become a concern.

The BRAM's port A is used to write the incoming values from the DMA blocks. Port B is used by the accelerator's computational blocks to read the values. The access from both ports is accomplished with the aid of a custom made BRAM controller, which provides an interface between the AXI4-Stream protocol and the usual BRAM ports, containing clock, reset, enable, write-enable, data-in, data-out and address signals, just like the controller made for the datapoint local memory. The controllers themselves consist of a series of counters which cycle through the memory positions according to the parameters given from the registers in the AXI4-Lite interface. Each controller is able to output one value to the floating-point operators every 2 cycles.

The BRAM controller for the centers' memory uses the number of centers, number of points and dimensionality parameters to control the memory accesses. It needs both the number of centers and the dimensionality to figure out when a datapoint was indeed matched with all of the centers. It also needs the number of datapoints in order to figure out when all the datapoints were processed and the centers currently stored in memory became outdated.

4.4.2 Invalidate Block

The invalidate block was created to solve a small problem that would occur when the several accelerators have a different number of centers. Given the way the centers are distributed throughout the accelerators, there is the possibility that some accelerators get an extra center. This happens when the number of total centers is not divisible by the number of accelerators. Each value needs to be broadcasted a number of times equal the maximum number of centers per accelerator. The accelerators with one less center will receive an extra broadcast, due to this feature. The invalidate block's job is to detect this extra broadcast, and invalidate it by deasserting the *tvalid* signal of AXI4-Stream.

The invalidate block counts the number of times a value gets broadcasted, and once the number of broadcasts surpasses the number of centers in the accelerator, the following value is invalidated.

4.4.3 Floating-Point Cores

Each accelerator has 3 floating-point hardware blocks, each one responsible for one of the 3 floating-point operations needed to compute the *Manhattan* distance. All of them were generated from the Xilinx LogiCORE IP catalogue. All the floating-point cores used in the architecture were configured for 32-bit

floating-point representation with support for denormalized numbers. The configurations regarding the cores' resource usage, latency and throughput were optimized for an operating frequency of 100 Mhz, which was considered the maximum frequency upon analysing several place and route timing reports, during the implementation phase.

The first block used in the computation is the subtracter block. This block has two slave interfaces, one for each element of the operation, and a master interface for the result. The subtracter core has the necessary logic for the computation as well as buffers for each input, allowing several values to be fed to the core ahead of time. A new value is computed once each buffer contains valid data.

Several different possible resource versus latency/throughput configurations were evaluated, for the base operating frequency of 100 Mhz. The chosen implementation has a latency of 4 cycles and a throughput of 1 result per cycle. The core was also configured to make use of the PL's DSP slices. The subtracter uses 2 DSP slices and LUTs.

Computing the absolute value of a value in floating-point representation simply means forcing the most significant bit to zero. The internal structure of the absolute value core is simply a direct connection from input to output of the 31 less significant bits and a zero in the most significant bit.

The accumulator block contains a 32-bit register, initially reset to zero, and a floating-point adder. The core receives incoming floating-point values through its AXI4-Stream slave interface and adds the incoming value with the value stored in the register, until a reset is issued. A reset is performed by asserting the *last* signal during a valid transaction.

Similarly to what was done with the subtracter core, the best resource usage versus latency/throughput configuration was chosen, for an operating frequency of 100 Mhz. The core was configured with 10 pipeline stages and can achieve a throughput of one accumulation per cycle. The core consumes 5 DSP slices and 694 LUTs.

4.4.4 Minimum Distance Block

The *Minimum Distance* block is responsible for receiving the incoming distances from the accumulator and comparing them with the lowest distance obtained so far. Its block diagram is shown in figure 4.5.

The minimum distance block receives the *Manhattan* distances from the floating-point accumulator block via AXI4-Stream. A floating-point comparator (portrayed with the "less than" symbol in the figure) checks if the newly received distance is lower than the lowest distance so far (represented by the signal *currentMinDistance*). The lowest distance is stored in one internal register, that is initially set with the infinity value, in floating-point representation, so that the the first received distance gets treated as the lowest distance so far. The register is again set to the infinity once the distances to all centers are all processed.

The comparator was also custom designed and produces two outputs: *newMin* and *minDistance*. The first signal is a mere 1-bit value which indicates if the received distance is indeed a lower value than the current minimum distance. The second signal is the lowest distance of the two values compared.

A counter that cycles between the IDs of the clusters associated to the accelerator is used to match the incoming distances to their correct clusters. The counter is incremented for each valid distance received.

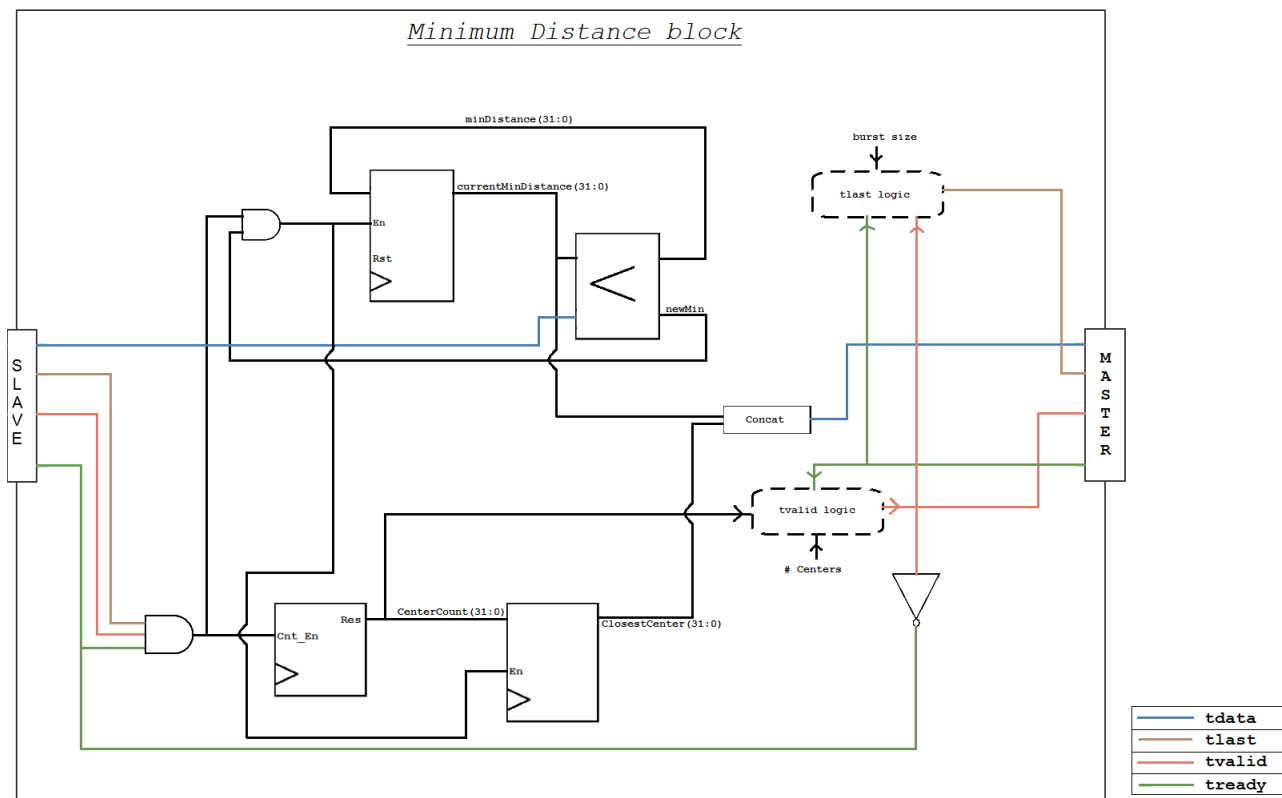


Figure 4.5: Block Diagram of the Minimum Distance block

The register following the counter stores the ID of the closest center. It does so by only storing the counter values when a new lowest distance was found.

Both the minimum distance and the closest center signal are concatenated and the resulting 64-bit signal is output to the AXI4-Stream data port.

The *tlast logic* and *tvalid logic* blocks are simple control blocks to appropriately assert the AXI4-Stream control signals. The specified burst size value dictates the periodicity in which *tlast* is asserted. The specified number of centers attributed to the accelerator dictates the periodicity in which *tvalid* is asserted.

The minimum distance core is able to evaluate one distance per clock cycle. However, not all the distances are counted as valid results in the output, since only the minimum distance is required. Therefore the core has an effective throughput of one minimum distance value every C cycles, with C being the number of centers attributed to the accelerator.

4.5 Tree Reduction Block

The *Tree Reduction* block is responsible for receiving the classifications provided by all accelerators and performing a tree comparison of each of the results. Since each accelerator only computes a classification given a subset of the centers, the results of the accelerators themselves do not actually represent the intended result. This block performs the remaining comparisons and produces the final classification with respect to all the centers.

The block is custom designed and consists of a series of AXI4-Stream registers¹ and floating-point comparators, organized in a tree like structure. The number of branches and levels in the tree needs to be defined beforehand, given the number of accelerators in the design. For K accelerators, the tree needs K branches and $\log(K)$ levels.

Figure 4.6 illustrates an example of the component, for a case using 4 different accelerators. Both classifications and minimum distances are stored in the AXI4-Stream registers. These are configured to have 64 bit *tdata* signals, since both classification and minimum distance values occupy 32 bits.

The comparator block is similar to the comparator used in the *Minimum Distance* block. The only two differences are on the comparison rule and the way the comparator handles the wider data signal.

The comparator in the *Tree Reduction* block operates on a “Less than or Equal” rule instead of a “Less than” rule. According to the way the centers were divided through the accelerators in the first place, the upper branch of the comparator will always handle the clusters with ID lower than the clusters handled by the lower branch. In order to maintain coherency with a strictly sequential comparison of the distances, in case of an equal distance for two clusters, the cluster with lowest ID is selected.

Since the data signal is wider and does not only contain the comparable distance, the comparator block needs to also be able to extract the distances from the data signals. The comparator evaluates 32-bit distance from the 64-bit data signal (the distance was established to be the in the 32 less significant bits of the data signal) and outputs the complete 64-bit data signal correspondent to the lowest distance and its respective AXI4-Stream control signals.

The comparisons are performed similarly throughout the several levels of the tree, until they reach the last AXI4-Stream register. Once all comparisons are done, the minimum distance itself can be discarded. Only the final classification, located in the 32 most significant bits of the data signal, is needed to proceed with the algorithm’s execution, and therefore the final register is 32 bits, unlike the previous registers.

The *Tree Reduction* block is capable of producing 1 valid result per clock cycle, due to its pipelined structure.

4.6 Software Component

This section describes the tasks performed by both ARM cores, during the execution of the K-means algorithm and how these interact with the hardware architecture described in the previous sections. This overview of the complete procedures performed by the CPUs is important to contextualize all the computation done by the previously described accelerators, as well as to further explain the computational parallelism achieved by using both available ARM *hard-cores*.

When working in a parallel computing environment, the need of synchronization variables is to be expected. In this specific case, synchronization variables will be needed as the data intended to be shared among the two CPUs will change amongst iterations. To avoid data cache coherency issues, a simple 8KB BRAM in the PL side is used as an uncached shared memory for both ARMs. The access to this memory is done through a simple AXI BRAM controller IP core (provided by Xilinx LogiCORE cat-

¹AXI4-Stream register is the name given to a core with a master and slave AXI4-Stream interfaces and a register with width equal to all AXI4-Stream signals combined

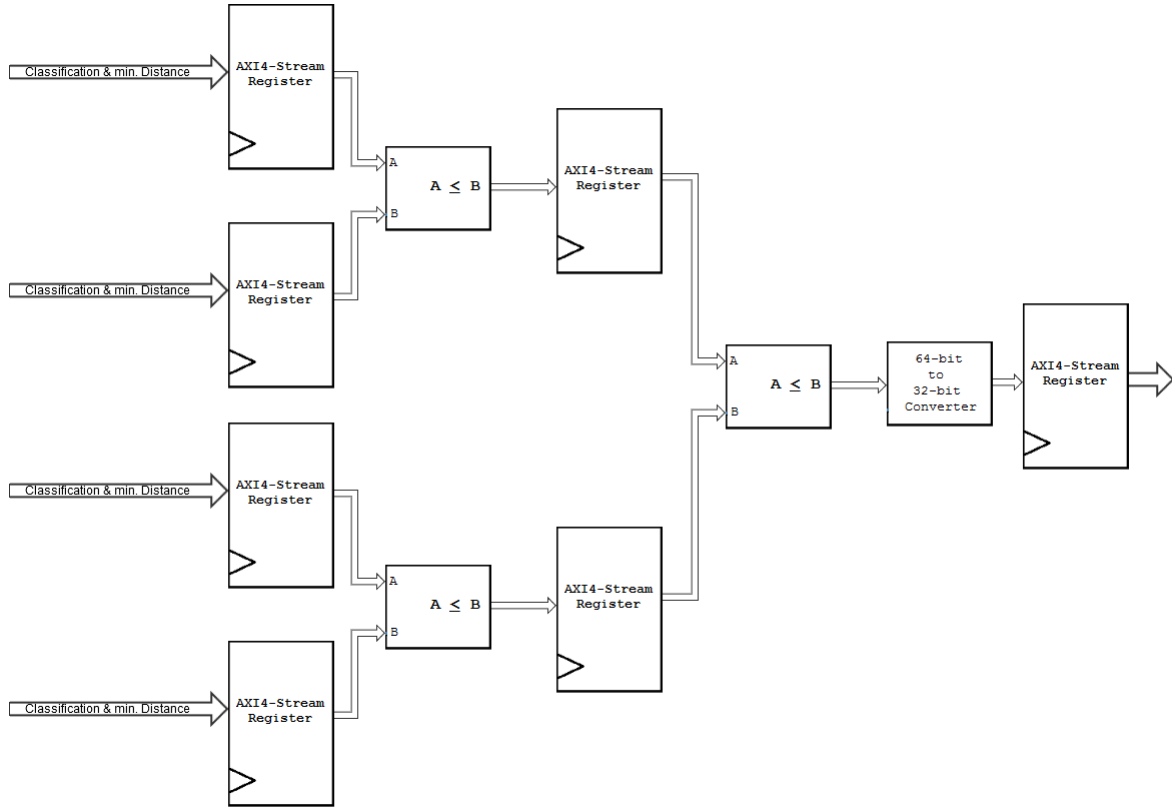


Figure 4.6: Block Diagram of the Tree Reduction block

alg), which enables the ARM cores to use the memory the same way as they would use the DDR memory.

The pseudo-code for the master ARM program is presented in algorithm 4.1 and the pseudo-code for the slave ARM is presented in algorithm 4.2. Starting with the first step of the K-means algorithm, the master ARM will initialize all the centers. Then, several hardware specific initializations must be performed. Each ARM core initializes one half of the accelerators (though a simpler solution where a single ARM performs all the initializations should not cause a big negative impact on the execution time). Finally, the master ARM initializes the datapoint DMA core and each ARM initializes its own centers' DMA core.

After all initializations are done, the iterative process of the algorithm starts. Each ARM needs to reset its accumulators and counters, the same way as in the sequential version. After all the resets are performed, a barrier function is used, so that both centers' DMA send their centers to the accelerators at the same time. The barrier function uses the shared uncached memory in the PL and it is based on the comparison of values from several memory addresses (one address per processor). The master ARM then orders the accelerators to start computing by issuing the send command to the datapoint DMA.

The code also uses two new variables, identified in the pseudo-code as *resultsReady* and *resultsProcessed*, that also are reset in the beginning of each iteration. *resultsReady* is a variable that only gets modified in the datapoint DMA's interrupt routine, that gets triggered once a burst of results arrives to the DDR memory. Inside the interrupt routine, *resultsReady* gets incremented by *B*, with *B* being the defined burst size, and a new receive call is issued to the DMA, if there are still more results left to

receive. Once outside the interrupt routine, the master ARM can start updating the accumulators and counters by successfully entering the second *while* loop. This prevents the ARMs from having to wait for all results to be delivered in order to start updating the accumulators and counters, providing parallelism between both subtasks of the assignment step. The second variable, *resultsProcessed*, keeps track of how many results were processed by both ARM cores. Once its value reaches the number of datapoints, the assignment step is completed and the update step can start.

The accumulations and counts are preceded by another barrier function, which serves as a way to warn the ARM slave core that a new burst of results has arrived and to force both CPUs to perform the updates at the same time. The accumulator and counter updates are parallelized on the ARM cores, such that each ARM processes one half of each result burst received from the hardware blocks independently by simply accessing different base addresses in the DDR memory. The amount of memory space needed on the OCM for the accumulator and counter results will vary depending on the number of centers and dimensionality. Once all result bursts arrive and all results are processed, the final accumulations and counts from both ARMs are merged.

Once the assignment step is complete on both ARMs, the merging needs to be done. The master gets through the final barrier function once the slave has written its values in the appropriate base addresses for the master to access. Then, the master merges all accumulators and counters.

Finally, the center updates are done the same way as in the sequential version.

Although the tasks performed by the slave were already mentioned, we briefly summarize them here. After all the initializations are done, the slave sends the centers through the DMA, waits in the barrier function for the master to receive the result bursts, performs the updates and writes the partial values in the uncached memory so that the master can perform the merge. The slave performs each iteration indefinitely, until the master issues the slave to break out of the infinite *while* loop. This functionality was implemented inside the barrier function itself.

To conclude this section, a few metrics regarding the software will be presented. The master software was compiled and linked using the ARM *gcc* tool. The total program size was roughly 165KB. The master code contains 600 lines in total. Given the reduced complexity of the slave's software, a program size of 51KB, with 292 lines of code was obtained.

4.7 Summary

This chapter provided a detailed explanation of the hardware/software co-design architecture proposed to compute the K-means algorithm, with a detailed presentation of each main hardware component and software.

The DMA block was the first component described in more detail, followed by the AXI4-Stream broadcaster and interconnect blocks. The hardware accelerator blocks deserved a bit more emphasis, as they are the main computational blocks of the design and directly map the classification process of the K-means algorithm. Finally, the tree reduction block was explained, which is responsible for merging the results of all accelerators together.

To conclude, the Software Component section described the computation on both ARM cores and how the software components interface and control the computation of the hardware components.

In this chapter, the hardware frequency and a few characteristics of the hardware components were already mentioned. The next chapter will expand upon these features and provide a more in-depth analysis, as well as a report of other project metrics.

Algorithm 4.1 master ARM pseudo-code

Input: dataset, dataset size N , number of clusters C , data dimensionality D , burst size B , number of accelerators K

Output: classifications, center coordinates

```
1: centerInitialization();
2: for each accelerator  $k = 0$  to  $k = K/2$  do
3:   accelInit( $k$ );
4: end for
5: initDatapointDMA(INTERRUPT_MODE);
6: initCentersDMA(POLLING_MODE);
7: repeat
8:   for each center  $c$  do
9:     classAccumulator[ $c$ ] = 0;
10:    classCounter[ $c$ ] = 0;
11:   end for
12:   resultsReady = 0;
13:   resultsProcessed = 0;
14:   Barrier();
15:   DMA Send(CENTERS_BASE_ADDR,  $C/2$ );
16:   DMA RecvInit(CLASSIFICATIONS_BASE_ADDR,  $B$ );
17:   DMA Send(DATAPOINTS_BASE_ADDR,  $N$ );
18:   while resultsProcessed <  $N$  do
19:     while resultsProcessed < resultsReady do
20:       Barrier();
21:       for  $d =$  resultsProcessed;  $d <$  resultsProcessed +  $B/2$ ;  $d++$  do
22:         classifications[ $d$ ] = *(CLASSIFICATIONS_BASE_ADDR +  $d$ );
23:         classAccumulator[classifications[ $d$ ]] += *(DATAPOINTS_BASE_ADDR +  $d$ );
24:         classAccumulator[classifications[ $d$ ]] ++;
25:       end for
26:       resultsProcessed +=  $B$ ;
27:       Barrier();
28:     end while
29:   end while
30:   Barrier();
31:   for each center  $c$  do
32:     classCounter[ $c$ ] += *(SLAVE_COUNTERS_BASE_ADDR +  $c$ );
33:     classAccumulator[ $c$ ] += *(SLAVE_ACCUMULATORS_BASE_ADDR +  $c$ );
34:   end for
35:   for each center  $k$  do
36:     newCenter[ $k$ ] = classAccumulator[ $k$ ] / classCounter[ $k$ ];
37:   end for
38: until (centers don't change)
```

Algorithm 4.2 slave ARM pseudo-code

Input: dataset, dataset size N , number of clusters C , data dimensionality D , burst size B , number of accelerators K

Output: classifications, center coordinates

```
1: for each accelerator  $k = K/2$  to  $k = K$  do
2:   accelInit( $k$ );
3: end for
4: initCentersDMA(POLLING_MODE);
5: while TRUE do
6:   for each center  $c$  do
7:     classAccumulator[ $c$ ] = 0;
8:     classCounter[ $c$ ] = 0;
9:   end for
10:  resultsProcessed = 0;
11:  Barrier();
12:  DMA_send(CENTERS_BASE_ADDR,  $C/2$ );
13:  while resultsProcessed <  $N$  do
14:    Barrier();
15:    for  $d = \text{resultsProcessed} + B/2$  ;  $d < \text{resultsProcessed} + B$ ;  $d++$  do
16:      classifications[ $d$ ] = *(CLASSIFICATIONS_BASE_ADDR+d);
17:      classAccumulator[classifications[ $d$ ]] += *(DATAPOINTS_BASE_ADDR+d);
18:      classCounter[classifications[ $d$ ]] ++;
19:    end for
20:    resultsProcessed +=  $B$ ;
21:    Barrier();
22:  end while
23:  for each center  $c$  do
24:    *(SLAVE_COUNTERS_BASE_ADDR +  $c$ ) = classCounter[ $c$ ];
25:    *(SLAVE_ACCUMULATORS_BASE_ADDR +  $c$ ) = classAccumulator[ $c$ ];
26:  end for
27:  Barrier();
28: end while
```

5

Analysis and Experimental Results

Contents

5.1	Hardware Throughput	50
5.2	PS/PL Bandwidth	51
5.3	Hardware Resources	52
5.4	Execution time and Speed-ups	54
	5.4.1 Analysis and predictions	54
	5.4.2 Experimental Results	58
5.5	Floating-Point Performance	64
5.6	Summary	67

This chapter will cover all the measurements and testing made to the architecture designed in this work. The measurements include the typical important metrics in any hardware solution, such as the throughput and the amount of programmable logic resources needed. Another important aspect in hardware/software co-designs is also the achieved bandwidth between hardware and software domains, which is also evaluated in this chapter. The experimental testing will portray the achieved speed-up and performance values for several test cases. The experimental values will be put side by side with theoretical predictions, which were calculated during an *a priori* analysis for both the execution time and performance metrics.

5.1 Hardware Throughput

An important metric to take into account when evaluating hardware performance is the throughput. The throughput quantifies how much of a given output produces per unit of time, usually measured in either clock cycles or seconds. A system's throughput value may refer to several different outputs. For instance, it can refer to the number of instructions executed (instruction throughput), or to the number of data results computed (data throughput). In this section, the result throughput offered by the hardware solution will be evaluated.

In order to evaluate the hardware throughput, the path each portion of data takes in the architecture needs to be analysed. In the designed solution, the input data are the datapoints and the centers, which go through roughly the same path. After they reach their respective DMA block, the datapoints go through the AXI4-Stream Broadcast block and then go into the accelerators. The centers go through the AXI4-Stream interconnect block and then go to the accelerators. Since the datapoints and the centers don't arrive to the accelerators at the same time and the accelerators are forced to wait for at least one datapoint and one center to arrive in order to start the computation, a rigorous latency value can only be attributed to the hardware covered from the accelerator block onwards.

Recalling the latency and throughput of each block within the accelerator:

- the initial BRAM controller has a latency of 2 cycles and a constant throughput of 1 value every 2 cycles
- the floating-point subtracter has a latency of 4 cycles and a throughput of 1 value per cycle
- the floating-point absolute value block has no latency
- the floating-point accumulator has a latency of 10 cycles and a throughput of 1 value per cycle
- the *minimum distance* block has a latency of 1 cycle and a throughput of one partial result per cycle

Outside the accelerator blocks, the *tree reduction* block has a latency of $\log_2 K$ and a throughput of 1 result per cycle.

To compute the overall throughput of the entire path, the combined throughput of all components needs to be found. The initial throughput of the BRAM controller is the first limiting factor, since it

can only produce an output every 2 cycles. The second limiting factor is the accumulator: despite its throughput of one value per cycle, a valid *Manhattan* distance is only available every D cycles (with D representing the data dimensionality), due to the several needed accumulations. The *minimum distance* also implies a similar limitation. Despite its latency and throughput of 1 cycle, a comparison between the distances to all centers assigned to the accelerator must be made, in order to obtain the correct classification. In total, $\lceil \frac{C}{K} \rceil$ comparisons must be made. The overall throughput is related to all the limiting local throughputs. The hardware design outputs **1 classification every $2D\lceil \frac{C}{K} \rceil$ cycles**.

5.2 PS/PL Bandwidth

This section covers the bandwidth results observed in the data transfers between the PS and the PL, accomplished using the AXI protocol. As described in the previous chapter, the main data transfers occur through the HP interfaces of the Zynq-7000 device. These offer the largest bandwidth of all of the available interfaces. In their maximum bandwidth configuration, they can provide up to 1200 MB/s of bandwidth, when configured in 64-bit mode and when the operating frequency is the highest allowed by the interface (which peaks at 150 Mhz). Since the designed architecture deals with single precision floating-point values and each value needs to be fed sequentially to the hardware, the 64-bit configuration mode can not be applied to this scenario, cutting off the maximum peak bandwidth by half. The designed hardware is also designed to operate at 100 Mhz. Since both the designed hardware and the AXI interface receive the same clock signal, the expected maximum throughput achieved using the HP ports drops to 400 MB/s.

The actual experimental results were measured using a performance monitor IP core, from Xilinx's LogiCORE catalogue. This core can be configured to monitor several AXI connections at once and can provide several useful metrics for both read and write transactions, such as minimum and maximum latency observed, total bytes transferred and number of beats transferred. To measure the bandwidth, the total read and write latency as well as the number of bytes transferred over the AXI4 channels were measured.

The measurements were made during several runs of the K-means algorithm, using different datasets. The bandwidth for transfers from the PS to the PL had an average of 364 MB/s, with a variance of up to 10 MB/s amongst all results. This results in a 91% occupation of the total theoretical available bandwidth in the HP port. The lower experimental value is mainly due to the transfer latency, which is not constant throughout, as well as the limited burst size of the AXI protocol. As for the bandwidth for transfers from the PL to the PS, the results obtained were a lot more consistent. An average value of 395 MB/s was observed, accounting for 98.75% of the total available bandwidth. Since the throughput of the hardware results is significantly inferior to the throughput of the initial data coming from the PS, and since each result gets immediately sent to the DMA block after it is computed, a much lower transfer throughput is to be expected.

Given the fact that the number of accelerators used does not alter the way data is transferred nor the amount of data transfer, the bandwidth values are the same for any given number of accelerators. Due to the datapoint broadcast and the computation split by centers instead of datapoints, the use of more

accelerators does not compromise the bandwidth, making this solution highly scalable.

5.3 Hardware Resources

Another important metric for evaluating hardware solutions is the amount of resources occupied by the architecture. The balance between area occupied by the hardware and performance is always a point of interest when evaluating hardware solution. Knowing the area occupied by the design will also allow to predict execution time results for implementations using several accelerators, e.g. using devices with more programmable logic resources.

The DMA blocks remain the same regardless of the number of accelerators used: there are always two *Centers DMA* blocks and a single *Datapoint DMA* block (see figure 3.7). The *Datapoint DMA* block requires significantly more resources, since it has both MM2S and S2MM channels, while the *Centers DMA* blocks only have a MM2S channel. The resource utilization for these cores was obtained through the utilization reports provided by the Vivado tool. The results for the DMA blocks are stated in table 5.1, with the last row containing the resources utilized by all 3 blocks.

Table 5.1: Resource usage for the DMA blocks

Block	LUTs	Registers	Slices	BRAM tiles
Datapoint DMA	1450	2030	600	3
Centers DMA	458	696	227	1
Total	2366	3422	1054	5

The amount of resources used by the AXI interconnect layer increases with the number of accelerators used, since each accelerator adds one more base address to be taken care of, adding up complexity to the crossbar block. However, the increase in the programmable logic resources used is fairly small.

Table 5.2 states the resource usage of the AXI interconnect layer for up to 3 accelerators. The usage for a single accelerator can be seen as a "base resource usage", that get incremented as more accelerators are introduced. The obtained results suggest an increment of roughly 300 LUTs, 300 registers and 120 slices per accelerator.

Table 5.2: Resource usage for the AXI interconnect layer

# Accelerators	LUTs	Registers	Slices
1	2748	3174	1047
2	2980	3566	1169
3	3371	3880	1299

Table 5.3 details the usage of each building block of the accelerator, along with the total resource usage of the complete accelerator. This analysis emphasises and makes clear the impact of using floating-point arithmetic in the occupied hardware area. The floating-point cores do take its toll in the amount of resources used, specially the accumulator block. The values regarding the BRAM usage were taken according to the memory sizes stated in the previous chapter. If the device's available BRAM usage starts to be a bottleneck in the overall occupied area, the memory dimensions can be tweaked in order to overcome this issue.

Table 5.3: Resource usage for the accelerator block

Block	LUTs	Registers	Slices	BRAM	DPSs
Invalidate block	94	32	26		
BRAM Controller	266	251	123		
BRAM memory				16KB	
Floating-point subtracter	284	251	123		2
Floating-point accumulator	694	475	279		5
Minimum distance block	245	98	73		
Total	1583	1016	597	16KB	7

The resource usage of the *Tree Reduction* component also depends on the number of accelerators. The resulting tree of comparisons has $\log_2(K)$ register levels, with K being the number of accelerators used. Being a full and complete binary tree, there will be $\sum_{i=0}^{\log_2 K} 2^i$ AXI4-Stream registers and $\sum_{i=0}^{\log_2 K-1} 2^i$ floating-point comparators. Table 5.4 states the usage of both the comparator and the register, as the resource increments for a variable number of accelerators. Despite the somewhat complex resource increments, when comparing with the previous results, it is important to mention that the amount of resources used is far inferior to the usage of the previous blocks. The growth of the reduction tree will carry a small footprint resource-wise.

Table 5.4: Resource usage for the Tree Reduction component

Component	LUTs	Registers	Slices
Comparator	48		
AXI4-Stream Register	18	65	26
Resource increment for each K	$48 \sum_{i=0}^{\log_2 K} 2^i + 18 \sum_{i=0}^{\log_2 K-1} 2^i$	$65 \sum_{i=0}^{\log_2 K-1} 2^i$	$26 \sum_{i=0}^{\log_2 K-1} 2^i$
K = 2	162	65	26
K = 3	339	195	78

The most significant part of the resources needed is due to the accelerator blocks and to the additional logic in some of the other components, such as the AXI interconnect blocks and the *Tree Reduction* block. It is possible to identify a base resource usage, which does not vary with the number of accelerators, and an overall incremental usage, needed every time an accelerator is added to the system. Within the incremental usage, a linear increment and a logarithmic increment can be identified. The area occupied by the accelerators and the AXI interconnect layer grows linearly with the number of accelerators used, and the *Tree Reduction* block grows logarithmically. Table 5.5 presents the total resource usage, distinguishing base usage from incremental usage. The AXI4-Stream broadcaster and interconnect blocks were not featured since the amount of resources used by both is very residual (close to 10 LUTs, in total).

The complete resource usage for any given number of accelerators allows any hardware designer to appropriately choose between several target devices, given its dimensions and amount of programmable logic offered. For the purpose of *Big Data* analysis, it is likely that a large number of accelerators could benefit the overall execution performance of the algorithm. So the number of accelerators able to fit in a single device is an important metric to estimate. Given these values, the Z-7100, which is the most complex device of the Zynq-7000 family, can hold up to 144 accelerators, being limited by the LUT usage. Whether or not such a large number of accelerators would benefit the execution depends from the

Table 5.5: Total resource usage

Block	LUTs	Registers	Slices	BRAM tiles	DSPs
DMA	2366	3422	1054	5	
AXI layer base usage	2748	3174	1047		
Datapoint Local Memory block	323	154	92	2	
PS shared memory	278	258	110	2	
Total Base Resource Usage	5715	7008	2303	9	
Accelerator	1583	1016	597	4	7
AXI layer incremental usage	300	300	120		
Linear Incremental Resource Usage	1883	1316	717	4	7
Tree Reduction Block					
Log. Incremental Resource Usage	66	65	26		

application. Some more specific analysis regarding execution times is made in section 5.4.

5.4 Execution time and Speed-ups

The most meaningful metric regarding the solution’s evaluation is the gain in execution time, as it directly translates to the architecture’s algorithm acceleration, which is the main goal of this work. As discussed previously, reduced execution time is achieved by both exploiting the available parallelism in the hardware and software domains, using 2 ARM CPUs and several hardware accelerators.

5.4.1 Analysis and predictions

The execution time analysis starts with a study of how the use of several CPUs and accelerators impact the execution time. It is not clear that blindly adding more computational units to the architecture would always benefit the results and it is important to be able to accurately estimate how each computational unit would impact the results, at least to some extent. A good analytical model of how the execution time will progress given any set of parameters will reflect how each parameter impacts the results and be able to predict the outcomes for any given dataset and configuration. This will be particularly useful for estimating the results when a large number of accelerators is used, since the smaller and cheaper devices can only fit a small and limited number of accelerators.

In order to evaluate the execution time progression throughout different types of datasets and different number of accelerators, judging the overall execution time of the algorithm is not enough. Instead, obtaining time estimates for each step of the algorithm is necessary, since not all the computational times for each step evolve the same way for different scenarios. This is not only because each step may use different computational elements (*i.e.* ARMs and/or accelerators) but also because the complexity of each step may depend on different parameters of the dataset (number of points, dimensionality, etc.). Therefore, the first step of this analysis is to provide expressions able to model the execution time of each step. The overall computational time will consist of the sum of the several individual expressions.

The expressions obtained are a mix of both purely analytical results and estimates made through the measurement of several experimental cases. The results are based on the accelerator’s estimated throughput and on the clock cycles spent by the ARM processor. Experimental results were obtained mainly for estimating portions of code linearly dependent on one single variable (as it would happen in a single loop within the algorithm). The expressions obtained model the following computational steps:

- $T_{classify_{ARM}}$, which stands for the time the ARM CPU takes to classify one single datapoint
- $T_{classify_{Accel}}$, which stands for the time the accelerators take to classify one single datapoint
- T_{accum} , which stands for the time the ARM CPU takes to update the accumulator and the counter, given a single classification
- T_{update} , which stands for the time the ARM CPU takes to update a single center

Table 5.6 provides the expressions obtained for each of the computational tasks. These expressions already account for both pure computational time and eventual memory accesses needed. For all expressions, N represents the number of datapoints in the dataset, C represents the number of centers, D represents the data dimensionality and K represents the number of accelerators used. All expressions are measured in clock cycles, using an operating frequency equal to the hardware frequency (100 Mhz, in this case).

Table 5.6: Expressions for the computational time of each step

Step	Expression
$T_{classify_{ARM}}$	$4.3CD + 4D + 8.5C + 2$
$T_{classify_{Accel}}$	$2D \lceil \frac{C}{K} \rceil$
T_{accum}	$8D + 3$
T_{update}	$7D$

The time taken by the ARM CPU to classify a datapoint produced the most complex expression, since it is also by far the most complex task to be performed, computational-wise. The code for this section requires nested loops dependent of both the number of centers and data dimensionality. This expression was obtained by both analysing the clock cycles taken by the ARM and measuring the time taken using several different datasets. However, being a computationally demanding task, the actual time taken by the developed architecture is indeed that taken by the hardware accelerator, $T_{classify_{Accel}}$. The first expression will be used to model the execution time of a complete software implementation, which will become relevant further on in the analysis of the speed-up.

The time taken by the accelerators to classify a single datapoint is roughly given by the hardware’s throughput. The expression does not account for any delays due to the data transfers between the PL and the PS. The experimental measurements made for this step only had a slight variance to the hardware throughput, since the bandwidth available for the result transfers is not shared by any other transfers.

The last two expressions, T_{accum} and T_{update} , were obtained using a linear regression of several measurements for different executions. This was possible since it is known that both steps only depend

on the data dimensionality. T_{accum} has an added constant term since updating the counter is in fact independent of the data dimensionality.

Although not covered in the presented computational steps, the designed solution also introduces a timing overhead, due to the accumulator merges between both ARM cores. However, and despite the merge operation complexity being linearly dependent from the data dimensionality, the timing overhead is negligible in the total computation time.

With the execution times of each computational task estimated, it is possible to formulate an overall execution time model for the complete algorithm. The K-means algorithm is purely iterative and each execution of the algorithm may require a different number of iterations each time. This makes the execution time of a single iteration more relevant for the analysis, rather than the complete execution, since comparing results of runs with different number of iterations would obviously not be viable.

Even within a single iteration, the way each computational step impacts the overall execution time may be different, depending on the situation. As the computation is parallelized in both ARM cores and the accelerators simultaneously, the accelerators send their results in bursts, so that the ARMs can proceed with their own computation without having to wait for all results from the accelerators to arrive. Depending on which computational unit finishes its part the fastest, the ARM cores may end up in one of two different situations: either they are faster than the accelerators and need to remain idle until the next burst arrives, or they are slower than the accelerators and each burst arrives when the previous one is still being processed. Each case produces different estimates for the computational time. The time diagrams 5.1 and 5.2 portray each of the cases visually.

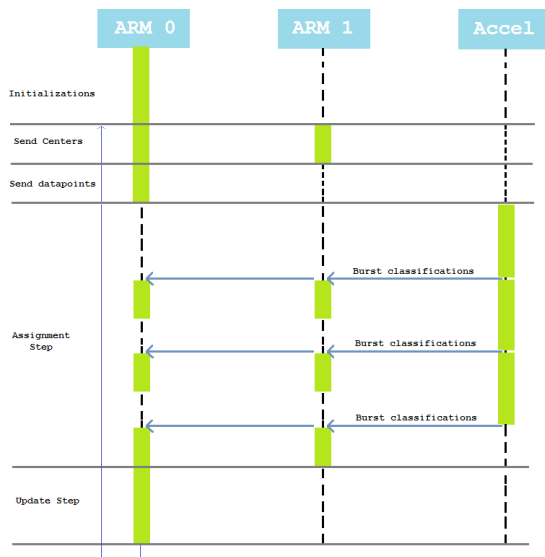


Figure 5.1: Case 1 - Time diagram for a faster ARM execution

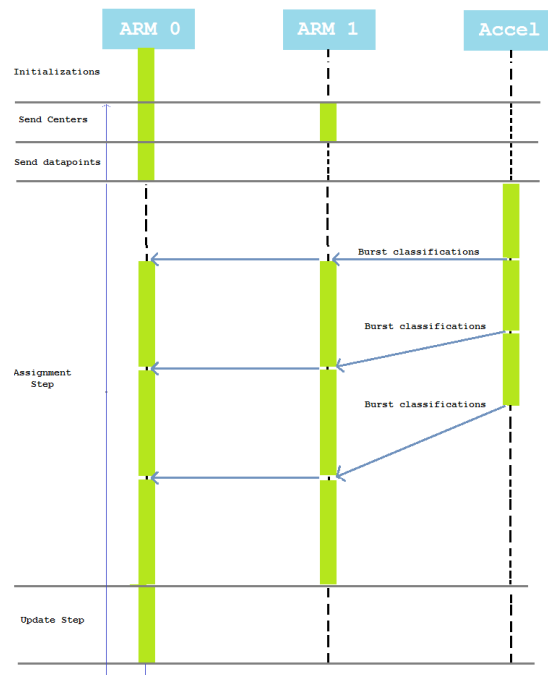


Figure 5.2: Case 2 - Time diagram for a faster accelerator execution

In the first case, the time taken by the accelerators occupies most of the execution time for the

assignment step. The opposite occurs in the second case, where the computation of the first burst of results is the only influence from the accelerator. Each of the cases produces its own execution time. From the time diagrams, the expressions for the execution time of each iteration can be deduced.

$$T_{iteration} = NT_{classify_{Accel}} + \frac{B}{2}T_{accum} + CT_{update} \quad (5.1)$$

$$T_{iteration} = BT_{classify_{Accel}} + \frac{N}{2}T_{accum} + CT_{update} \quad (5.2)$$

Expression 5.1 estimates the iteration time for case 1. The ARM cores remain idle until the first burst of results is available and, since the ARM cores are faster than the accelerators, they are always ready to handle the next burst, when it arrives. The total iteration time, that accounts for both the assignment step and the update step, is mostly influenced by the time spent by the accelerators. The influence introduced by the ARMs' computation is only noticeable in the last burst of results, where the accelerator has completed all the needed computation and stays idle.

Expression 5.2 estimates the iteration time for case 2. In this case, the accelerators perform their computation faster than the ARM cores. As a consequence, the ARM cores switch places, when it comes to who have the most impact in the execution time. The time taken by the accelerators is only relevant in the first burst of results. The remaining of the iteration time is given by the ARMs' computational time.

Both cases are obviously mutually exclusive, meaning that the system can only operate on one of them, for a given dataset and number of accelerators used. The boundary that separates both cases is given by the computational time each ARM and the accelerator spend in the assignment step. Case 1 occurs when the ARMs analyse their half of the burst of results faster than the accelerators are able to produce a single burst. Mathematically, case 1 occurs when $\frac{B}{2}T_{accum} < BT_{classify_{Accel}}$. Otherwise, case 2 occurs. The boundary $\frac{B}{2}T_{accum} = BT_{classify_{Accel}}$ represents the architecture's optimal working conditions, where the datapoint classification task takes the same amount of time to complete as the accumulation/counting task. Since the architecture can contain a variable number of accelerators but only features 2 ARM CPUs, the only possible way of further parallelizing the algorithm is to add more accelerators. This would cause the architecture to shift to case 2, meaning that the speed-up improvements would be less noticeable.

With the iteration times estimated for all possible scenarios, an estimate for the expected speed-up results can also be achieved. A pure software implementation on a single ARM processor was implemented. The software execution is forced to be completely sequential, meaning that the expected iteration time is the sum of all the computational times for each step. Equation 5.3 formalizes the mathematical expression for the sequential version, $T_{iteration_{Seq}}$.

$$T_{iteration_{Seq}} = NT_{classify_{ARM}} + NT_{accum} + CT_{update} \quad (5.3)$$

To obtain the theoretical speed-up value, one must simply perform the division $\frac{T_{iterationSeq}}{T_{iteration}}$, using the expression of $T_{iteration}$ for the appropriate case.

5.4.2 Experimental Results

This section presents all the experimental results obtained for the execution time of the algorithm, for several different datasets and number of accelerators used. The obtained results will be compared with analytical model described in the previous section and conclusions will be drawn, regarding both the speed-up behaviour for different settings and the accuracy of the theoretical model given the experimental results. In all experiments, the timing results were obtained using an AXI timer IP core provided by Xilinx, operating at the same clock frequency as the accelerators.

When testing the solution, whether using both real and synthetic data, it is important to try to cover a wide range of possibilities, in order to properly evaluate the solution’s response to several situations and how the many different possible parameter changes can alter the results, in either a positive or a negative way. In clustering, all the different specifiable parameters are tightly connected to the execution time. More specifically, in the K-means algorithm, the number of datapoints is the parameter that mostly impacts the execution time (since it is directly related to the amount of repetitive computation made within a single iteration), but the number of centers and dimensions also have their relevance as they dictate the degree of computational complexity within the several different steps of the algorithm.

The tests made relied on several datasets of both synthetic and real data taken from the UCI machine learning repository [UCI] . In order to evaluate the speed-up evolution with respect to the several parameters existent in a clustering problem, multiple runs and datasets were used. In each experiment, a sweep on one of the parameters was performed, ranging between usual values found in existing datasets, while keeping the remaining parameters fixed. By doing so, the influence of each parameter in the speed-up and execution time results will be clearly visible. Each experiment also evaluated the use of different numbers of accelerators, so that the impact of the hardware parallelism was also studied.

5.4.2.A Varying number of centers

The first experiments focused on the number of centers used. This is perhaps the most meaningful parameter for the presented solution, given that the datapoint classification task is parallelized throughout the accelerators by splitting the centers among them as equally as possible. It is expected that a lower workload on each accelerator would lead to a smaller execution time. It is also expected that, for an uneven load balancing among the accelerators, the accelerators with the highest load will dictate the execution time for the classification task.

The first test was performed on a dataset with 10126 datapoints. The data dimensionality is 15. A sweep in the number of centers parameter was performed, ranging from 2 centers to up to 100 centers. A burst size of 2500 was used, meaning that a total of 6 bursts will be transferred from the accelerators to the ARM cores. A relatively low number of transactions between hardware and software is preferable, in order for the communication overhead to stay small. Through trial and error, it was found that a number of transactions lower than $\frac{N}{10}$ would suffice. Testing was performed with up to 3 accelerators.

The graphs from figure 5.3 portray the obtained results for both the execution time per iteration and speed-up obtained when comparing with a full software implementation running on a single ARM core. In the speed-up graph, the dashed lines represent the expected values from the analytical model previously detailed, and the full lines represent the actual experimental values.

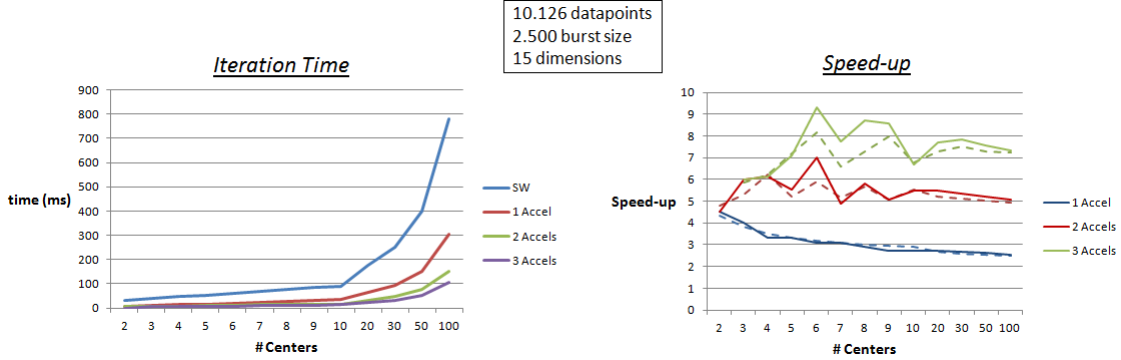


Figure 5.3: Execution time and speed-up results for the 1st experiment

First, we determined which computational entity performs its task the fastest: the ARM cores or the accelerators. In other words, which case and consequent theoretical expression (5.1, 5.2) models the execution time for each of the algorithm runs. According to the expressions 5.1 and 5.2, the ARM cores are faster when $\frac{B}{2}T_{accum} < BT_{classify_{Accel}}$ (case 1). The number of datapoints and dimensions is fixed, and hence the expression can be re-arranged for the number of centers and the number of accelerators. By replacing the time variables by their expressions (given in table 5.6), replacing the fixed parameters by their actual values and re-arranging the expression, it is concluded that the ARM cores are faster than the accelerators when $\lceil \frac{C}{K} \rceil > 2.05$. Only some of the runs with few centers don't fall in this criteria, meaning that, for a set number of accelerators, the addition of more centers will not cause the ARM cores to perform their task slower than the accelerators. This conclusion is expected, since the time taken by the ARMs updating the accumulators, T_{accum} , does not depend on the number of centers, unlike the time taken to classify the datapoints.

Analysing the actual obtained values, a clear timing gain can be seen by just adding one accelerator. Not only the hardware accelerator helps with the classification task, but the implementation of the parallel accumulation updates using both ARMs also greatly contributes to the large execution time decrease. The addition of more accelerators further improves the execution time. In fact, it is expected that the use of more accelerators should have a meaningful impact on the execution time and speed-up results. Since $\lceil \frac{C}{K} \rceil > 2.05$ holds true for most of the runs with higher number of centers, the addition of more accelerators should improve the results, since the iteration time is highly dependent from the time taken by the accelerators in the classification process (5.1). When more than 1 accelerator is being used, an oscillation in the speed-up results is noticeable. The oscillation is a result of the changes in load balancing caused by adding more centers. The speed-up peaks occur when the number of centers is divisible by the number of accelerators and thus the load balancing is perfect. The maximum speed-up observed was 9.3, achieved with 3 accelerators and 6 centers.

In all 3 cases using different numbers of accelerators, it is concluded that the optimal number of centers

delegated per accelerator is around 2 and 3 centers. This optimal value corresponds to the boundary between the two theoretical model cases, where adding more centers per accelerator starts to not be beneficial.

The speed-up graph reveals that the analytical formulae model the progress of the experimental graph fairly well, although in some cases there is some significant error, however, with the largest relative error reaching 13%. Given the way the model was constructed, some inaccuracy is to be expected. Furthermore, the size of the actual dataset may be not large enough to provide accurate timing measurements, as in some cases the execution time is as small as 1 ms. Small timing results may carry a large relative variance, since the latency inherent to the AXI transactions may not be constant throughout several algorithm runs. The next experiment eliminates some of these issues, by using a larger dataset and thus dealing with larger execution time results.

The following experiment performed the same sweep in the number of centers variable, while using a much larger dataset. This time, a dataset of 1 million points was used, with the same 15 dimensions. The burst size was increased to 100000, in order to maintain a relatively slow number of transactions between the accelerators and the ARM cores and thus keeping the data transfer and communication overhead small, while sacrificing some degree of computational parallelism between both entities. Once again, up to 3 accelerators were used. The experimental results are shown in image 5.4.

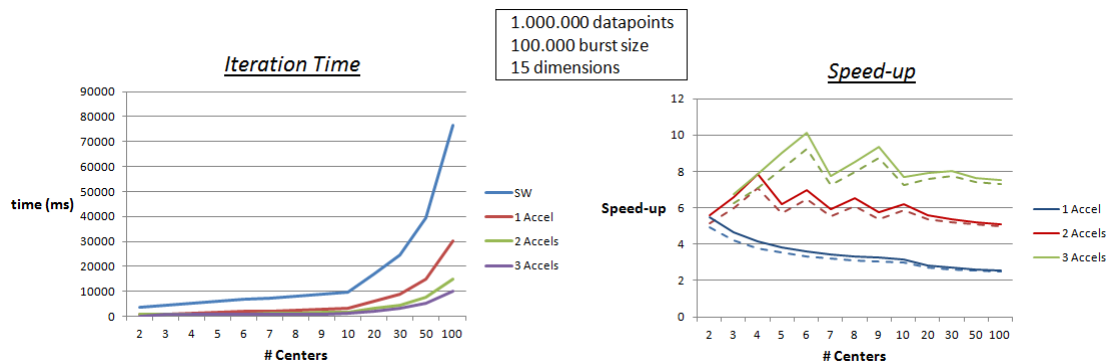


Figure 5.4: Execution time and speed-up results for the 2nd experiment

Regarding which computational element performs its task the fastest, the conclusion is the same as in the previous experiment, since the number of datapoints and burst size parameters do not affect the expression. The ARM cores will perform their task faster than the accelerators when $\lceil \frac{C}{K} \rceil > 2.05$.

Despite the larger execution times, the progress of both graphs is very similar to the previous ones. The addition of a single accelerator and the usage of both ARMs in parallel greatly benefits the execution time, and the further addition of more accelerators further improves the results. The same oscillation due to the load balancing is also observed in the speed-up results. The maximum speed-up achieved used the same parameter configuration as the previous experiment, with 3 accelerators and 6 centers. The speed-up obtained in this case was 10.1. Since the theoretical boundary of both timing cases was the same as in the last experiment, it is no surprise that the maximum speed-up was achieved in the exact same circumstances.

The comparisons between experimental results and the analytical model are also pretty similar. The model accurately tracks the speed-up progress throughout all the results, and the relative error is smaller this time around, peaking at roughly 10%. The increase in the problem size leads to higher execution times, which are favourable for measuring times more accurately.

The first two experiments allowed to further detail how the number of centers parameter influences the timing results for the presented architecture, as well confirm the relationship between number of centers, number of accelerators and execution time, given by the analytical model. Both iteration time graphs show that the time per iteration is linearly dependent to the number of centers per accelerator. The analytical model already stated this dependency in its mathematical expression (more specifically, in $T_{classify_{Accel}}$).

5.4.2.B Varying data dimensionality

The third experiment focused on the data dimensionality: the number of centers was fixed and the sweep was performed on the data dimensionality parameter. Studying the influence of the number of dimensions used on the execution time is important, as clustering problems in *Big Data* analysis focus on datasets with a large number of points and also on very high dimensionality datasets. According to the analytical model, it was expected that the iteration time grows linearly with the increase of dimensionality. The timing estimates for all the steps in the K-means algorithm are linearly dependent from the data dimensionality, so the total iteration time should also be linearly dependent.

The conducted experiment used a dataset of 1 million points, with the same burst size of 100000. The number of centers was fixed at 6. The sweep performed in the data dimensionality ranges from 15 dimensions to up to 55 dimensions. The results obtained are shown in the graphs in figure 5.5.

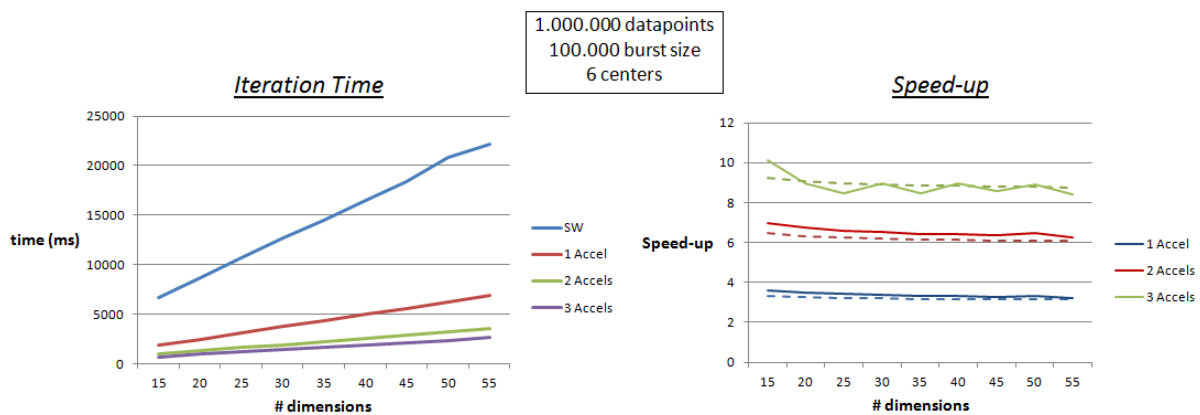


Figure 5.5: Execution time and speed-up results for the 3rd experiment

Once again, by evaluating the boundary expression from the analytical model, $\frac{B}{2}T_{accum} = BT_{classify_{Accel}}$, and by computing the result using the new parameters, it was concluded by computing the expression that, in the experiments using 1 and 2 accelerators, the ARMs were faster than the accelerators (case 1). For the experiments using 3 accelerators, the ARMs became slower (case 2).

The iteration time results confirm the expectations, and a clear linear growth is observed, as the

number of dimensions increases. Once again, the addition of a single accelerator and enabling both ARMs in the accumulator updating task proves to be very beneficial, cutting the iteration time down by more than half. The iteration time growth rate also decreases, in comparison with the software implementation results. Adding more accelerators further diminishes the growth rate and the iteration times improve even further.

For all 3 sweeps using a different number of accelerators, the analytical model indicates a clear stabilization of the achievable speed-up, at roughly 3, 6 and 9, when using 1, 2 and 3 accelerators, respectively. This stable result throughout the entire sweep is explainable once again by the linear dependency between iteration times and data dimensionality. Using more accelerators cuts back on the iteration time, but does not eliminate the linear dependency, existent in both software-only and hardware/software implementations.

The experimental results follow, once again, the progression estimated by the analytical model, with a substantial increase of variance in the results obtained with 3 accelerators. The maximum registered relative error is roughly under 9%.

5.4.2.C Varying number of datapoints

The 4th and final experiment focused on the number of datapoints in the dataset. The number of datapoints in the dataset is tightly connected to the iteration time, as it represents a direct correlation to the amount of repetitive computation done within a single iteration of the algorithm. An increase in the number of datapoints means an increase on the amount of distance calculations, classifications and accumulator/counter updates, and thus a slower assignment step.

Similarly to what happens with the data dimensionality parameter, the analytical model indicates that the iteration time should depend linearly on the number of datapoints. The timing expressions from both tasks in the assignment step (datapoint classification and accumulator/counter updates) vary linearly with the number of datapoints, and thus the total iteration time should vary linearly as well. Using the same reasoning as the one in the previous experiment, the observed speed-up throughout the sweep should also remain constant, and the addition of more accelerators should cut the iteration time throughout all runs within the sweep.

In the proposed solution, changing the number of datapoints should also imply a change in the adequate burst size. Similarly to what was observed and used during the previous experiments, the burst size should not be extremely small nor extremely large. A small burst size would increase the degree of parallelism between the ARMs and the accelerators, but the overhead due to multiple DMA transfer setups and multiple interrupt routine handling would start to become significant. On the other hand, a very large burst size sacrifices the degree of parallelism between the ARMs and the accelerators, since the ARMs would only start their computation after a significant amount of time. During all experiments, and through trial and error, it was observed that keeping the number of transfers relatively low (around 10 bursts) led to the best results.

In this experiment, an exponential sweep on the number of datapoints was performed, from 100 data-

points to 1 million datapoints, with 10 bursts used on each run (meaning that $BurstSize = \frac{\#Datapoints}{10}$). The data dimensionality was fixed to 20 dimensions. The number of centers used was 50. The results for both iteration time and speed-up are portrayed in the graphs of image 5.6.

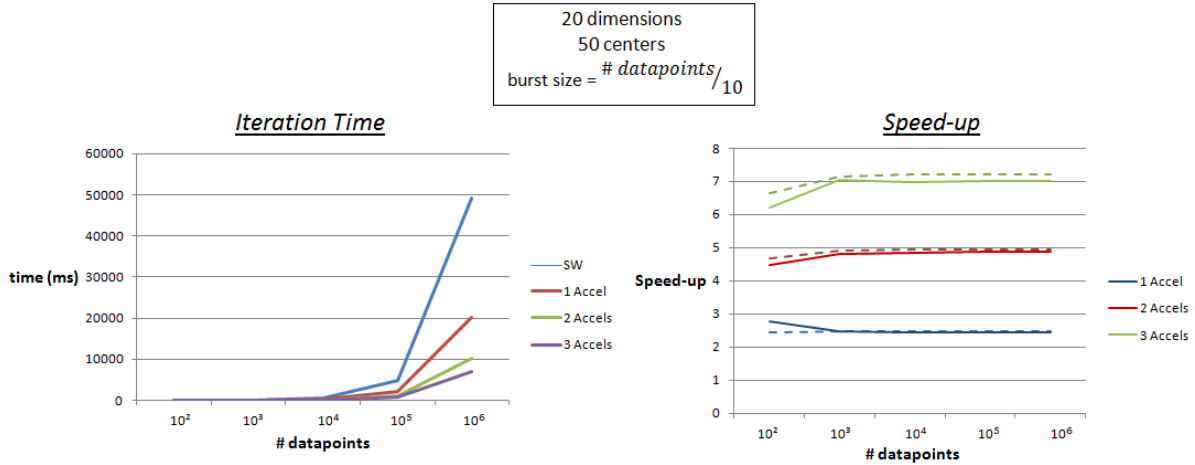


Figure 5.6: Execution time and speed-up results for the 4th experiment

Once again, the analysis started by figuring out which computational entity runs its task the fastest. In this experiment, only the number of datapoints is variable and all the parameters in the boundary expression are fixed. For the selected number of centers and dimensions, the boundary expression indicates that the ARMs are faster than the accelerators (case 1). This means that the usage of several accelerators would be greatly beneficial in the reduction of the execution time.

As expected, the obtained timing results are consistent with a linear dependency among number of datapoints and iteration time. Since an exponential sweep was performed, the iteration time graphs also resemble an exponential function. A consequence of this linear dependency is, once again, the stable speed-up value, as the number of datapoints increases. The conclusion is similar to the one made in the previous experiment: the iteration time for the assignment step for both software-only and hardware/software implementations is linearly dependent from the number of datapoints and thus the resulting speed-up should be constant throughout the sweep.

The experimental results for the speed-up follow the analytical model closely, with the largest relative error, which is roughly 12%, clearly visible in the runs with few datapoints. The cause for the larger difference between analytical and experimental results in the early stages of the sweep is again due to the inaccuracy of the timing measurements, for very short execution times. The experiments with only 100 datapoints has iteration times well under 1 ms. Iterations times as small as these aren't accurately measured easily, as the AXI transactions have variable latencies and several runs of the algorithm with the same parameters can lead to very different measurements.

Also as expected, the increment in the number of accelerators used cuts back on the execution time, with the highest observed speed-up achieved with 6 accelerators, being 7 times faster than the software-only implementation.

5.4.2.D Peak Speed-up

In all the experiments conducted up until this point, it was proven that the number of accelerators used in the computation weighs heavily on the iteration time and speed-up results. This remained true throughout all 4 parameter sweeps performed. However, the experiments only covered a rather small number of accelerators (up to 3). Although an architecture with 3 accelerators may already occupy a significant area in a small SoC device, the larger devices from the Zynq family can hold a much larger number of accelerators. In section 5.3 it was concluded that the most complex device from the Zynq family could hold up to 144 accelerators.

The comparisons done in all experiments between theoretical and experimental values were able to validate the analytical model, as it could track the obtained experimental results rather accurately and with low relative error. The model should, then, be able to provide some predictions on results obtainable using a high number of accelerators, e.g. if using high-end SoC devices.

Unfortunately, given the parallelization through splitting the centers causes 1 setback. Using a high number of accelerators such as 144 would only make sense if there were at least 144 centers. Otherwise there would be accelerators idle throughout the entire execution, since they would not receive any centers to process. Depending on the context of the clustering problem, the maximum number of accelerators used may also be limited by the number of total centers used in that specific application. On the other hand, if a specific application does indeed use such a large number of centers, the use of many accelerators would be extremely beneficial. If, for example, the dataset from the 2nd experiment was to be classified using 144 centers (and thus 1 center per accelerator), the analytical model predicts that a speed-up of up to 165 could be achieved. This speed-up value could potentially be increased even further if the system was forced to operate in the boundary between both timing cases (which, in this experiment, was given by the expression $\lceil \frac{C}{K} \rceil > 2.05$).

These predictions are able to show the absolute best outcomes possible, when it comes to the speed-up analysis. The dependency on the number of centers used in the clustering problem, which varies from application to application, ends up being a major factor when dictating the maximum speed-up achievable. This is the big disadvantage of the parallelization through splitting the centers. If the solution relied on splitting the datapoints instead, the main conditioning factor would instead be the bandwidth (as discussed in section 5.2).

5.5 Floating-Point Performance

This section analyses the performance measured in Floating-Point Operations per Second (FLOPS). This performance measurement may provide a rough comparison with implementations on very different platforms, e.g. on multi.core GPPs or on GPUs.

The proposed architecture relies on both hardware (accelerators) and software (ARMs) to perform the needed computations. For both entities, the peak performance values may be unequivocally estimated. The ARM processors operate at a frequency of 650 Mhz, and it is known that the floating-point units of both processors are able to process one floating-point operation per cycle [ARM]. This means that each

of the ARM processors can potentially achieve a peak performance of 650 MFLOPS. Each accelerator has three floating-point cores, a subtracter, an accumulator, and a comparator. All three are able to produce one result per cycle (although the *Minimum Distance* comparison only produces a valid result every $\lceil \frac{C}{K} \rceil$ cycles). So, with the operating hardware frequency of 100 Mhz, a single accelerator peak performance is 300 MFLOPS.

In each iteration, the K-means algorithm computes a fixed number of floating-point operations (this number does not vary from iteration to iteration). For each iteration of the algorithm, the following floating-point operations need to be performed:

- Distance Calculation

NCD subtractions

NCD accumulations

- Datapoint Classification

NC comparisons

- Accumulator/Counter Updates

ND accumulations

- Center Updates

CD divisions

In total, $N(2CD + C + D) + CD \approx N(2CD + C + D)$ floating-point operations are performed per iteration.

Given the iteration times from the previous experiments, the performance values can be obtained by simply dividing the number of floating-point operations by the time taken in each iteration. The measured iteration times were then re-used, and a new graph for each of the sweeps performed was created with respect to the performance values. The values obtained for the sweeps in the number of centers are portrayed in figure 5.7.

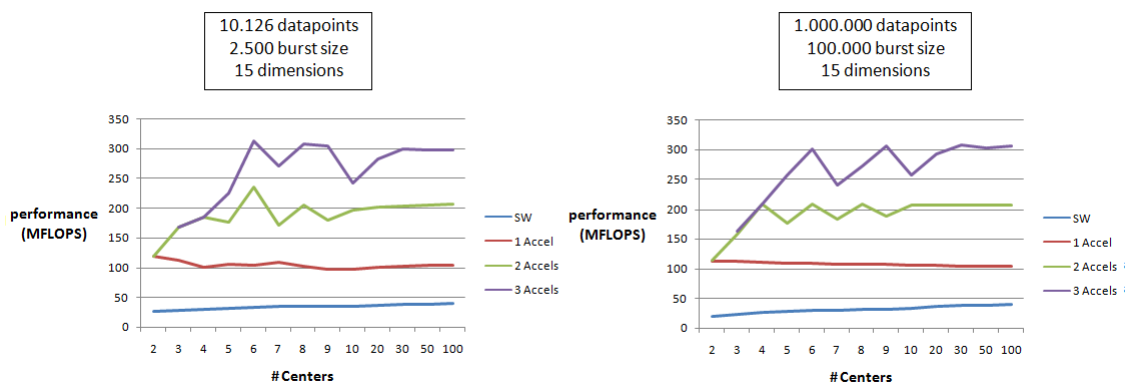


Figure 5.7: Performance values for the 1st and 2nd experiments

As shown, both software and hardware/software implementations stabilize their performance values, as the problem size increases. The software implementation stabilizes at around 40 MFLOPS. As expected, the floating-point performance of the hardware/software implementations significantly increases, as more accelerators are introduced. For up to 3 accelerators, a maximum performance value of roughly 300 MFLOPS can be achieved.

In the 3rd experiment the sweep was performed in the data dimensionality variable. Figure 5.8 shows the resulting performance values.

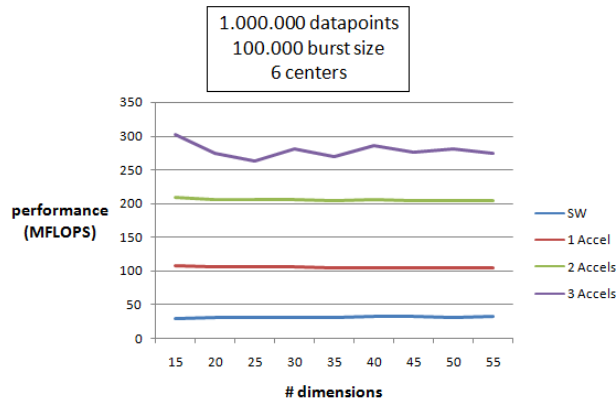


Figure 5.8: Performance values for the 3rd experiment

From analysing the resulting graph, it can be concluded that the change in number of dimensions has far less impact on the resulting system performance. In all cases except the sweep involving 3 accelerators, the performance results were constant throughout the entire sweep. However, the performance values seem to stabilize to roughly the same values as in the previous experiments, as the problem size increases. This time around, the observed floating-point performance also comes close to 300 MFLOPS, using 3 accelerators, however slightly decreases throughout the sweep, stabilizing at around 280 MFLOPS.

The final experiment focused on the number of datapoints. The results are shown in figure 5.9.

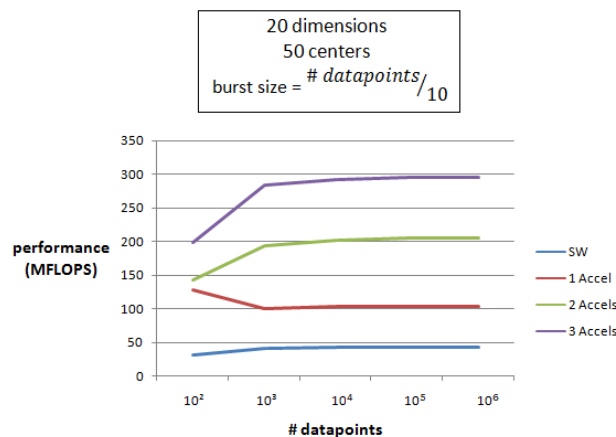


Figure 5.9: Performance values for the 4th experiment

Similarly to what happened in the dimensionality sweep, the performance values seem to be resistant to the variable number of datapoints. The values measured using small datasets vary considerably from

the rest of the results, mainly due to inaccuracies measuring very small execution times. Once again, the performance values stabilize as the dataset size grows, reaching roughly the same static values as in the previous experiments. A maximum observed performance of 295 MFLOPS was observed, using 3 accelerators.

Given all the performance results, a final remark should be made, regarding the overall system's efficiency. It was noticed that, for every experiment, the performance obtained stabilized at roughly the same values in all cases. These values can now be compared with the estimated peak values, in order to evaluate the system's performance efficacy. The full software implementation revealed to have the lowest efficiency. For a stabilized performance value of around 40 MFLOPS, the efficiency obtained is roughly 6%. As more accelerators are introduced in the system, the efficacy starts to increase slightly. With just 1 accelerator, the efficiency increases to 6.6%. With just 3 accelerators, the efficiency reaches more than double of the efficiency of the software implementation, reaching up to 13.9%.

5.6 Summary

This chapter presented the evaluation results of the proposed custom hardware/software solution. The throughput of the hardware blocks in each accelerator was analysed, in order to formulate the throughput of the entire accelerator block. It was concluded that, using K accelerators, the hardware is capable of producing one result every $2D\lceil\frac{C}{K}\rceil$ cycles.

The available bandwidth between hardware and software domains (PS/PL bandwidth) was measured. The bandwidth needed by the architecture proved to be one of the major upsides of this solution. Since the computation is parallelized by splitting the centers through the accelerators, the bandwidth required does not increase with the addition of more accelerators, due to the hardware datapoint broadcast. This makes the solution highly scalable in the hardware domain. The only downside of the center splitting is that the number of accelerators is limited by the number of centers. For clustering problems using few centers, the number of accelerators possible to use would also be low, and the potential maximum speed-up could be not as good as a solution with datapoint splitting.

The measured bandwidths were close to the maximum bandwidth available in each of the AXI HP ports. In both read and write transactions, over 90% of the total available bandwidth was achieved, meaning that the system was capable of using almost all of the available bandwidth in each of the used ports.

The area occupied by the architecture consists of a base resource usage, independent from the number of accelerators used, and of a linear and logarithmic incremental usage dependent of the number of accelerators. It was concluded that Zynq-7020 device can fit up to 24 accelerators, while the most complex Zynq device can fit up to 144 accelerators.

The timing analysis reported the speed-up obtained in several runs of the algorithm, compared to a software single core implementation. The analysis started by formulating an analytical model for the execution times, for any given dataset and number accelerators. This model was then validated experimentally. Several experiments were conducted. In each experiment, a sweep in one of the parameters was

made, while keeping the remaining parameters fixed. For only 3 accelerators, the maximum speed-up registered was 10.1. However, with the values from the resource usage analysis and the theoretical model, it was predicted that the designed architecture could potentially reach speed-ups of 165, and higher, on high-end devices.

Finally, a floating-point performance analysis was performed. The system performance was measured in FLOPS, according to the experimental timing results. It was concluded that, with the addition of more accelerators, the architecture's efficiency is able to increase from the software implementation, although the maximum observed floating-point performance is still far from the peak values. An efficiency of 13.9% was observed for the architecture with 3 accelerators.

6

Conclusions

Contents

6.1	Future work	71
-----	-----------------------	----

The work developed and presented in this master’s thesis focused on the research and development of custom hardware/software architecture to efficiently execute clustering applications and the K-means algorithm in particular. The simple clustering concept adopted by the K-means algorithm and its popularity in several scientific areas made it a clustering algorithm of choice for implementation in this work.

With the clustering problem sizes growing rapidly and the emerging concepts of *Big Data* analysis and real-time clustering, timing constraints start to become increasingly relevant, as the conventional algorithms are unable to compute the results in an acceptable amount of time. In recent years, several solutions tackling the large computational times were proposed and discussed in the scientific community. Previous solutions cover different acceleration techniques, from parallel and distributed computing approaches up to complete custom hardware architectures. Some proposals also have tried to take advantage of the computational capacity of modern GPUs. With the recent introduction of FPGA SoC devices that offer both software programmability and hardware resources, hardware/software architectures can be promising as a way to further accelerate the intended clustering applications.

In this thesis, a hardware/software architecture for executing the K-means clustering algorithm was proposed and demonstrated on a Xilinx Zynq-7010 device. The solution takes advantage of the pre-built ARM CPUs as well as of the available FPGA resources and partitions the computation among hardware and software domains. The K-means algorithm allowed that both hardware and software computations performed their tasks in parallel, further contributing to the possible algorithm acceleration. The hardware domain became responsible for the most computationally demanding task of the K-means algorithm: the assignment step, consisting on the distance calculation and datapoint classification. Custom hardware blocks were designed and an overall accelerator block was developed. The independence between datapoints in the assignment step allowed for the computational task to be parallelized among several identical hardware accelerators. At this stage, two possible parallel solutions were analyzed. Computation could be parallelized by splitting the dataset through several accelerators, or by splitting the centers. The latter solution was selected, as it resulted in a highly scalable hardware solution: the amount of hardware/software bandwidth remains unaffected, regardless of the number of accelerators used. The only downside relies on the number of accelerators used: the maximum number of accelerators possible is limited by the number of centers used in the clustering problem.

After implementation, extensive testing was performed and the hardware/software architecture was fully evaluated. The evaluation covered any possible given dataset and number of accelerators. Hardware throughput was the first metric analysed, followed by the (accelerator independent) PS/PL bandwidth. The analysis of hardware resource consumption allowed to estimate how many accelerators can fit in any device and thus to predict the amount of parallelism and speed-up that can be achieved using this architecture in higher-end devices. An analytical model was derived from the timing analysis, able to track the speed-up progress for any given dataset and number of accelerators. The model was validated with experimental results and was shown to be quite accurate. The maximum experimental speed-up observed, using up to 3 accelerators, was 10.1, although the model predicts that a much higher speed-up can be achieved in larger devices. Finally, a floating-point performance analysis provided efficiency results

for the architecture, given the experimental timing results.

Overall, the devised solution produced very satisfactory accelerating results. Unlike most of the designs presented previously, this architecture proved to be capable of handling any kind of dataset (as long as it fits in the device’s memory) and achieve promising speed-ups, when a large number of centers is used (a scenario that is common in *Big Data* analysis).

6.1 Future work

Despite the satisfactory results, several other variants of the architecture may be considered and some improvements may be introduced. As stated previously, the presented solution is scalable in the number of accelerators used, due to the fixed bandwidth usage. The Zynq SoC contains several AXI ports that connect the hardware and software domains, and the possibility to use more bandwidth than the total used by this architecture should be explored. One proposed variant would be to choose a dataset splitting, rather than center splitting. Parallelizing the computation throughout the accelerators by splitting the dataset would remove the dependency between number of accelerators and number of centers, but the available bandwidth would start to be an issue for a large number of accelerators. This alternative solution would be ideal for large datasets, with many dimensions, that are meant to be classified in just a handful of clusters.

The software parallelization can also be improved. The proposed solution only relies on the two ARM cores to perform the accumulator/counter updates. For the cases where the ARM cores perform their tasks slower than the accelerators perform theirs, further parallelizing the accumulator/counter updates would be very beneficial. If there are no other *hard-core* CPUs in the device, delegating the updates to the hardware domain could be an improvement. A set of hardware accumulators can be used per center, or even shared among several centers. Another possible alternative would be to rely on *soft-core* processors, such as the Microblaze, to aid in the accumulator/counter updating task. Instead of splitting the task evenly among *hard-core* and *soft-core* processors, a larger load should be assigned to the *hard-core* processors, since they usually operate at much higher frequencies.

References

- [Alt] Altera. FPGA devices. Available: <https://www.altera.com/products/fpga/overview.html>.
- [AMY09] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, pages 126–131. IEEE, 2009.
- [ARM] ARM. Cortex-A9 Floating Point Unit Technical Reference Manual. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0408i/DDI0408I_cortex_a9_fpu_r4p1_trm.pdf.
- [CMSS07] Shuai Che, Jiayuan Meng, Jeremy W Sheaffer, and Kevin Skadron. A performance study of general purpose applications on graphics processors. In First Workshop on General Purpose Processing on Graphics Processing Units, page 10, 2007.
- [ELTS01] Mike Estlick, Miriam Leeser, James Theiler, and John J Szymanski. Algorithmic transformations in the implementation of k-means clustering on reconfigurable hardware. In Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays, pages 103–110. ACM, 2001.
- [EpKSX96] Martin Ester, Hans peter Kriegel, Jörg S, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.
- [Fab94] V. Faber. Clustering and the continuous k-means algorithm. pages 138–144. Los Alamos Science, 1994.
- [FRCC08] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H Campbell. A parallel implementation of k-means clustering on gpus. In PDPTA, volume 13, pages 212–312, 2008.
- [GFM⁺03] Maya Gokhale, Jan Frigo, Kevin Mccabe, James Theiler, Christophe Wolinski, and Dominique Lavenier. Experience with a hybrid processor: K-means clustering. The Journal of Supercomputing, 26(2):131–148, 2003.
- [HBES11] Hanaa M Hussain, Khaled Benkruid, Ahmet T Erdogan, and Huseyin Seker. Highly parameterized k-means clustering on fpgas: Comparative results with gpps and gpus. In Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on, pages 475–480. IEEE, 2011.

- [HBSE11] Hanaa M Hussain, Khaled Benkrid, Huseyin Seker, and Ahmet T Erdogan. Fpga implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on, pages 248–255. IEEE, 2011.
- [KBA13] Jithin Sankar Sankaran Kutty, Farid Boussaid, and Abbas Amira. A high speed configurable fpga architecture for k-mean clustering. In Circuits and Systems (ISCAS), 2013 IEEE International Symposium on, pages 1801–1804. IEEE, 2013.
- [KS14] Mohammad Kakooei and Hadi Shahriar Shahhoseini. A parallel k-means clustering initial center selection and dynamic center correction on gpu. In Electrical Engineering (ICEE), 2014 22nd Iranian Conference on, pages 20–25. IEEE, 2014.
- [Kuc14] T. Kucukyilmaz. Parallel k-means algorithm for shared memory multiprocessors. Journal of Computer and Communications, (2):15–23, 2014.
- [Lav00] Dominique Lavenier. Fpga implementation of the k-means clustering algorithm for hyperspectral images. In Los Alamos National Laboratory LAUR. Citeseer, 2000.
- [LHC05] Wei-Chuan Liu, Jiun-Long Huang, and Ming-Syan Chen. Kacu: k-means with hardware centroid-updating. In Emerging Information Technology Conference, 2005., pages 3–pp. IEEE, 2005.
- [Ord03] Carlos Ordóñez. Clustering binary data streams with k-means. In Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery, pages 12–19. ACM, 2003.
- [PDF09] D. Pettinger and G. Di Fatta. Scalability of efficient parallel k-means. In E-Science Workshops, 2009 5th IEEE International Conference on, pages 96–101, Dec 2009.
- [UCI] UCI. Machine learning repository. Available: <https://archive.ics.uci.edu/ml/datasets.html>.
- [WH07] G.A. Wilkin and Xiuzhen Huang. K-means clustering algorithms: Implementation and comparison. In Computer and Computational Sciences, 2007. IMSCCS 2007. Second International Multi-Symposiums on, pages 133–136, Aug 2007.
- [WL07] Xiaojun Wang and Miriam Leeser. K-means clustering for multispectral images using floating-point divide. In Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on, pages 151–162. IEEE, 2007.
- [Xila] Xilinx. AXI DMA v7.1 LogiCORE IP Product Guide. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.
- [Xilb] Xilinx. AXI reference guide. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

- [Xilc] Xilinx. Zynq-7000 All Programmable SoC device family. Available: <http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>.
- [Xild] Xilinx. Zynq-7000 All Programmable SoC system design. Available: <http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/developer-tools/system-design.html>.
- [ZG09] Mario Zechner and Michael Granitzer. Accelerating k-means on the graphics processor via cuda. In Intensive Applications and Services, 2009. INTENSIVE'09. First International Conference on, pages 7–15. IEEE, 2009.
- [ZWH⁺11] Jing Zhang, Gongqing Wu, Xuegang Hu, Shiyong Li, and Shuilong Hao. A parallel k-means clustering algorithm with mpi. In Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on, pages 60–64, Dec 2011.

