THE UNIVERSITY *of* EDINBURGH

Dynamically and Partially Reconfigurable Hardware Architectures for High Performance Microarray Bioinformatics Data Analysis

Hanaa M. Hussain

A thesis submitted for the Degree of Doctor of Philosophy

The University of Edinburgh

September 2012

Declaration of Originality

I hereby declare that the research reported in this thesis and the thesis itself was composed and originated entirely by myself in the School of Engineering at The University of Edinburgh.

Hanaa M. Hussain

Sep 2012

Edinburgh, U.K.

Acknowledgements

I would like to thank all people who have supported me throughout my PhD study and who have contributed in anyway to the completion of this thesis. At first I would like to acknowledge the support and guidance of my supervisor Dr. Khaled Benkrid and thank him for his invaluable efforts, suggestions, and profound expertise which helped in transforming my knowledge in the subject of reconfigurable hardware design. I would like to also thank my co-first supervisor, Dr. Huseyin Seker from De Montfort University, for his supervision and suggestions throughout my PhD study. I would like to also thank Dr. Ahmet Erdogan for his support and suggestions.

In addition, I would like to acknowledge the support of my employer, the Public Authority for Applied Education and Training (PAAET) in Kuwait, and thank them for the financial support of this project. I would like to also thank the Cultural Office of the Embassy of the State of Kuwait in London for all the administrative support and guidance they have provided during my stay in the U.K.

I would like to thank all members of my family for all their support and encouragement throughout the period of my study, without them I would have never been able to reach this level of knowledge. At last, thanks to all my colleagues at SLIg group.

Abstract

The field of Bioinformatics and Computational Biology (BCB) is a multidisciplinary field that has emerged due to the computational demands of current state-of-the-art biotechnology. BCB deals with the storage, organization, retrieval, and analysis of biological datasets, which have grown in size and complexity in recent years especially after the completion of the human genome project. The advent of Microarray technology in the 1990s has resulted in the new concept of high throughput experiment, which is a biotechnology that measures the gene expression profiles of thousands of genes simultaneously. As such, Microarray requires high computational power to extract the biological relevance from its high dimensional data. Current general purpose processors (GPPs) has been unable to keep-up with the increasing computational demands of Microarrays and reached a limit in terms of clock speed. Consequently, Field Programmable Gate Arrays (FPGAs) have been proposed as a low power viable solution to overcome the computational limitations of GPPs and other methods.

The research presented in this thesis harnesses current state-of-the-art FPGAs and tools to accelerate some of the most widely used data mining methods used for the analysis of Microarray data in an effort to investigate the viability of the technology as an efficient, low power, and economic solution for the analysis of Microarray data. Three widely used methods have been selected for the FPGA implementations: one is the un-supervised K-means clustering algorithm, while the other two are supervised classification methods, namely, the K-Nearest Neighbour (K-NN) and Support Vector Machines (SVM). These methods are thought to benefit from parallel implementation. This thesis presents detailed designs and implementations of these three BCB applications on FPGA captured in Verilog HDL, whose performance are compared with equivalent implementations running on GPPs. In addition to acceleration, the benefits of current dynamic partial reconfiguration (DPR) capability of modern Xilinx' FPGAs are investigated with reference to the aforementioned data mining methods.

Implementing K-means clustering on FPGA using non-DPR design flow has outperformed equivalent implementations in GPP and GPU in terms of speed-up by two orders and one order of magnitude, respectively; while being eight times more power efficient than GPP and four times more than a GPU implementation. As for the energy efficiency, the FPGA implementation was 615 times more energy efficient than GPPs, and

31 times more than GPUs. Over and above, the FPGA implementation outperformed the GPP and GPU implementations in terms of speed-up as the dimensionality of the Microarray data increases. Additionally, the DPR implementations of the K-means clustering have shown speed-up in partial reconfiguration time of ~5x and 17x over full chip reconfiguration for single-core and eight-core implementations, respectively.

Two architectures of the K-NN classifier have been implemented on FPGA, namely, A1 and A2. The K-NN implementation based on A1 architecture achieved a speed-up of ~76x over an equivalent GPP implementation whereas the A2 architecture achieved ~68x speed-up. Furthermore, the FPGA implementation outperformed the equivalent GPP implementation when the dimensionality of data was increased. In addition, The DPR implementations of the K-NN classifier have achieved speed-ups in reconfiguration time between ~4x to 10x over full chip reconfiguration when reconfiguring portion of the classifier or the complete classifier.

Similar to K-NN, two architectures of the SVM classifier were implemented on FPGA whereby the former outperformed an equivalent GPP implementation by ~61x and the latter by ~49x. As for the DPR implementation of the SVM classifier, it has shown a speed-up of ~8x in reconfiguration time when reconfiguring the complete core or when exchanging it with a K-NN core forming a multi-classifier.

The aforementioned implementations clearly show FPGAs to be an efficacious, efficient and economic solution for bioinformatics Microarrays data analysis.

Table of Contents

List of Figures

List of Tables

Acronyms and Abbreviations

AML	Acute Myeloid Leukaemia
ALL	Acute Lymphatic Leukaemia
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ASMBL™	Advanced Silicon Modular Block
BCB	Bioinformatics and Computational Biology
BRAM	Block RAM
BW	Bandwidth
CAD	Computer-Aided Design
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CQP	Constrained Quadratic Programming
CUDA	Computer Unified Device Architecture
DCM	Digital Clock Manager
DP	Distance Processing unit
DDR	Double Data Rate
DMA	Direct Memory Access
DPE	Distance Processing Element
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processing
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
ERPROM	Erasable Reprogrammable Read Only Memory
FDA	Food and Drug Administration
FIFO	First-in, First-out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPP	General Purpose Processor
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
HGP	Human Genome Project
HLL	High Level Language
HLS	High Level Synthesis

HPC	High Performance Computing
HPRC	High Performance Reconfigurable Computing
HWICAP	Hardware Internal Configuration Access Port
I/O	Input/Output
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
IOB	Input/Output Block
IP	Intellectual Property
IRE	Internal Reconfiguration Engine
J	Joule
JTAG	Joint Test Action Group
K-NN	K-Nearest Neighbour
KNN	K-Nearest Neighbourhood finder kernel
LC	Logic Cell
LE	Learning Element
LUT	Loop-Up Table
MAC	Multiply-Accumulate Circuit
Mbps	Mega Bits Per Second
MCUPS	Million Cell Updates Per Second
MPx	Mega Pixel
MRI	Magnetic Resonance Imaging
MSB	Most Significant Bit
N/A	Not Available
NGC	Xilinx Native Generic Database
NRE	Non-Recurring Engineering
PAL	Programmable Array Logic
PAR	Place-And-Route
Pblock	Partition Block
PCA	Principal Component Analysis
PCI	Peripheral Component Interconnect
PE	Processing Element
PLL	Phase Locked Loop
PM	Personalised Medicine
PMT	Photomultiplier Tube
PR	Partial Reconfiguration
RAM	Read Access Memory
RGB	Red Green Blue colour model
RISC	Reduced Instruction Set Computing
RM	Reconfigurable Module
ROM	Read Only Memory

RP	Reconfigurable Partition
RTL	Register Transfer Level
SDRAM	Synchronous Dynamic RAM
SMO	Sequential Minimal Optimisation
SoC	System on Chip
SOM	Self Organizing Map
SRAM	Static RAM
SV	Support Vector
SVE	Support Vector Element
SVM	Support Vector Machine
TIFF	Tagged Image File Format
UCF	User Constraints File
VLSI	Very Large Scale Integration
W	Watt
WL	Wordlength

Published Papers

Conferences

H. Hussain, K. Benkrid, H. Seker, and A. Erdogan, "FPGA Implementation of K-means Algorithm for Bioinformatics Application: An Accelerated Approach to Clustering Microarray Data," in *Proc. of the 2011 NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*,San Diego, CA, USA, June 6-9, 2011, pp.248-255.

H. Hussain, K. Benkrid, H. Seker, and A. Erdogan, "Highly Parametrized K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs," in *Proc.of the 2011 Int. Conf. on Reconfigurable Computing and FPGAs (ReConFig11)*,Cancun, Mexico, Nov. 30-Dec. 2, 2011, pp.475–480.

H. Hussain, K. Benkrid, and H. Seker, "An Adaptive Implementation of a Dynamically Reconfigurable K-Nearest Neighbour Classifier on FPGA," in *Proc.* of *the 2012 NASA/ESA Conf. on Adaptive Hardware and Systems (AHS)*, Nuremberg, Germany, June 25-28, 2012, pp.xx-xx (Accepetd).

H. Hussain, K. Benkrid, C. Hong, and H. Seker, "An Adaptive FPGA Implementation of Multi-core K-Nearest Neighbour Ensemble Classifier Using Dynamic Partial Reconfiguration" in *Proc.* of *2012 Int. Conf. on Field Programmable Logic and Applications (FPL12)*, Oslo, Norway, Aug 29-31, 2012, pp.xx-xx (Accepetd).

C. Hong, K. Benkrid, X. Iturbe, and **H. Hussain**, "Efficient Run-time System Support for High Perfromance Reliable Reconfigurable Systems," *Int. Conf. on Reconfigurable Computing and FPGAs (ReConFig12)*,Cancun, Mexico, Dec. 5-7, 2012. (in Press).

Journals

H. Hussain, K. Benkrid, C. Hong, and H. Seker, "Novel Dynamic Partial Reconfiguration Implementation of K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs," *Int. J. of Reconfig. Comput* .(in Press).

Chapter-1

Introduction and Motivation

1 Introduction and Motivation

The field of Bioinformatics and Computational Biology (BCB) is an interdisciplinary field that has emerged as a result of the increased computational demands in biology. These demands are induced by the sophisticated and high throughput nature of current state-of-the-art biotechnologies. The announcement of the completion of the Human Genome Project (HGP) on June of 2000 has altered the fields of biology and medicine significantly. HPG has sparked the beginning of the era of molecular biology and medicine as a consequence of sequencing three billion bases forming the genome of human. Since the completion of HGP, the field of bioinformatics has been flourishing as an interdisciplinary field forming an intersection between molecular biology and computer science. Bioinformatics is defined as the science or techniques of organising, storing, retrieving, and analysing biological data resulting from genomics and proteomics. Genomics is a subspecialty in molecular biology dealing with the genome of an organism. A genome is the DNA of the whole organism consisting of genes, and is made-up of millions or billions base pairs depending on the type of organism and species. Genes are so important because they encode the synthesis of proteins which determine the look of the organism, the behaviour, food/drug metabolism, response to environmental factors, health conditions and illnesses. Proteomics on the other hand focuses on protein structures, interactions and functionality. Proteins are so important because they determine the aforementioned characteristics of an organism [1]-[2].

Recent advances in biotechnologies and molecular biology which are attempting to leverage HGP data have resulted in techniques generating enormous amounts of data such as DNA and protein sequences, gene expression profiles, and protein-protein interactions. Those genomic data have led to computational analyses such as bio-sequence alignment, phylogenetic trees, molecular dynamics, genome mining and classification. Such methods are contributing to an uncontrollable growth in the size of biological data. As a consequence of data growth, scientists are facing computational problems related to higher execution times and larger power consumption. Additionally, the availability of the aforementioned molecular data has increased the complexity of the biological questions that can now be asked by scientists which is calling for the development of new mathematical models or algorithms to answer such questions. These demands have led to wider integration of principles of engineering and computer science into molecular biology to help in converting biological experimental data into biomedical knowledge and hypotheses, BCB has emerged to cater to these demands [1]-[2].

Furthermore, genomics medicine, the application of genomics data to healthcare has been gathering the interest of the health care industry in recent years. Genomics medicine in concerned about leveraging genomics data in disease diagnosis and patient care, which has prompted new developments in pharmacogenomics, a field concerned about the development of new treatments and drugs making use of genomic data produced by current state-of-the-art biotechnologies. These developments have widened the scope of BCB and led to a plethora of complex problems associated with high volume of data and intensive computational demands [3].

Today's BCB definition has distinctively isolated the roles of Bioinformatics and Computational Biology, whereby the former deals with the research and development of tools and databases to be used for acquiring, storing, archiving, and analysing biological data while the other deals with the development or application of those tools and databases in inferring biological knowledge. Additionally, Computational Biology deals with developing new mathematical models or new algorithms to help solve new and complex problems [1].

As a result of the aforementioned developments, sequential computing is unable to keep-up with the growing computational demands mainly due to physical limits in terms of power consumption associated with higher clock speeds. As such, BCB turned towards parallel computing with the deployment of multi-processor systems and Grid computers to tackle complex biological problems. Applying such methods has been successful in handling emerging BCB computational demands to some extent, but at the same time was accompanied with higher purchase, power and operation cost. Consequently, current research in BCB is leveraging GPUs and FPGAs to circumvent the limitations of sequential GPPs and expensive parallel solutions [4].

DNA Microarray or the DNA chip is one of the high throughput biotechnologies used to measure the gene expression profiles of tens of thousands of genes simultaneously. Gene expression is basically the process of retrieving and interpreting the genetic code embedded into genes for the synthesis of proteins or other molecules. Microarrays measure gene expressions by checking for the presence of sequences in DNA samples hybridised to the Microarrays. A single DNA chip can have 100,000 arrays each having a diameter of ~150 µm. Theoretically it can be stated that a single Microarray chip runs several thousands of experiments in parallel resulting into large amount of genomic data [2]. The explosive amount of data made available from various DNA chips which have been archived in many freely accessible databases through the World Wide Web (e.g., GenBank, EMBL, Ensembl, and DDBJ) is introducing growing challenges in BCB. Complex studies integrating results

from multiple Microarray experiments are contributing to considerable data growth leading to proliferation in computational demands. This is imposing requirement for new reliable data analysis tools and computing systems to tackle the added complexity. Microarrays span a plethora of applications which include investigating cellular states, identifying new genes and pathways, disease diagnosis, drug discovery, pathogen resistance and disease prediction [2]. However, current BCB technologies impede the exploitation of Microarray data in the formulation of complex biological questions and limit its applications to smaller volume data and relatively simple biological questions. As such, adopting new technologies is inevitable for overcoming the curse of data growth in biology.

1.1 Thesis Objectives and Contributions

To unlock the powerful potential of Microarrays in biological and clinical settings, new computing platforms that are capable of overcoming the limitations of current computing methods, namely, GPPs and supercomputers need to be adopted. In this thesis, FPGAs are proposed as state-of-the-art high performance reconfigurable computing (HPRC) platform for the analysis of Microarray data. Analysing Microarray data is an application of BCB spanning a wide range of un-supervised and supervised methods used to infer biologically or clinically relevant results from Microarrays. Un-supervised methods such as data clustering deal with large datasets without prior knowledge about them while supervised methods deal with data with prior knowledge such as knowing the disease state or class membership of some of the data points. A typical un-supervised Microarray problem can include 18,000 genes and 40 features or dimensions while typical supervised Microarray problem include smaller member of genes and larger samples (e.g., Leukemia data has 3,571 genes and 72 sample, Lymphoma data has 4,682 genes and 81 samples). Note that the word feature or dimension represents different Microarray samples in un-supervised problem while in supervised problem it represents genes. In this thesis, three widely used methods for the analysis of Microarray data are implemented on FPGAs, which are suitable for parallel implementation, namely:

- K-means clustering.
- K-Nearest Neighbour (K-NN) classification.
- Support Vector Machines (SVM) classification.

When dealing with large datasets without any prior knowledge, several statistical and pattern recognition methods are usually applied to reduce the size of data and identify distinct patterns that could indicate certain relationships between data. However, when it comes to dealing with genetic or genomic data, selecting a subset from the original data might impede some of the significant features and degrade the integrity of the findings. As a result, using large Microarray datasets is desirable for asking particular clinical or research questions such as identifying genes related to specific diseases or conditions e.g., cancer, Parkinson, Alzheimer, or others. The purpose of these studies is to define signature genes associated with some diseases that will help in developing diagnostic or prognostic tests for them. K-means clustering is one of the most popular clustering methods used for identifying signature genes from large Microarray datasets. Additionally, some biological studies on evolutions attempt to integrate genome data of different species to study or establish relationships between those species. Following the identification of gene signatures, supervised methods such as K-NN and SVM are applied to the resulted data to form classifiers. Such classifiers are then trained to predict the classification of unknown samples [6]-[7].

Unfortunately, when data are large and are highly dimensional as in the case of Microarray data, the aforementioned methods become computationally intensive and expensive. The latter is particularly true due to the exponential growth in the requirement of some hardware resources e.g., memory, and the prolonged execution time leading to higher power consumption. As such, this thesis proposes the use of FPGA to accelerate the execution times of the aforementioned applications, and to overcome the limitations of current computational methods in terms of power and energy consumptions. Additionally, the thesis investigates the benefits and roles of applying dynamic partial reconfiguration (DPR) to the FPGA implementations presented in this thesis to determine whether DPR add any benefits to BCB applications in terms of flexibility and performance. The detailed objectives of this thesis are therefore as follows:

- Design and implementation of highly parameterised and efficient architectures of the K-means clustering on FPGAs targeting Microarrays. Current state-of-the-art DPR technology is then to be used for achieving high performance and flexibility.

- Design and implementation of highly parameterised, flexible and efficient architectures of the K-NN classification on FPGAs targeting Microarray data. Here also DPR technology is to be used for achieving high performance and flexibility.
- Design and implementation of highly parameterised, flexible and efficient architectures of the SVM classifier on FPGAs targeting Microarrays. Similar to K-NN, several variations of the SVM classifier based on DPR will be considered.
- Investigate the use of DPR to dynamically reconfigure specific regions within the FPGA with different classifiers.
- Evaluate the use of FPGAs as HPRC platform in BCB applications compared with other platforms, namely, GPPs and GPUs.

In sum, the ultimate aim of this thesis is to use the aforementioned case studies to determine whether FPGAs are viable economic and high performance computing platform for BCB applications.

1.1.1 Contributions

The FPGA implementations of the K-means clustering, K-NN and SVM classification have led to the following contributions:

- A total of five adaptive architectures of the K-means clustering were created including non-DPR and DPR based architectures. The proposed implementations have outperformed equivalent implementations in GPP and GPU in terms of execution time and power consumption, whereby FPGA achieved a speed up of two orders of magnitude over GPP and one order of magnitude over GPU. In addition, the FPGA implementation was 8x more power efficient than GPP and 4x more power efficient than GPU leading to energy efficiency of 615x and 31x, respectively. Moreover, the novel DPR architectures have increased the flexibility of the K-means implementations with respect to server deployment whereby the implementations permit the modification of single or multiple K-means cores without re-booting the FPGA or interrupting other tasks running on it, achieving up to 17x speed-up in partial reconfiguration time over full chip reconfiguration.

- A total of eight adaptive architectures of the K-NN classification were created including non-DPR and DPR based architectures. The proposed implementations have outperformed GPPs in terms of execution time by up to 76x. In addition, the novel DPR architectures have shown increased level of flexibility in core modification, continued operation of other tasks placed on the FPGA, and between 4x to 10x speed-up in partial reconfiguration time over full device reconfiguration.
- A total of six adaptive architectures of the SVM classification were created including non-DPR and DPR based architectures. The proposed implementations have outperformed GPPs in terms of execution time by up to 61x. Furthermore, the novel DPR architectures have shown increased level of flexibility in core modification, continued operation of other tasks placed on the FPGA, and speed-up of up to 8x in partial reconfiguration time over full device reconfiguration.
- A DPR implementation of multi-classifier based on SVM/K-NN was achieved allowing the two cores to be swapped in and out of the FPGA, achieving 8x speed-up in partial reconfiguration time over full device reconfiguration.
- The performance of the K-means, K-NN, and SVM implementations on FPGAs with respect to the dimensionality of Microarray data were compared with equivalent GPPs showing superiority of FPGAs in terms of execution time when dimensions were high.
- An automated tool for performing fixed-point analysis has been developed in Matlab to facilitate the pre-processing of the Microarray data prior to hardware design.
- FPGAs have been evaluated as a high performance computing solution for BCB applications with respect to design entry, power, energy, area footprint, cost, and practicality, compared to GPPs and GPUs. The FPGA solution outperformed GPP and GPU in terms of execution time, power and cost efficiency. In terms of purchasing cost, the FPGA device used in this thesis was 1.69x more expensive than the GPP and 1.17x more expensive than the Nvidia GeForce 9600M GT used in the comparison.

1.2 Thesis Structure

The remainder of the thesis is organised as follows:

- Chapter 2 presents fundamentals of Microarray biotechnology, characteristics of Microarray data, and analysis methods used to extract biologically relevant information from them including un-supervised and supervised methods. Furthermore, the current and future applications of Microarrays in clinical settings are introduced with specific emphasis on applications related to cancer. Additionally, the challenges in current computational methods used in the analysis of Microarray data are highlighted which are thought to have led to the adoption of alternative computational platforms such as FPGAs. Finally, the chapter presents the main features and characteristics inherent to Microarray analysis methods lending them for hardware implementation.
- Chapter 3 presents a brief historical overview about the circumstances which have led to the invention of FPGAs and their evolution. Then an essential background on FPGAs is presented including details about their architecture, the available resources, and their main applications. Moreover, the methodology of mapping algorithms to FPGAs is presented covering the steps involved from design entry to on-chip verification. Lastly, DPR technology of modern FPGA is introduced covering methodology, design consideration, and advantages gained from applying DPR.
- Chapter 4 presents the hardware implementation of the K-means clustering algorithm consisting of five distinctive implementations. At first, the chapter introduces essential background on the K-means algorithm and prior work done in this area. Second, the chapter covers the main requirements for hardware design which include performing range analysis, precisions analysis, and fixed-point conversion; a tool will be developed to automate this analysis. Third, the actual hardware implementation of the single K-means core is presented followed by a multi-core implementation. Then, three novel DPR implementations of K-means clustering are presented whereby the first is based on a reconfigurable kernel within the K-means core, namely, the distance kernel. The second DPR implementation is based on a reconfigurable single K-means core. The last DPR implementation is based on reconfigurable eight-core architecture of the K-means core. Following the presentation of the architectures, the implementation results are presented and analysed for the five implementations.

Furthermore, the chapter presents a comparison between the proposed single/multi-core FPGA implementations of the K-means clustering and equivalent GPU implementations reported in the literature. Finally, the chapter lays out a comparative study about the power consumption of the K-means clustering of three platforms, namely GPP, GPU, and FPGA.

- Chapter 5 presents the hardware implementation of the K-NN classifier consisting of eight variable architectures. The chapter begins with essential background about the K-NN classification and prior work on FPGA implementation of the K-NN classification. First, the single-core architecture is presented consisting of two variable architectures, namely, A1 and A2. Third, a multi-core architecture based on fitting multiple K-NN cores onto the same FPGA targeting server solution is proposed. The remaining five architectures of the K-NN classifier are presented based on novel DPR implementations including the followings: architecture based on reconfigurable kernel within the single-core K-NN classifier, reconfigurable single-core architecture, a reconfigurable multi-core, a special case of the multi-core architecture known as the ensemble classifier, and a DPR implementation particularly suitable for facilitating the update of the memory contents based on reconfigurable memory content. Additionally, the advantages of the DPR implementations are outlined. Finally, implementation results of the aforementioned architectures are presented and analysed.
- Chapter 6 presents the hardware implementation of the SVM classification, consisting of five variable architectures. The chapter begins with essential background on SVM classification and prior work done on the implementation of SVM on FPGA. Following this, the chapter presents the five architectures consisting of a single-core SVM classifier based on two variable architectures called A1 and A2, quad-core architecture, novel DPR architecture based on single-core SVM, novel DPR architecture based on quad-core SVM architecture. The last DPR implementation is based on multi-classifier architecture, specifically-K-NN/SVM. Moreover, advantages of the DPR implementations are highlighted along with design considerations and implications. Finally, the implementation results of the aforementioned architectures are laid out and analysed.
- Chapter 7 presents an evaluation of the FPGA implementations presented in this thesis as high performance solutions for methods used in the analysis of Microarray

data. The evaluation includes applying optimised HDL coding for parallel FPGA designs, available hardware resources and computing bottlenecks. Furthermore, the chapter presents a comparative study between three computing platforms, namely GPPs, FPGAs and GPUs, which evaluates power and consumption, area footprint, cost, practicality in each of the three platforms. The ultimate aim of this chapter is to evaluate the performance of FPGA in BCB applications, and determine whether FPGAs are viable economic and efficient high performance solutions for BCB applications.

- Chapter 8 presents thesis summary and general conclusions. In addition, plans for future work are laid out including short and long term goals.

Chapter-2

Data Mining for Microarray Data

2 Data Mining for Microarray Data

2.1 Introduction

Microarray is a high throughput biotechnology which is used to measure the expression profiles of tens of thousands of genes simultaneously [7]-[10]. Consequently, it can be said that Microarrays perform multiple experiments in parallel producing vast amount of data in the form of gene expression profiles. This technology has revolutionised genomics, a field concerned about how genes work together to produce a phenotypic effect rather than trying to study genes in isolation [7]. As a result of its large throughput capacity and the valuable information it holds, Microarray is becoming an essential instrument for scientists studying gene regulation and molecular biology. However, the data resulting from Microarrays' are enormous and highly dimensional leading to difficultly in interpreting them and being non useful from the biological point of view. Consequently, many Microarray studies require complex data analysis to extract biologically meaningful information from such data.

The analysis of Microarray data is a significant task for unravelling the wealth of biological information embedded into the genome; this task demands high computational power, which has called for applying high performance computing (HPC) to the analysis of Microarray data. HPC has been gathering the interest of the bioinformatics community in recent years [7], [9]. The analysis of Microarray data leads to three main studies: first, discovering differentially expressed genes (up or down regulated), which is normally carried out using modern and classical statistics e.g., t-Test, non-parametric tests, and bootstrap analysis; second, studying the relationships and interactions between genes or samples carried out using un-supervised data mining methods, which includes wide range of clustering methods; third, classifying unknown samples based on prior knowledge of the classification of other gene expression samples using supervised data mining methods [7]-[11].

Today, Microarray analysis has helped scientists identify many genes associated with some types of cancers or other diseases which can be used in diagnosing diseases, discovering drugs, personalising treatment plans, and predicting treatment outcomes. More efforts are paid toward learning more about the regulation and interaction between genes to uncover new classes of tumours and to develop genomic based diagnostic models [8].

The remainder of this chapter will present an overview about Microarray technology, which will briefly cover the different types of Microarrays, the procedure of Microarray experiment, the instrument used, the required pre-processing, and the type of data produced. Second, a brief presentation about the methods used in the analysis of Microarray data for extracting relevant biological inferences. Then, a brief discussion about the rationale behind the implementations of data analysis methods in hardware will be given, which will include an overview about current and potential future applications of Microarrays applied to medicine and biology, the challenges of currently used methods, and the suitability of the data analysis methods considered in this thesis for hardware implementation. Finally, summary and conclusions of this chapter are presented.

2.2 Microarray Biotechnology

The main components of Microarray is a microscopic slide onto which tens of thousands of single DNA strands are mounted to them, those DNA molecules are referred to as DNA probes. The DNA consists mainly of genes which encode the instructions for protein synthesis required for all biological and functional processes in any living cell. The DNA probes attached to the Microarray glass slide can either be a complete DNA molecule, or short segment of DNA strands called oligonucleaotide, which are basically genes [6][13]. Each of the DNA probe types requires the implementation of different technology to spot those probes onto the Microarrays, different Microarray vendors use different methods each requiring different type of instrumentation and data processing. The attachment sites of the DNA probes are called Microarray spots or features, and a typical Microarray glass slide contains several thousands of those spots, where each spot contains millions of identical copies of the DNA molecule [13].

When performing a Microarray experiment, a sample of tissue or cells containing the DNA is labelled (with red or green dye) and injected onto the Microarray glass slide containing the spotted DNA probes as shown in Fig. 2.1. The slide is then placed inside a sealed chamber and left to incubate for 12-24 hours at temperatures varying between 45 and 65^{o} C. During the incubation period, the sample DNA will hybridise to the DNA of the probe via the Watson-Crick duplex formation. The role of microarray is to detect the level of hybridisation occurring which reflects the amount of gene expression occurring in each array. After the incubation period, the Microarray is washed to remove any excess hybridisation solution used in the experiment and eliminate cross-hybridisation effect

between spots. Then, the slide gets subjected to special scanner for image acquisition [7]. The procedure may vary depending on the Microarray platform used.

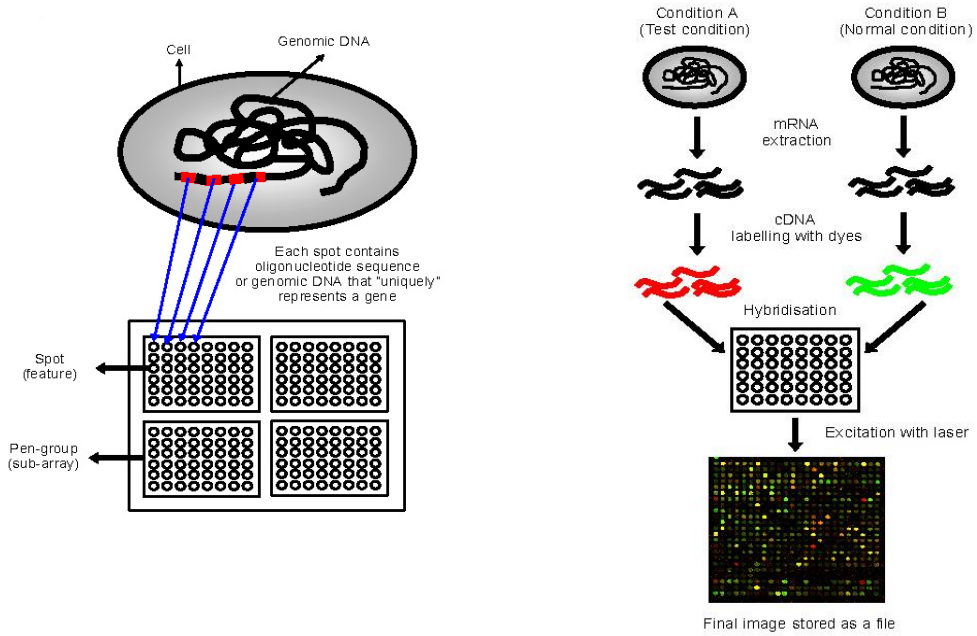


Figure 2.1: *An illustrative diagram of the Microarray technology (a) showing the arrangement of spots within a microarray slide containing the DNA probes, and (b) the procedure of injecting two labelled samples onto the Microarray spots, allowing it to hybridise for 12-24 hr, and finally exciting the spots with red and green lasers causing the spots to fluoresce (source ref. [13]).*

The Microarray image scanner consists mainly of two instruments: single or multi-colour laser emitter and a detection system which is usually a Photo-Multiplier Tube (PMT) or a photodiode; those are detectors transducing the light to electrical current that gets amplified reflecting the intensity of the light backscattered from the labelled DNA samples.

During image acquisition, every spot in the Microarray slide gets subjected to two or more laser beams (of different colours), which excite the dyes used to label the DNA samples, as a result of this excitation, the dyes fluoresce releasing red or green colours. The PMT detects the fluorescence emitted by the sample, and the intensity of the received light is recorded. Finally, a digital image is constructed consisting of a grid of pixels representing the light intensity emitted of every point in the array. In the case of using two lasers, two images will result each associated with one of the lasers; the two images are then combined together to form the famous red-green Microarray image shown in Fig 2.2. Every spot in the Microarray image is associated with one gene whereby the red colour reflects a

diseased sample and the green reflects a normal sample, the yellow and black reflect no change in gene expression and no hybridisation, respectively [6], [7].

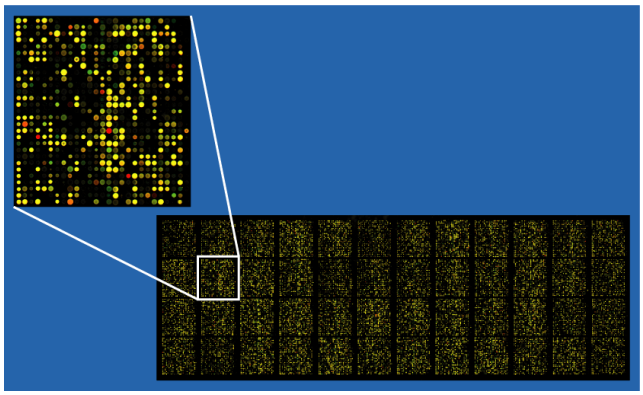


Figure 2.2: *An image of Microarray spotted with approximately 37,500 DNA oligonucleotides. The coloured spots are the intensities of the lights backscattered from the spots after hybridisation (source ref. [14]).*

Microarrays vary in the way they are made according to the type of DNA probes used and the method of attachment or spotting of those probes on the glass slide. There are two main methods used to spot the DNA probes, one is based on robotic spotting, while the other is in-situ synthesis. The former is based on using a robot to spot the DNA probes onto a glass slide, this type is additionally subdivided according to the type of DNA probe (complete DNA strand vs. DNA oligonucleotides vs. RNA strand); and the attachment method used for binding the DNA probes to the surface of the glass (covalent vs. non-covalent bonds). For more details on the matter, the reader is referred to [6]-[9] and [13].

As for in-situ synthesis method, the DNA probes are built up base-by-base on the glass slide, this type can be further subdivided according to the technology employed in attaching or printing the bases onto the Microarray slide, for instance the commercially well-known Microarray technology Affymetrix GeneChip^{TM} employs photolithography to direct light to the specific attachment site via a light mask every time a base is added [6]-[7]. Affymetrix GeneChip^{TM} has been well-known for its high detection sensitivity enabling the detection of low expression levels; however this technology is more expensive than spotted arrays [15].

Other in-situ technologies include the Micromirror and Inkjet Microarray, the former deploys computer controlled solid state Micromirror arrays to direct light to the specific attachment sites. Furthermore, the highest quality microarray images can be produced from

the inkjet array synthesis, which is based on spotting the bases to the attachment site via a computer controlled nozzle, similar to that used in inkjet printers allowing the bases to chemically bind to the glass slide without any involvement of light [6]-[7].

Although different technologies are employed in Microarrays, the resulting image in all is a map of light intensities, which is considered raw data as it cannot be used directly in inferring biological indications. Therefore, image pre-processing is required to convert those light intensities to numeric form prior to commencing data analysis. Most of Microarray manufacturers such as Affymetrix GeneChip^{TM} and Inkjet Microarrays supply special software packages customised for processing the raw images resulting from their scanners to obtain numerical representation of light intensities known as gene expression profiles, which can then be analysed using supervised or un-supervised methods. However for some spotted Microarray types, no such software packages are recommended by the supplier which requires user interventions to quantify the raw Microarray images through applying few image pre-processing algorithms and steps as discussed in the next subsection. On the other hand those steps are incorporated in most manufacturers' software packages [7]-[9].

2.3 Analysis of Microarray Data

Once the Microarray image is constructed, it is usually stored as a TIFF image reflecting the light intensities resulting from the hybridisation process. The following subsections will address the main steps required for extracting relevant information from those images as summarised in Fig. 2.3.

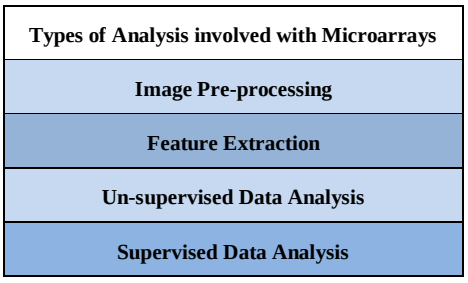


Figure 2.3: *Typical data analysis involved with Microarray*

2.3.1 Image processing and Feature Extraction

This step is typically done using software packages assisted with the human assessment; the involved steps are enlisted in Fig. 2.4 comprising of the followings: first, the identification of the locations of features appearing on the microarray image, and the pixels associated with the image along with the nearby pixels, the latter is used as background pixels needed for image subtraction as will be explained shortly; this step identifies any un-evenness in size or position of the features which constitute image artefacts. Second, the extraction of relevant features that are going to be quantified using image segmentation algorithms employed in many image pre-processing and feature extraction packages such as ScanAnalyze, GenePix, QuantArray, Dapple, Agilent, ImaGene and others [7]. Such packages implement the segmentation using different methods such as Fixed Circle, Variable Circle, Histogram, or others, for details on those methods the reader is advised to consult [6]-[7].

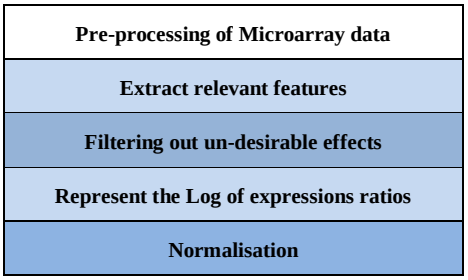


Figure 2.4: *Pre-processing's applied to Microarray data.*

As a consequence of image segmentation, the intensities of the Microarray features are quantified numerically based on the mean or median intensities of the extracted pixels, with the median usually favoured over the mean due to its robustness to outliers. Another factor usually taken into account when quantifying pixel intensities which is incorporated in most feature extraction software packages, is the standard deviation of the feature intensities and the background pixels. High pixel standard deviation relative to pixel mean results in bad features that are best excluded from the subsequent data analysis. Additionally, when the intensity of a pixel is less than that of the background, a negative numeric representation of the feature will result, which is considered noise. Low pixel intensities result into dark features, which are irrelevant to the analysis of Microarray data. Furthermore, as a consequence of technical mishaps such as having scratched slides, scanner misalignment,

aging of instrument, poor hybridisation and many others, missing data could result which must be dealt with prior to analysing the data [6]-[9].

Following the aforementioned pre-processing, a third process commences for the filtration of the aforementioned image artefacts composed of high standard deviation pixels, negative, dark, and missing features. Alternatively, the original image must be re-segmented and re-processed in an attempt to correct for any mistakes that could be attributed to the used methodology or to human errors that might have led to these artefacts. One of the filtering measures taken at this point is the subtraction of the background intensities from the feature's intensities due to the fact that background signals include signals coming from the slide surface itself and from non-specific hybridisation contributing to the overall intensities of pixels, such contribution is un-wanted since it degrades the quality of the numerical representation of the Microarray image. Positive results from the subtraction are maintained while negative results are removed as they indicate presence of ghost pixels which have signals lower in intensities than those of the background [13].

The next pre-processing step involves taking the logarithms of the relative expression ratios to correct for inconsistent intervals between ratios of up versus down regulated genes. This is followed by normalisation of the logarithmic expression ratios, which is necessary for correcting the effect of non desirable experimental variability causing variable pixel intensities e.g., having two samples of different DNA quantities or different amount of fluorescence dyes [17]. The role of normalisation is basically to adjust for such experimental variability and enhance the distribution of the gene expressions from random toward normal Gaussian distribution [9], [13]. Fig. 2.5 illustrates the effect of normalisation on the Microarray image. Sometimes normalisation is carried out before taking the logarithm of the gene expression ratios.

The aforementioned processes are the most relevant steps related to the pre-processing of the Microarray image, which result into data being presented as a numeric matrix as shown in Fig. 2.6. The matrix includes the measurements of genes across many Microarrays; those Microarrays reflect different samples (e.g., normal v.s. diseased vs. disease subtype), experimental conditions, patients, or perhaps samples taken from tissues exposed to different medical treatments [15]. An example of a Microarray matrix is one having 20,000 rows and 400 columns reflecting the measurement of the expression profiles of those 20,000 genes across 400 microarrays [9].

As a consequence of the heavily involved pre-processing mentioned above, it can be stated that the quality of the final numeric representation of Microarray data is subjective to the methods and algorithms used in the pre-processing of the light intensity image [7], [9], [13], [17]. Using sophisticated software packages will simplify the process.

Having converted the Microarray data to the matrix form shown in Fig. 2.6, the next step is to analyse them by applying statistical analysis, or data mining using un-supervised or supervised machine learning methods to extract relevant biological meanings. The concern of this thesis is to investigate the feasibility of applying FPGAs to the analysis of Microarray data based on un-supervised or supervised methods; therefore the next two subsections will be dedicated mainly to introducing those two methods, the reader interested in statistical methods is advised to consult [7] and other relevant references in the subject.

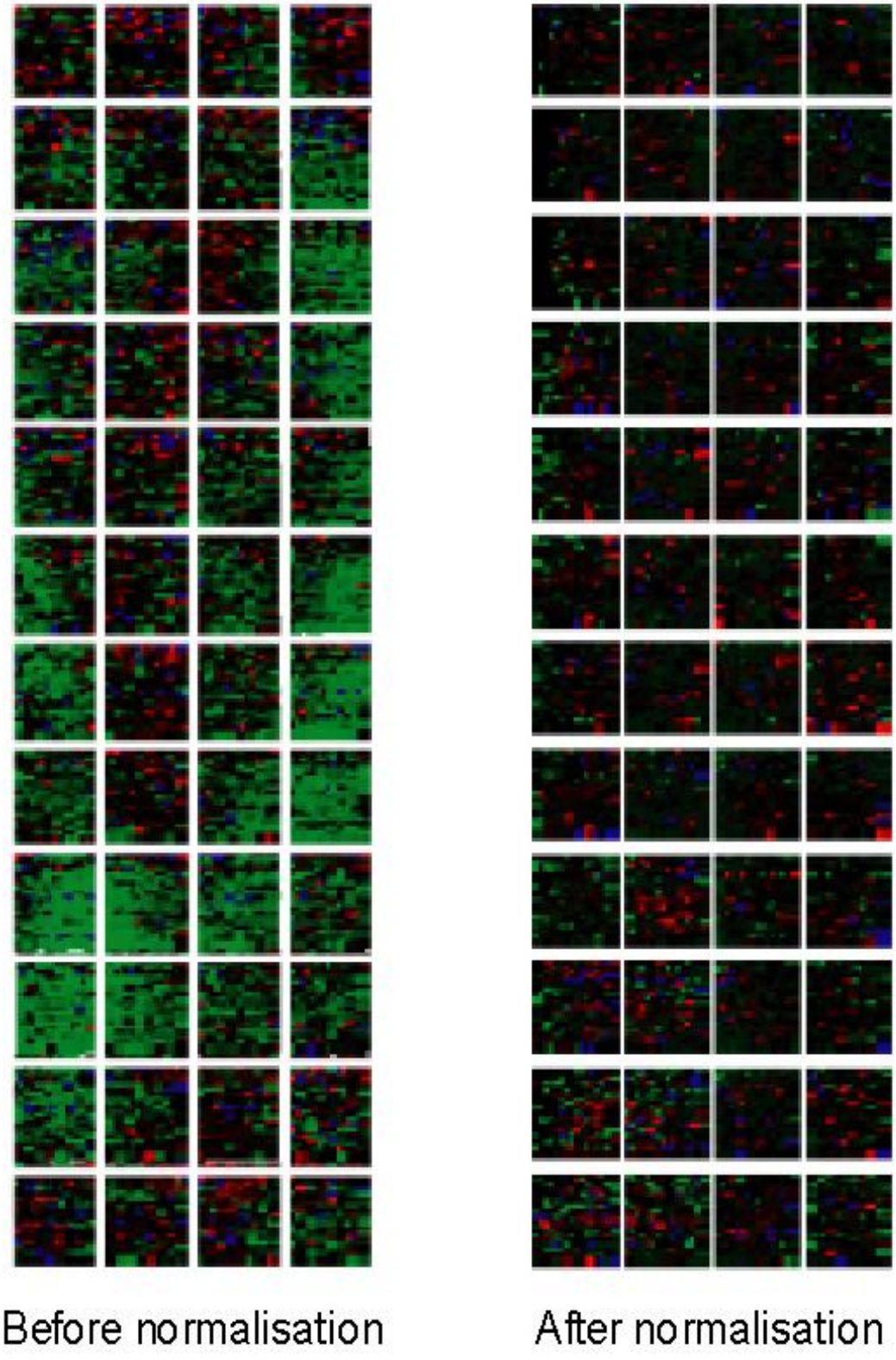


Figure 2.5: *Illustration of the effect of normalisation on the gene expression image: (a) before normalisation showing spots having widely variable intensities, (b) same image after normalisation showing only distinctively variable spots (source ref. [13]).*

Gene ID	Sample 1	Sample 3	Sample 4	……..	……..	…….	Sample M
Gene 1	x11	x12	x13	……	…….	……	x1M
Gene 2	x21	x22	x23	……	…….	……	x2M
Gene 3	x31	x32	x33	……	…….	……	x3M
Gene 4	x41	x42	x43	…….	……..	……	x4M
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
GeneN	xN1	xN2	xN3	xN4	xN5	xN6	xMN

Figure 2.6: *Gene expression matrix where rows represent genes and columns represent samples which could be from different tissues, or same sample studied at different experimental conditions.*

2.3.2 Un-Supervised Methods

The field of pattern recognition has been dealing with the analysis of data without having priori knowledge about them (e.g., un-classified tissue samples); the methods implemented to analyse data of such nature are referred to as un-supervised methods. Among the widely used un-supervised methods is clustering, which is subdivided into two main categories, one is hierarchical while the other is non-hierarchical or partition clustering (also known as divisive). The aim of both types is to try to identify patterns among the given set of genes and group them such that genes with similar expressions or patterns are placed into the same groups, similarity is usually measured using distance metric such as Euclidean, Pearson's correlation, or many others [6], [9]-[11] and [15].

The hierarchical clustering is further subdivided into two types: agglomerative and divisive, where the former attempts to form a branching tree called dendogram (or clustergram) starting from the assumption that every gene is a cluster, and then similar genes are joined together based on pairwise distance computation to form larger clusters. The process iterates till smallest number of clusters is reached. On the other hand, the divisive clustering is based on the assumption that all genes are in one cluster which get broken down by means of principal component analysis (PCA) to smaller clusters containing related genes. Agglomerative clustering is more common than divisive, and can be implemented using different algorithms such as single-linkage, complete-linkage, average-linkage, and others [13] and [11].

Hierarchical clustering has been widely used with Microarrays since the early 2000s as a result of the introduction of simplified automated tools such as the one developed by Eisen *et al.*, [15]-[16] to present the result of clustering in the form of heat map, and more recently as a result of its incorporation in most Microarray software packages. Eisen *et al*. developed software capable of applying hierarchical clustering to Microarray data and presenting the clustering result graphically with respect to the gene expression image as shown in Fig. 2.7, where adjacent genes within the tree are co-expressed and thought to have similar functions, for more details about this method, the reader is advised to consult [16]. In contrast to its simplicity related to visual presentation of clustering results, hierarchical clustering does not perform well when data are so large leading to the requirement to use other clustering methods [10] and [17].

Non-hierarchical clustering are divisive or partitioning methods requiring the pre-determination of the number of clusters to assign genes having similar gene expressions to the same clusters. The main two non-hierarchical clustering methods used in the analysis of Microarray data are the K-means (borrowed from pattern recognition) and Self Organizing Maps (SOM) (borrowed from Neural networks), both are based on iteratively assigning genes to one of the predefined clusters according to distance computation between the genes to be clustered and the cluster's centroids. In SOM, clusters are related using spatial topologies, whereby the orientation of clusters within a grid is pre-determined along with the number of clusters [17]. In addition, K-means is considered the second most common clustering method applied to Microarray data, and one proven to be robust when used with large data; partition clustering algorithms are particularly useful in cases when hierarchical clustering is not feasible [10] and [15].

Furthermore, K-means has been favoured over hierarchical clustering for large data due to having smaller time and space complexity. K-means clustering is used to either cluster genes or samples depending on the requirements of the study, Fig. 2.8 presents a graphical representation of the clustering results of filtered genes from yeast data using Matlab Statistical Toolbox illustrating the power of K-means in partitioning genes of similar profiles in same clusters; more details will be presented about the K-means algorithm in subsequent chapter as it presents an essential chapter in this thesis.

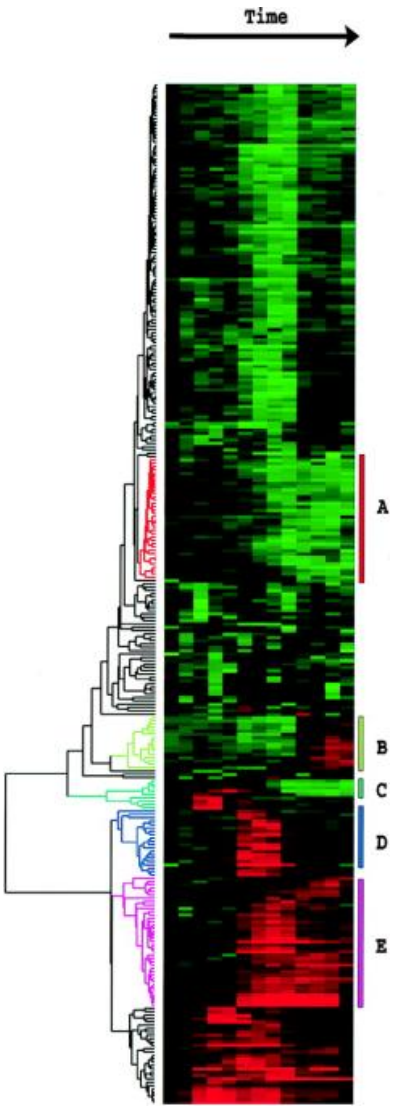


Figure 2.7: *Hierarchical clustering of data from human fibroblast serum, where the colour map (heat map) consists of the gene expression spots, whereby the green spots represent down-regulated genes while the red represent up-regulated. The tree (dendogram) in the left represents the corresponding hierarchical clustering (source ref. [16]).*

In all clustering methods, it has been widely accepted that genes in same clusters have similar expression profiles, and believed to share some degree of similarity in the transcription factors of DNA leading to the assumption that they have similar binding sites, same protein, or gene functions [13]. However, clustering methods have the disadvantage of having to determine the number of clusters beforehand, yielding variable shapes, and are being difficult to assess or validate. Additionally, the quality of the clustering is subjective to the selected method, the normalisation, and the similarity measure leading to the fact that clustering methods may produce better clusters than others. Consequently, no single

clustering solutions is ideal leading to the requirement to try several methods or combine results from different methods [10]-[11], and [17]. In many studies, clustering is used to identify patterns embedded in large data and separate them into clusters having particular class labels, which can then be used as priori data for additional data analysis using supervised methods such as constructing predictive models [10]-[11].

The validity of the K-means clustering of Microarray data can be carried out using one of the following methods. The first is by visually checking that the genes in one cluster have similar expression profiles. The second method is by performing biological validity, by checking if some of the genes in one cluster have similar biological functions or resulted from same process. Third, by re-clustering the same data using the same number of clusters K, if the new clustering is same as the original clustering, then the original clustering is performing well. The last method is based on statistical analysis such as bootstrapping [7].

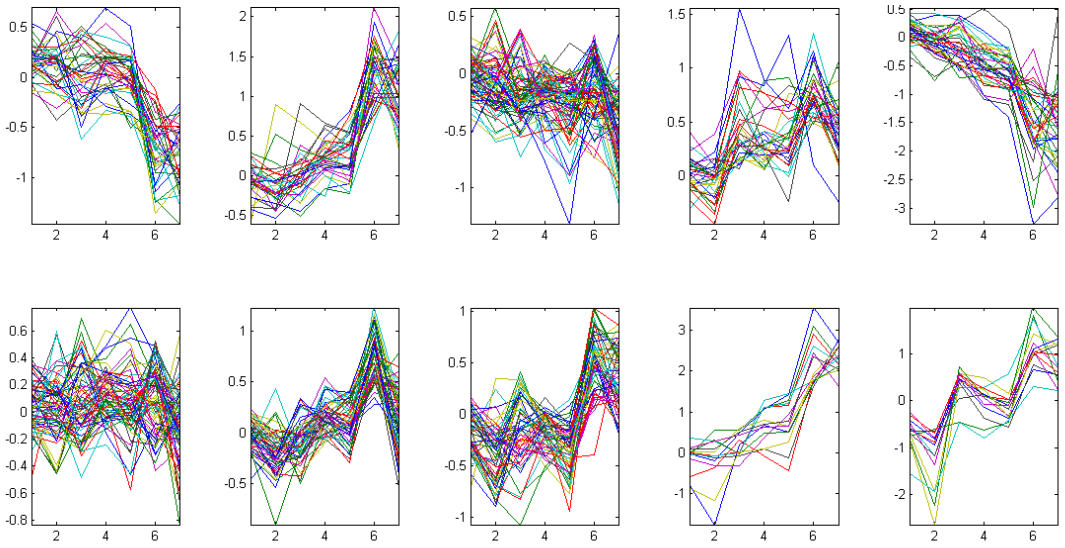


Figure 2.8: *A graphical representation of the result of applying K-means clustering on filtered yeast data (having 415 genes × 7 dimensions (samples)) using ten clusters, where the x-axis is the sample number and the y-axis is the gene expression profile. Each line illustrates the expression profile of one gene across all the seven samples. The genes within one cluster appear to exhibit similarity in the expressions with some clusters appearing to have some outliers. Clustering performed using Matlab Statistical Toolbox.*

2.3.3 Supervised Methods

Supervised methods belong to pattern recognition, a field studying how machines can observe an environment, leaning how to distinguish the patterns associated with it, and

making decisions about categorising the patterns [10]. Supervised methods make use of priori knowledge about the given data such as the class label of some of the samples e.g., cancer vs. non-cancer cells, normal vs. diseases. This information is used to construct a prediction model whose aim is assigning a class label to an unknown sample [11]. The data used for constructing the model are called the training data, which constitute the expression profile of genes across samples, with an additional attribute forming the class label. There exist many supervised methods performing the prediction task which are collectively called classifiers such as nearest centroids, K-NN, Parzen, Neural networks, and SVM, with each having advantages and disadvantages [7]-[8].

Some of the issues surrounding this family of machine-learning methods are the linearity of the data, number of classes involved, and training time, some classifiers applicable to cases when the Microarray data are linearly separable, while others can be applied to non-linearly separable cases. K-NN classifier is among the simplest and commonly used methods due to its applicability to classify linearly and non-linearly separable expressions, and its ability to handle data with more than two class labels. In addition, K-NN does not require training as opposed to SVM [7].

SVM is also a popular classifier in Microarray data, which can work with linearly and non-lineally separable data, however it can readily deal with binary class labels only, special arrangements can be made to accommodate cases of non-binary data. Nearest centroids is one of the simplest classifiers which has the advantage of being suitable for classifying linearly separable data, it has been established that non-linearly separable data lead to many misclassifications. Furthermore, Neural networks has been applied widely to classification problems of Microarray as a result of being able to train non-linearly separable expressions, and of being able to deal with multi-class labels, however Neural networks have the disadvantage of being slow to train [6]-[8] and [11].

In the analysis of Microarray data obtained from experiments based on using samples taken from patients with known diseases or pathologies, bioinfomaticians have successfully been able to identify genes that are associated with specific diseases such as cancer. Accordingly, many databases have been made available to the scientific community through the world wide web, which include various lists of genes confirmed to be associated with particular types of cancers e.g., Leukemia, breast cancer, or others. Those databases, or new data obtained from new patients, can be harnessed to construct a predictive model using one of the aforementioned classifiers to classify unknown samples [8].

Different classifiers operate in different manners and have the potential to yield different classification results. This calls for experimenting with different classifiers to select one which works best for the application in hand. As a consequence, combining the classification of multiple classifiers by means of voting or averaging leads to improved classification accuracy, this method is one the ensemble classifiers [10].

2.4 Rational for selecting the proposed mining methods

The high throughput nature of Microarray has resulted into considerably high volume of genomic datasets, which are difficult to analyse and interpret or even make sense of. This has led to the application of many supervised and non-supervised data mining methods to transform Microarray data into useful forms that can reveal biological inferences. In this work, three methods have been selected for hardware implementation. The following subsections will shed some light on the reasons behind the selections made which will include review of some clinical studies revealing the potential impacts of such methods with respect to cancer; followed by statements about the limitations imposed by current methods. Lastly, a discussion about the features available in such data mining methods which make them candidate for hardware implementation will be given.

2.4.1 Applications of Microarray

One of the main research areas in Microarray is applying un-supervised and supervised methods to cancer data; cancer has been the second leading cause of death in many countries around the world after cardiovascular disease, and one that is gaining tremendous attention by the scientific community and governments to try to decrease its deaths and enhance survival rates. Microarray has been one of the fundamental instruments in modern cancer research due to the fact that cancer is a multi-disease which genetic and genomic underlies are believed to determine the prognosis and treatment response of the disease.

Current applications of Microarrays related to cancer are the identification of new tumor classes by using clustering methods. This mainly aims toward discovering new cancer types or subtypes. Additionally, supervised methods are used to classify unknown samples into known cancer types; this implementation is particularly promising for the development of molecular diagnostic tools that can be used to diagnose patients with cancer based on their gene expression profiles.

In addition, cancer research related to Microarrays has already established that some genes determine disease outcome or its prognosis, e.g., whether the type of cancer is highly invasive, likely to spread to specific remote site, or likely to recur. Most importantly, Microarrays help in predicting the response of specific treatment. Current cancer treatments have severe side effects and are toxic to non-cancerous tissues; examples of methods used to treat cancer are: surgery, chemotherapy, immunotherapy and radiotherapy. Consequently, knowing how a tumor will respond to any of those treatments help clinicians in planning the most effective therapy reducing the harmful effects of unnecessary treatments the patient is less likely to respond to. This vision comes from the fact that many patients having the same cancer subtype (based on same tumour grade determined using histopathology) were found to develop different response to the same administered treatment, and experience different prognosis; this finding has led clinicians to believe that there are other factors (mostly genetic) playing crucial role in the prognosis of cancer and the response to specific treatment [18]-[20].

Research applying supervised and un-supervised data mining methods to Microarray data of cancer tissue has led to the development of diagnostic tools that are currently used to predict the outcome of the disease. MammaPrint, a Microarray based test kit developed by Microarray giant Agendia has been used successfully in predicting the risk of recurrence in breast cancer patients [21]. MammaPrint has been approved by the Food and Drug Administration (FDA) in the US leading to commercialising the kit for use as part of a complete breast cancer management suite called SYMPHONY^{TM} developed by Agendia. The company claims that MammaPrint is capable of classifying breast cancer patients as low or high risk of having disease recurrence with 98.9% accuracy. The test kit is based on using 70 genes established as being breast cancer signatures as DNA probes in a Microarray slide. Those 70 genes were first identified by van't Veer *et al.* in 2002 who used Microarrays to analyse the expressions of 25,000 genes in 78 patients using supervised methods and concluded that among the 25,000 genes 70 genes can be used in predicting the outcome of the disease [19]-[20] and [22]. This commercial test is an outcome of several Microarray studies to validate the results obtained by van't Veer *et al.* Ongoing studies are attempting to find gene markers related to different types of cancer that can help in understanding the mechanism of the disease and lead to the development of more test kits. Additional test kits are made commercially available but are requiring more trials for validation before being clinically acceptable such as ColoPrint, a test kit aiming for predicting the risk of relapse in stage II colon cancer and predicting the outcome of systematic chemotherapy in those

patients [23]. Results of such diagnostic tools are helping clinicians to some extreme to personalise treatment plans and introduce appropriate recurrence screening to people of high risk of breast cancer recurrence.

Microarray is part of a collection of other high throughput genetic and genomic biotechnologies capable of analysing many samples, which are expected to revolutionise healthcare in the upcoming years. Personalised Medicine (PM), a field of molecular medicine gaining popularity in medicine nowadays is based on using genetic and genomic information of patient along with protein and environment knowledge to diagnose and treat diseases such as cancer. Cancer has thought of as a heterogeneous disease resulting from a combination of genomic and environmental factors, PM uses Microarray results to uncover the genomic factor to identify people at risk of some types of cancer, consequently providing a mean for cancer screening. For instance two genes have already been identified to have direct association with colon cancer whereby people who have mutations in those genes are at high risk of developing the disease at a later stage in life, such high risk individuals could be subjected to some preventive measures if known environmental factors or nutritional ones are found to give rise for the disease or they can be screened at regular intervals to help in detecting the occurrence of the disease at its earliest stage [21].

Moreover, the work presented in [24] emphasises the role of Microarray analysis in the prediction of cancer subtypes. The work in [24] has resulted in an automated class prediction classifier which was able to assign a leukaemia sample to a known leukaemia subtype using supervised K-NN to classify the expressions of leukaemia samples into two distinctive classes: acute myeloid (AML) or acute lymphocytic leukaemia (ALL). The authors provided one of the well-known benchmark leukaemia cancer data used today in validating many classifiers. Other studies resulted in similar prediction models for other types of cancer e.g., prostate, lung, breast, pancreas, bladder, liver, and kidney using different supervised methods [18]. More work is required to confirm the diagnostic power of such data to be able to use them clinically.

In summary, supervised and un-supervised methods applied to Microarray data have been key elements in applying PM for tailoring individualised therapy. Such potential applications not only enhance recovery from the disease, but also reduce the cost associated with treatments that are expected to be non-responsive.

The aforementioned overview covered small applications of Microarrays supported with a number of studies to highlight the impacts of Microarray data analysis methods on

transforming the large numeric matrices to valuable information that can be employed in clinical medicine. Additional works are required to analyse more cancer types and identify sets of predictive genes that could lead to more test kits similar to MammaPrint or Oncotype DX (both used in breast cancer). Moreover, there are additional clinical potential embedded in Microarray that are not approached yet. For instance, combining data from multiple Microarray experiments enables biologists to ask more questions such as the relationship between genes obtained from different cancer types/subtypes, the likelihood of one type of cancer to spread to specific remote sites, or to study the genomics of other diseases e.g., Alzheimer, Parkinson, Down Syndrome, and many others to determine the underlying genetic causes that may lead to more effective treatments. Furthermore, more efforts are required to transform individual diagnostic kits targeting specific cancers to a comprehensive single diagnostic tool which integrates the signatures of all identified cancers with their subtypes to be able to better exploit the Microarray technology in PM. This comprehensive integration of genome data increases the requirement of powerful computational methods, which current GPPs are unable to cater for efficiently leading to the requirement for adopting new computing platforms.

2.4.2 Challenges in Current Methods

The analysis of Microarray data using K-means clustering, K-NN, and SVM methods are computationally intensive as a result of large data size, high dimensionality, and involved computations. For instance, K-means clustering is iterative algorithm leading to long computational time in GPPs. The time complexity of the algorithm imposes limitations on the type of analysis that could be carried out, thus restricting the size of studies that could be conducted to small sizes. Current GPPs have reached the maximum capacity in terms of clock speed mainly due to associated high power consumption, consequently a cap has been placed on the complexity of questions which biologist attempt to ask leading to partial exploitation of the wealth of information embedded in Microarrays. As such, biologists are calling for advances in computing platform that can revolutionise Microarray analysis, be adaptive to the different analysis methods, and able to handle the volume of data efficiently. Such demands are accompanied with the requirements for affordable purchasing cost and low power consumption leading to economical solutions to the problem. Additionally, simplicity of usage is essential for its acceptance by the biologist, bioinfomaticians, and clinicians.

The analysis of large and complex Microarray data has called for exploiting other methods such as GPP clusters, supercomputers, ASICs, GPUs, Multi-core GPPs, and FPGAs. Using GPP clusters and supercomputers imposed issues of high cost and power consumption, which has led to the exploitation of alternative platforms, such as ASICs. ASICs offer high performance computing at low cost, but are inflexible and associated with large development and manufacturing times; they do not allow easy customisation or re-programmability thus are inflexible. Microarray data are variable in size and complexity demanding highly scalable solutions.

FPGAs, the rapidly growing reconfigurable computing hardware platforms, have established themselves as promising accelerators of many BCB applications due to their high performance. Additionally, it has been thought that the high power consumption of supercomputers and GPP clusters has further catalysed the movement of BCB applications toward FPGAs from one hand, and the re-configurability of FPGAs from another hand. The latter makes FPGAs more fortunate in terms of flexibility over ASICs, and as such more favoured.

More and above, GPUs, which were mainly used for graphics and games, have gained popularity in recent years as accelerator to BCB applications. Various works have been reported in the literature about the acceleration of BCB applications using GPUs [25]. Although, GPUs benefit from lower cost compared to FPGAs and from shorter development time, they are inferior to FPGAs in terms of power consumption.

In sum, hardware implementations of data mining methods used in the analysis of Microarray data such as K-means clustering, K-NN, and SVM classifiers are among the current frontiers of research in BCB. FPGA has been selected here as a consequent of its high performance, low power, and successful FPGA implementations of other BCB applications presented in [25]-[27], and in many other works reported in the literature.

2.4.3 Suitability for Hardware Acceleration

The main requirement for the hardware implementation of any of the data analysis methods is to have kernels within the algorithm that can benefit from the parallelism offered by hardware. K-means clustering involves distance computations between every gene and all clusters; consequently it contains kernels that can exploit hardware parallelism. K-NN and SVM involve distance computations between a query sample and all the dimensions of all

the genes, and contain other kernels that can be parallelised and pipelined. As a result, the three methods are expected to benefit from hardware implementation.

Additionally, mapping algorithms to hardware require that the computations involved in the algorithm can be realised by the available configurable logics, this requires breaking down complex functions or arithmetic operations into smaller segments that can be easily realised by the hardware logic harnessing the fine grain granularity of FPGAs. In the aforementioned data mining methods, simple arithmetic operations are involved requiring subtractions, multiplications, finding absolute values, and division leading to the possibility of mapping them to hardware.

Various other groups who have implemented the aforementioned methods in FPGAs targeting other applications reported promising results which encouraged the selection made here for applying FPGAs to the analysis of Microarray data. In addition to implementing K-means, K-NN, and SVM algorithms on FPGAs to specifically target Microarray data, the implementations presented in this thesis propose novel approach based on DPR for adding high level of flexibility when dealing with server solutions. The adopted DPR approach has not been reported in literature. In sum, the selected data mining methods have intrinsic parallelism making them eligible for acceleration in FPGAs and are due to benefit greatly from current state of the art features of modern FPGAs.

2.5 Summary and Conclusions

Microarray is a high throughput biotechnology requiring special strategies for preparation and acquisition. Pre-processing of the Microarray images is fundamental process in the preparation of a Microarray image and its conversion to numeric matrix of gene expression profiles. The size of the matrix is usually large requiring high computational power to apply statistical, un-supervised, or supervised methods for analysing such data. Un-supervised methods assume no information is available about the classification or type of the samples as in the case of clustering methods (hierarchical or K-means), or Neural networks (SOM). On the other hand, supervised methods are associated with known information about the presented samples such as their class labels e.g., normal vs. diseased, or disease subtypes.

While clustering genes are used for the identification of new subtypes of the disease or group related genes, supervised methods are used in the classification of samples such as assigning classes to unknown samples based on predictive model constructed using data of known classes. As a result, Microarray biotechnology has established itself as a fundamental tool in studying human genome to reveal the wealth of information embedded within it, which has led to a wide range of applications such as gene discovery, disease prognosis, and treatment response.

Although Microarray has aided the development and commercialising of few test kits used in the diagnosis and prediction of disease outcome, more efforts are needed to learn more about the mechanism and pathways of genes with respect to many diseases such as cancer to transform Microarray data to clinically relevant information that can be used diagnostically. Understanding relationships between the genome and clinical outcome will have strong impact on clinical medicine, patient survival rates, and cost of healthcare. To un-lock the high potential of Microarrays, high performance computational platforms need to be integrated to the field of Microarray analysis to overcome limitations of current methods. Parallel computing is one of the promising methods that holds the key for un-locking the potential of Microarrays. FPGA is an example of high performance parallel computing platform that can be applied to the analysis of Microarray data.

Chapter-3

An Introduction to Computing with FPGAs

3 An Introduction to computing with Field Programmable Gate Arrays (FPGAs)

3.1 Introduction

Computing is part of the everyday life of most people around the world from using personal computers to handheld devices such as smart phones, game consoles, and tablet computers. Additionally, computers are involved in all industries and have become part of many household appliances, work and public domains. The most popular forms of computing platforms are GPPs, which are based on fixed hardware circuits executing instructions sequentially. As a result of Moore's law which states that the number of transistors within an integrated circuit (IC) doubles every 18-24 months, semiconductor chips have become very complex and capable of implementing complete System-on-Chips (SoC). Accordingly, GPPs have been growing in speed and computing power at rapid pace, and have been so popular as a result of their re-programmable nature, affordable cost, and simple usability.

Fixed application-specific ICs (ASICs) are highly dense and heterogeneous integrated circuits customised to perform particular tasks, which have emerged in the 1960s and kept evolving rapidly since then. Fixed ASICs have become essential parts in the circuitry of mobile phones, household appliances, consumable electronics, and many more. The high performance of fixed ASICs, their low cost, and low power consumption have led to their widespread use. However, once fabricated, the functionality of fixed ASICs remains permanent as they can not be re-configured. To change the functionality, fixed ASICs must be re-fabricated which incurs high non-recurring engineering cost (NRE) even if the required change is so small. Fixed ASICs are characterised of having low cost production because they are manufactured in high volumes to include the amount of logic necessarily to perform the intended logic circuits only, this involves customising transistors and their physical connections within the IC. The end users of fixed ASICs are manufacturers of specific products or manufacturers of particular industry such as cars, mobile phones, computers or many others. Such markets require high performance and large volume of application specific digital circuits catered by fixed ASICs [28]-[29].

Unfortunately, like any other technology, fixed ASICs do have some disadvantages related to their long time to market, and non-re-programmability. The long time to market is

a consequence of the complex development process associated with specifying the intended functionality in HDL using sophisticated CAD tools followed by physically mapping and placing the design into the IC requiring custom fabrication. As a consequence of those two disadvantages and the constant technological advancements in semiconductors, a new form of programmable IC has emerged in the 1970s called Programmable Array Logic (PAL), which can be re-programmed after manufacturing to implement simple combinational or sequential circuits. PALs contain small number of logic gates and are not suitable for realising complex digital circuits. A further development in transistor density and IC fabrication has led to packing multiple PALs into single IC forming complex programmable logic devices (CPLDs), consequently allowing for mapping complex digital circuits into small ICs. However, there is a limit to the number of PALs that can be incorporated into CPLDs, and packing large digital circuits into them is usually associated with inefficient use of the logical resources within the CPLDs. In the 1980s, the advent of FPGAs has overcome some of the limitations of CPLDs, and shortly after FPGAs started to overtake CPLDs and fixed ASICs [29].

FPGAs are reconfigurable computing platforms which have been evolving at a rapid pace over the last three decades growing as ICs of few hundreds of logic gates to several millions. FPGAs are based on ICs containing enormous amount of small logic cells that can be configured or programmed to do many logical operations specified in HDL code. The process of converting the HDL code to logic on the FPGA is called mapping an algorithm (or task) to a FPGA. FPGAs have also evolved from being homogenous to being heterogeneous architecture including dedicated functional blocks and soft/hard intellectual property (IP) blocks.

One of the main advantages of FPGAs is that they can be configured to execute multiple instructions in parallel and can pipeline tasks leading to high performance computing. The level of parallelism inherent in FPGAs is responsible for their popularity in applications requiring high performance, given that such applications lend themselves to hardware implementations. The availability of abundant local memories has assisted the leverage of parallel and pipeline computing in FPGAs. Additionally, the unlimited re-programmability or re-configurability of FPGAs at low or almost no NRE, the high density of logic gates embedded into them, relatively low power consumption, and immediate time to market have all led to the adoptation of FPGA in wide range of applications and by many industries. Today, FPGAs have been used as accelerators to many applications serving as coprocessors

to GPPs, whereby specific segments of an algorithm running on a GPP are ported to FPGA where they can be executed faster [30].

The remainder of this chapter is organised as follows; first essential background on FPGAs will be presented covering the architecture, layout, and applications. Second, the procedure of mapping an algorithm to FPGAs will be presented covering the design methodology, required tools, and actual implementation on the device. Third, methodology, features, and advantages of DPR technology will be briefly discussed as it constitutes an essential instrument in the contributions presented in this thesis. Then, brief introduction to the ML 403 board will be given since it is the testing platform used in the implementations presented in subsequent chapters. Finally, chapter summary and conclusions will be presented.

3.2 FPGA Essentials

FPGAs have been attracting the HPC industry due to their capabilities in exploiting coarse-grained functional parallelism as well as fine-grained instruction level parallelism at low power [30]. The IC technology implemented by FPGA vendors vary from one vendor to another and from one device family to another within the same vendor as a consequence to continuous and rapid advancements in semiconductors embraced by FPGA developers, and as a result of different consumer requirements. For instance, Xilinx Virtex-II was fabricated based on 130 nm technologies, whereas the subsequent devices Virtex-4, Virtex-5, Virtex-6, and Virtex-7 are based on 90nm, 65 nm, 28 nm, and 28 nm, respectively leading to the integration of more logic cells (LCs) and achieving faster clock speed [31]-[32].

As a result of the current existence of many FPGA vendors applying different technologies and architectures, it is difficult to address a generic view of the FPGA architecture. Consequently, the subsequent description of the FPGA architecture is based on Xilinx Virtex-4 architecture which embraces Xilinx' novel Advanced Silicon Modular Block (ASMBL^{TM}), a unique columnar architecture which is based on arranging specific resources such as CLBs, I/O Banks, DSP blocks, and Block RAMs into separate columns. According to Xilinx, ASMBL^{TM} has led to significant enhancements in power, distribution of resource, and routing, which have been the reasons for using ASMBL^{TM} in subsequent FPGAs developed by Xilinx [32]-[33]. Additionally, Xilinx divides the FPGA into two equal halves separated by a middle column; both sides are further divided into equal and symmetrical clock regions.

3.2.1 FPGA Architecture

The FPGA IC consists mainly of two-dimensional array of configurable logic blocks (CLBs) arranged as tiles of repeated pattern as shown in Fig. 3.1 interconnected by programmable interconnects forming a complex routing network. Additionally, FPGAs consist of programmable I/O blocks (IOBs) which are arranged around and within the FPGA depending on the architecture embraced by the FPGA vendor and the specific device family. Xilinx is one of the leading FPGA vendors, which was established by the inventor of FPGAs Ross Freeman in the mid 1980s. Altera, Actel, Lattice Semiconductor, and Atmel are among current vendors of FPGAs [30]. Xilinx and Altera both share a significant share of the FPGA worldwide market. Although most FPGAs contain similar resources, Xilinx' FPGAs and design tools are specifically considered in this chapter because Xilinx Virtex-4 devices are used in the hardware implementation presented in following three chapters.

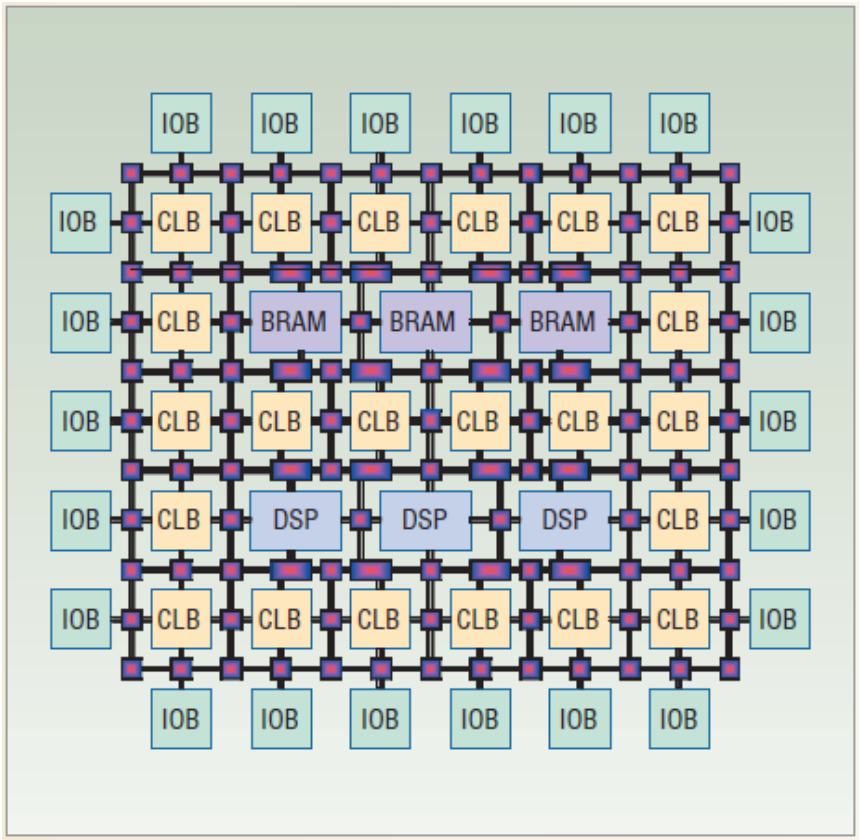


Figure 3.1: *A generic architecture of Xilinx FPGA illustrating the structure of the chip and the most common resources, IOBs are arranged as a ring around the FPGA, arrangement of resources vary according to the vendor and device family (source ref. [30]).*

The main component in the FPGA tile is the CLB composed of four slices in Xilinx Virtex-4 arranged in pairs, whereby each slice consists of two logic cells (LCs) constituting the sequential and combinational circuits. There are two types of slices: SliceM and SliceL, which are located to the left and right sides of the CLB column, respectively as shown in Fig. 3.2, whereby each slice is labeled according to its location in the matrix as $X_{\text{column no.}}$ $Y_{\text{row no.}}$. The number of columns counts up from left to right while the number of rows counts up from bottom to top as illustrated in Fig. 3.2. Each column is connected to a switch matrix which acts as a bridge to connect the CLB to the routing network of the FPGA as shown in Fig 3.1. While SliceM can be used to implement logic, shift register, or Distributed RAM, SliceL can only be used to implement logic [31]-[32].

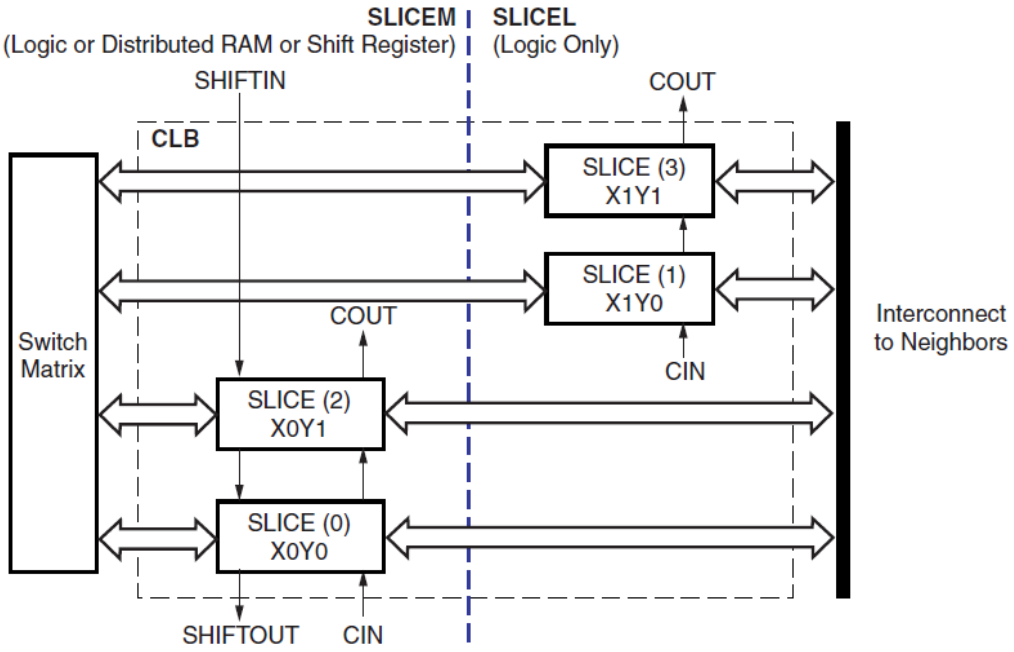


Figure 3.2: *The architecture of a Xilinx-4 CLB, which constitutes of four slices of two types arranged in pairs (source ref. [32]).*

The logic resources of one complete CLB vary from one device family to another, Table 3.1 summarises the resources in the most recent Xilinx device families. The main building block of a Virtex-4 slice is two LCs, whereby each LC constitutes of 4-input lookup table (LUT), a multiplexer, and a flip-flop, used to realise the combinational and sequential logical operation, respectively. Fig. 3.3 illustrates the components of single SliceM, and LC.

Table 3.1: Summary of the resources in a single CLB of Xilinx Virtex-4, Virtex-5, Virtex-6, and Virtex-7 extracted for vendor's user guides

Resource Type	Virtex-4	Virtex-5	Virtex-6	Virtex-7
Slices	4	2	2	2
LUTs	8	8	8	8
Flip-Flops	8	8	16	16
Arithmetic & Carry Chains	2	2	2	2
Distributed RAM (SliceM only)	64 bits	256 bits	256 bits	256 bits
Shift Registers (SliceM only)	64 bits	128 bits	128 bits	128 bits

Recent FPGAs pack even more logic resources per CLB; for instance Table 3.1 shows that Xilinx Virtex-5 includes about double the resources of a Virtex-4, Virtex-5 and subsequent families are based on using 6-input LUTs instead of 4-input LUT used in Virtex-4 [33]. This has increased the functional capacity of a single slice. CLBs and other components within the FPGA are interconnected via a network of programmable connections arranged in a grid.

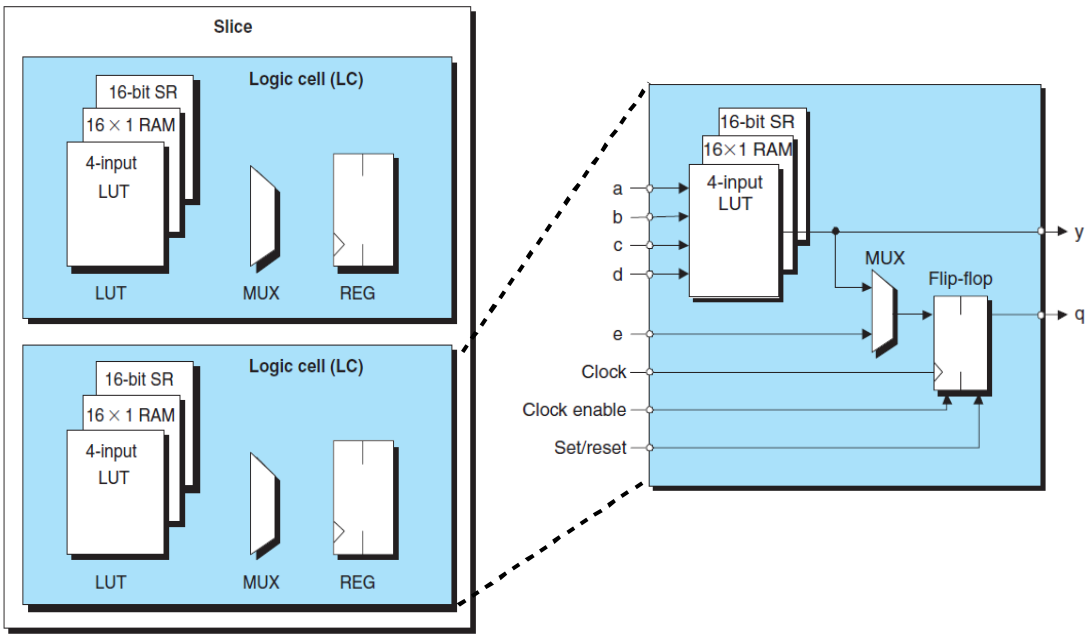


Figure 3.3: *The components of one Virtex-4 slice, (SliceM) and components of a logic cell, illustrating that LUTs can be configured as 16 bits shift register or a distributed RAM (source ref. [33]).*

Although CLBs and IOBs are the main required components to form digital circuits, modern FPGAs include various other dedicated and hard IP blocks which are capable of transforming FPGAs to complete SoC architectures. Block RAMs and DSPs are among the essential hard wired components integrated into modern FPGAs. DSP blocks are capable of functioning as fast multipliers, multiply-accumulate circuits (MACs), which are particularly useful for signal processing applications. The number of CLBs, Block RAMs, and DSPs vary across the numerous FPGAs within a device family; such variability allows a user to select the FPGA most suitable for the application in hand. For instance, some applications require large number of DSPs, Block RAMs and small CLBs while other applications require the opposite. Consequently, the user chooses the device according to the needed CLB density and resources required to realise the required functionality. Table 3.2 illustrates the variability in the amount of resources among the three platforms of Virtex-4, with each platform constituting a family of FPGAs having different combinations of resources and densities.

Modern FPGAs have been characterised by abundant Block RAMs to cater for local memory demands by many applications. Many applications require the storage of the input data locally to facilitate parallel computing, and in many cases intermediate results need to also be stored locally and accessed frequently. Block RAMs can be customised to work as FIFIOs, single or dual port memories, or as ROMs. Additionally, the LUTs of a single SliceM can be configured to store 16 bits of memory; this type of memory is referred to as distributed memory and can be used as RAMs or ROMs.

Table 3.2: The resources available in FPGAs from three Virtex-4 device families

Resources	Virtex-4 LX	Virtex-4 FX	Virtex-4 SX
Logic	14 – 200 K	12 – 140 K	23- 55 K
Memory	0.9-6 Mb	0.6-10Mb	2.3-5.7 Mb
DCMs	4-12	4-20	4-8
DSPs	32-96	32-192	128-512
I/Os	240-960	240-896	320-640
Power PC	N/A	1-2	N/A
Rocket I/O	N/A	0-24 channels	N/A
Ethernet MAC	N/A	2 - 4	N/A

Other resources which have been integrated into modern FPGAs are soft or hard core processors, FPGA vendors use different processors as shown in Table 3.3. Soft core processors are not physically embedded into the FPGA, but are inferred from the HDL instant instantiated into the main HDL design. The processor is then physically realised into FPGA primitives during synthesis and design implementation. The main advantage of such cores is that they are instantiated only when needed, consequently they do not occupy unnecessary footprint when they are not required in the design. On the other hand, hard core processors such as Power PC 405 (RISC processor) are physically embedded into the FPGA occupying an area footprint. The integration of processors into the FPGA fabric has increased the popularity of FPGAs and widened the range of their applications which include hardware/software co-deigns or FPGA coprocessors [32]-[39].

One of the additional hard wired blocks integrated into FPGA is Digital Clock Manager (DCM) which is used to generate deskewed clocks, shift the phase of the global clock, synthesise clocks of different frequencies. Similar to other blocks, the number of DCMs varies from one FPGA device to another. Fig 3.4 illustrates the distribution of the hardware resources in one of Xilinx' Virtex-4 FPGAs, namely *XC4VFX12*, which is available in the ML 403 platform board used in testing the implementations covered in the next three chapters [32]-[36].

Table 3.3: A list of commonly used FPGA-based processors as cited from [36]

Processor Name	Type/Bits	Interface Bus	FPGA Vendor
MicroBlaze™	Soft/32	IBM Coreconnect	Xilinx
NIOS®	Soft/32	Avalon	Altera
LatticeMicro32	Soft/32	Wishbone	Lattice
CoreMP7	Soft/32	APB	Actel
ARM Cortex-M1	Soft/32	AHB	Vendor Independent
LatticeMicro8	Soft/8	I/O ports	Lattice
Core8051 s	Soft/8	APB	Actel
PicoBlaze™	Soft/8	I/O ports	Xilinx
PowerPC	Hard/32	IBM Coreconnect	Xilinx
AVR	Hard/8	I/O ports	Atmel

Furthermore, current Xilinx FPGAs contain many other resources such as Ethernet transceiver I/Os, I/O banks, tri-mode Ethernet Media Access Controller–MAC, Internal Configuration Access Ports (ICAPs), and clock buffers.

FPGA interconnect technologies are of three types, the first is SRAM based adopted by Xilinx and Altera; the second is based on erasable Flash EPROM as adopted by Actel, and the third and least used is an Antifuse interconnect. SRAM based FPGAs are based on using SRAM to store the configuration and data associated with each LUTs. They are volatile and require continuous power to maintain their configuration. On the other hand, Flash FPGAs are non-volatile requiring less power than SRAM based FPGAs. As for the Antifuse interconnects, they can only be programmed once leading to using them in specific application which do not require re-programming such as in aerospace or defense [36].

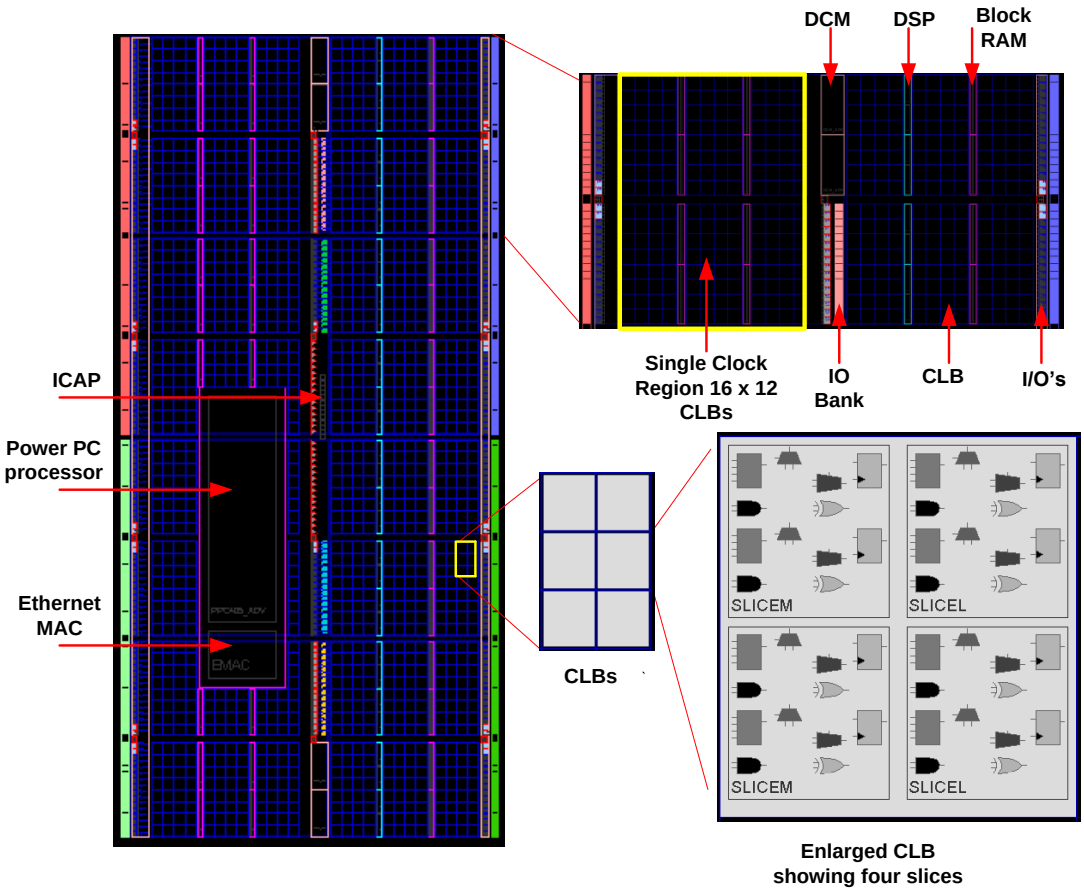


Figure 3.4: *The main resources available in a Virtex-4 FPGA, namely XC4VFX12, highlighting the columnar architecture, arrangement of CLBs, Block RAMs, DSPs, and IOBs. The device consist of eight clock regions, one clock region is enlarged spanning the height of 16.*

3.2.2 Applications of FPGAs

FPGAs have been replacing ASICs in some applications due to their short time to market, and instant re-programmability feature which are attracting applications requiring frequent updates or total changes in functionality. Additionally, FPGAs are more cost effective than ASICs when targeting products of small volume or meant for use in particular short term projects. Accelerating some applications is another main market for FPGAs, which span wide range of fields including digital signal processing, BCBs, medical imaging, image processing, SoCs, network security, HPC, aerospace military, cryptanalysis, and many more. Furthermore, FPGAs are heavily involved in the control systems of some of the aforementioned applications due to their high performance and integration of embedded processors.

3.3 Mapping Algorithms onto FPGAs

Designing FPGA based systems is an involved process which requires a level of expertise and training. Sophisticated set of electronic design automation (EDA) tools are employed in the design and verification phases. All FPGA vendors provide suite of EDA tools necessary for FPGA design which include simulation, synthesis, implementation (included translate, map, place and route), and the generation of the configuration file needed for download to the FPGA. In addition, EDA suites may include a debugging tool, and FPGA floorplanner tool, Table 3.4 summarises common design tools provided by two of the leading providers of FPGA: Xilinx and Altera. The following subsection will provide more details about the design flow of a FPGA based system using Xilinx design suite.

Table 3.4: The main FPGA design tools provided by two of the leading FPGA vendors, as cited from [36]

Functionality	XILINX	ALTERA
Design synthesis, mapping, place and route	ISE™	Quartus II®
FPGA Embedded Processor Design	EDK®	SoPC builder®
Custom Peripheral Support	Yes	Yes
Debug using On-Chip signal Analyser	ChipScope™ Pro	SignalTap®
MATLAB® co-simulation and IP cores library	System Generation™	DSP Builder®

3.3.1 Design flow

The procedure of designing FPGA based system is summarised in Fig 3.5, which highlights the following design steps: capturing the algorithm in HDL (design entry), simulating the design, implementing the design, generating the configuration file, and verifying it through debug and download to the FPGA.

Capturing the algorithm in HDL language such as Verilog or VHDL is the main step in design entry, any text editor can be used for that, however all FPGA vendors include such editors as part of their design suites. Prior coding in HDL, the user need to consider requirements for hardware design such as precision of the data throughout the algorithm, parameterisation, synchronisation, registering I/O's to facilitate pipelining, and parallelism. Additionally, the user needs to be aware of HDL constructs that are not synthesizable which include some data types e.g., real, realtime, and trior; continuous assignment such as delay; procedural assignments e.g., fork/join, release, forever, and event @; compiler directive e.g., Timescale, uselib, resetall, and any others. Such non-synthesisable constructs can only be used in pre-synthesis simulation, however if used in the actual design, they would be trimmed out by the synthesis tool which will harm the intended functionality. Post synthesis simulation is more important than pre-synthesis and projects more realistic timing analysis as its results are based on actually realised hardware. HDL is the best way used for capturing the parallelism in any algorithm as it permits parallel execution of operations which can be either synchronous or combinational [34]-[36].

Coding style has an enormous impact on the area and timing performance of the design, as such many FPGA vendors recommend coding styles appropriate for inferring the logics required for some operations such as arithmetic, counters, accumulators, Block RAMs, and many others. Using the recommended styles suggested by the vendors leads to highly optimised design. Most vendors provide language templates to assist designer in complying with best coding styles. As a consequence of applying language templates, the resulting area footprint and attained frequency are improved significantly. Over and above, some FPGA vendors such as Xilinx have integrated a tool called CORE Generator™ into its ISE design suite, which provides an extensive library of customisable IP cores including math functions, DSP algorithms, bus interfaces, memories, and many others. CORE Generator™ can be invoked from within ISE or externally, in both cases a wizard will assist the designer through the steps required to create and customise the IP for the targeted FPGA. Fig. 3.6 is an

example of a Xilinx Core Generator™ wizard invoked from within ISE to create a divider core [32]-[36].

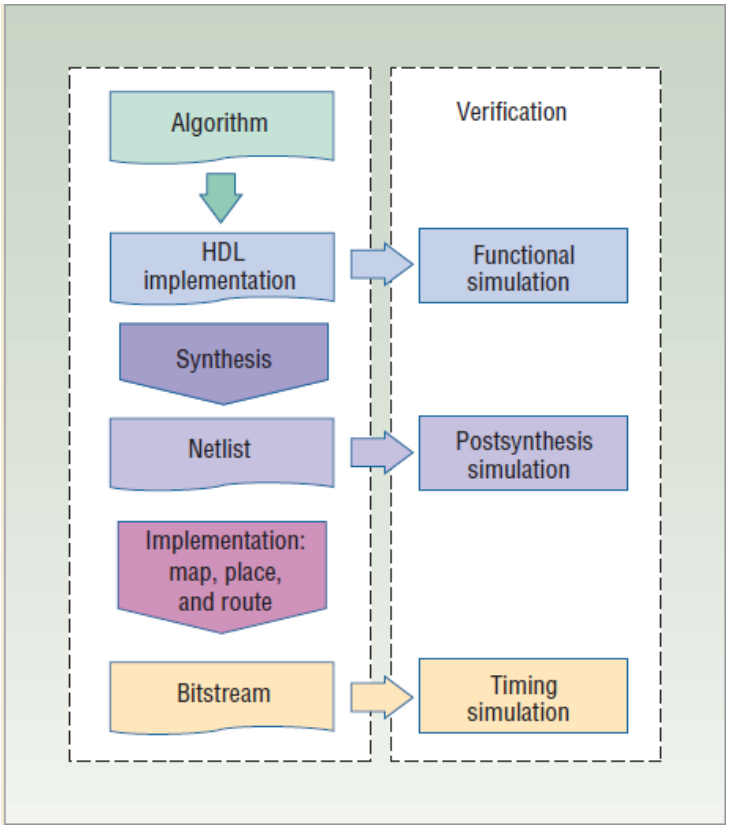


Figure 3.5*: Flowchart of FPGA designs (source ref. [30]).*

Additionally, vendors also provide libraries of HDL codes which can be inserted into the HDL code of the design Wrapper to instantiate specific device primitives such as CLBs, RAMs, ROMs, DCM, DSPs, soft/hard processor cores or any other device primitive. According to Xilinx, instantiating device primitives ensure optimum design performance.

Furthermore, there exist many resources providing libraries of IP cores that can be added to the design to capture specific functionalities. Some of those IP cores may be obtained free of charge from vendor's websites, or from other developer's websites with the requirement that the user acknowledge the source of the IP core to preserve the rights of the developer. Many IP cores can also be purchased from developers at some cost. IP cores are becoming an important industry market nowadays and growing at a rapid pace, some of those IP cores are forms of patents. Worth noting that many FPGA vendors allow mixed HDL language

designs, that is combining cores or primitives of verilog and VHDL. This feature facilitates the compatibility for integrating wide range of primitives and various IP cores.

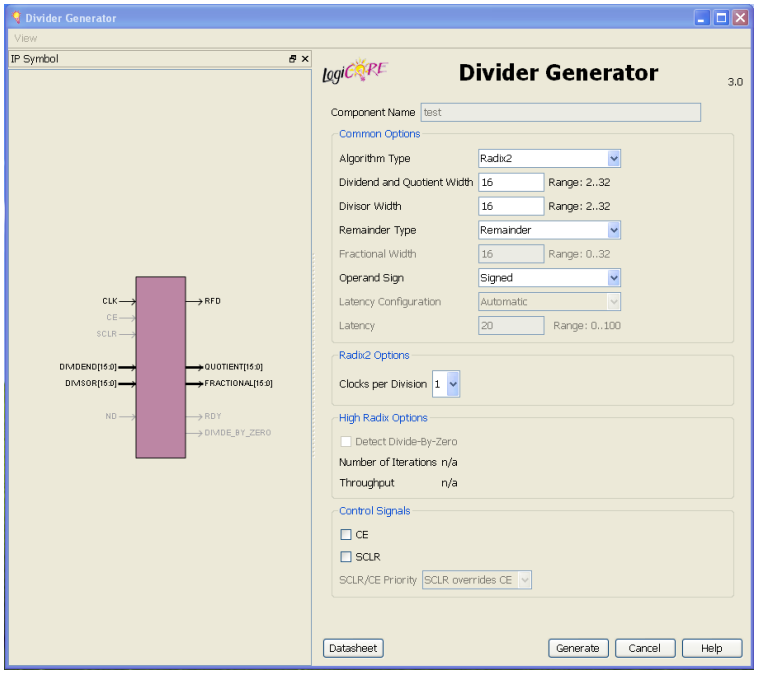


Figure 3.6: *A view of Xilinx Core Generator wizard used to generate a Divider, whereby the core is generated based on user entries.*

Following design entry, the functionality of the algorithm is then simulated using any simulation software such as Xilinx ISim simulator, Mentor Graphics Modelsim or others, to verify that the HDL code is functioning properly and doing the intended operations. Simulation involves creating testbenches for supplying test data and control signals to the design and checking the outputs throughput the design stages using waveform viewer within the simulation tool. As mentioned earlier, there are two types of simulations one is pre-synthesis while the other is post-synthesis to assist in validating the design entry.

After a successful design entry, the design is synthesised to generate a netlist file called NGC (Xilinx specific netlist), which is basically a realisation of the logic required to implement the design entry. The realised logic constitutes of combinational logic and macros such as multiplexers, flip-flops, adders, subtractors, counters, FSMs and RAMs. The synthesis tool tries to infer the largest possible macros, remove any constructs that are non-realisable in hardware and check the correctness of the design entry; the removal of non-synthesisable constructs is referred to as design optimisation. The created NGC file is

basically a combination of two things: the logics of the digital circuit and the constraints specified by the designer to help the synthesis tool meet design goals e.g., area, speed, or the inference of specific device primitives. The contents of the NGC file are based on two separate files, one is referred to as User Constrain File (UCF), while the other is the electronic data interchange format (EDIF) which contains the data of the logical design [32].

Following a successful synthesis, a post-synthesis simulation is then carried out to verify the design before actual implementation. Actual implementation constitutes of three main phases: Translate, Map, Place and Route. During Translate phase, the implementation tool, namely, NGD BUILD, converts the NGC file to NGD netlist, whereby the latter contains approximations about switching delays and is dependent on a SIMPRIM library as opposed to UNISIM in the former. The Map phase then converts the NGD netlist into physical device primitives or resources such as LUTs, flip-flops, Block RAM forming CLBs, I/Os, or any others. The MAP program stores the results into NCD file containing precise results of switching delays. The Place and Route phase is then performed by a PAR program, which actually places the inferred primitives onto the selected FPGA, connects them together, and finally reports the timing of signal's propagations. The Place and Route phase leads to actual design being laid out on the selected FPGA, other FPGA tools can then be used to view the actual placed and routed design such as Xilinx PlanAhead and FPGA Editor, which both come as part of the Xilinx ISE design suite. During every phase, the implementation tool generates specific files containing some details of things such as timing, resources and routings. These files can be used by other Xilinx tools such as Floorplanner, FPGA Editor, and Xpower Analyser, for details about those tools, the readers is referred to Xilinx documentations [32].

The implementation process is fairly straightforward and is done automatically by the tool, it uses default options unless the designer specifies other options. There are plenty of additional options that can be chosen by the designer to help meet design goals. Although the PAR program applies placement algorithms to locate the resources such that design goals are met, the user have the option to constrain the placement of some resources to particular locations on the FPGA, this is referred to as area or placement constraints, other timing and synthesis constrains are also permitted by the tool. Design constraints can be specified in text using UCF file, or can be specified in PlanAhead using GUI which then infers the contents of the UCF file automatically. Current FPGA EDA suites offer comprehensive options and tools that allow for analysing the placed and routed design in terms of timing, area, power

consumptions, and viewing technology or RTL schematics of the synthesised design to visualise the inferred digital circuits [32].

Having implemented the design successfully and met design goals, the next step is to generate the bitstream file using Bitgen tool available in Xilinx' ISE suite, which is a file of binary codes containing the placed and routed design used for configuring the FPGA. The bitstream can be full or partial whereby the former is used to configure (program) the whole FPGA, while the latter is used to configure specific portion of the device, more details will be given about partial reconfiguration in the subsequent section.

The configuration file (bitstream) can be downloaded from host to the FPGA via JTAG, PCIe, serial links, or any other modes by invoking Xilinx iMPACT tool which is particularly used for configuring the FPGA. Other configuration files could be created according to user preferred configuration mode and location, for instance, the FPGA could be configured via PROM or SystemACE attached to the FPGA. Furthermore, Xilinx allows for configuring the FPGA using some of its other tools such as EDK suite, PlanAhead, or ChipScope™ Pro. Table 3.5 enlists the configuration modes available for Virtex-4 and subsequent devices [32].

Table 3.5: Configuration mode available for Xilinx Virtex-4 and subsequent devices

Configuration Mode	Data Width (bits)
Master Serial	1
Slave Serial	1
Master SelectMAP	8
Slave SelectMAP8	8
Slave SelectMAP32	32
JTAG/Boundary-Scan only	1

One of the current state of the art features of modern FPGAs is using on-chip analyser to debug the configured FPGA. An example of such analysers is Xilinx ChipScope™ Pro which allows the insertion of specific cores into the design prior to synthesis, those embedded cores are integrated into the bitstream which will be used in configuring the FPGA. Once the FPGA is configured, the designer can view various signals using a waveform analyser invoked from within ChipScope™ Pro. The latter is an on-chip signal analyser which allows the designer to debug the design efficiently without having to use external signal analysers, and assist in verifying the design by monitoring the output signals and checking them against simulation results [32], [34]-[35].

3.4 Dynamic Partial Reconfiguration (DPR) on FPGAs

Most applications of FPGAs have been based on static configuration that is downloading a full bitstream to the FPGA. As such, altering the functionality of the hardware involves modifying the HDL code, re-implementing the design, and generating a new bitstream to reconfigure the FPGA which then require re-booting the FPGA. Recent evolution in FPGAs has led to the possibility of modifying the configuration of the FPGA fully or partially while algorithms are being executed without the requirement of system re-boot. Dynamic partial reconfiguration (DPR) is a design flow which enables changing subset of the FPGA configuration while the device is operating; the reconfiguration overhead is subject to the amount of changes made to the device configuration. This feature is offered by some of the current FPGA vendors, the following overview about DPR is based on Xilinx DPR flow.

3.4.1 Methodology and Considerations of DPR

The first step in the DPR design is to implement modular hierarchical design, which is based on having a top design consisting of several partitions or submodules each performing specific tasks. Secondly, a bottom-up synthesis is applied, which is based on creating separate synthesis projects of the top design and each of its submodules resulting in multiple netlist (NGC files). One of the main design considerations is to disable the I/O insertion in the synthesis option when creating the netlists of the submodules meant to be dynamically reconfigured, and keep the option checked when creating the netlist of the top module. Submodules which are not meant to be partially reconfigurable, along with logic embedded in the top design form the static logic which are both preserved throughout the partial reconfiguration. Third, Xilinx PlanAhead tool is invoked, and a new PR project is created targeting the same FPGA used when creating the netlists, the process of creating a PR project is facilitated by a wizard in PlanAhead. This process involves importing the netlist of the top-design, define Reconfigurable Partitions (RPs) which are basically the submodules meant to be reconfigurable, import the various netlists created for each partition constituting the logics of the RP regions, those are called the Reconfigurable Modules (RMs). Following this, the RPs are floorplanned by manually creating Pblock rectangles or specifying the range of CLBs and other resources in text, this intends to constrain each RP to specific location on the FPGA having the hardware resources needed. Additionally, timing constrains are specified at this point to meet design goals by importing UCF files. Once all constrains are specified, design is checked using *run DRCs* in the PlanAhead tool [37]-[40].

At this point the design has been structured, constrained, and checked, the next stage is to create configurations, then place and route them. Creating a configuration is a process of creating a complete design constituting the static region and one RM for each RP as shown in Fig. 3.7. To create a configuration, the run window in PlanAhead is used to specify a name for the configuration, and select the RMs for each RP from a drop down menu which enlists all possible RMs for each RP. Then the design is run to invoke Xilinx implementation tool to place and route the configuration. The Xilinx Bitgen tool is then invoked to generate two bitstreams, one is a full and the other is partial. The partial bitstream is used to partially reconfigure the FPGA while the device is running. Having created the first configuration, the design is then promoted which will copy the static logic of the current implementation for use when creating subsequent configurations. The process of creating new configurations is then repeated as many times as needed to create different configurations based on different RM combinations. Finally, bitstreams are stored into host memory or in non-volatile off-chip memory such as Flash or SRAM for access by the configuration system as needed [39].

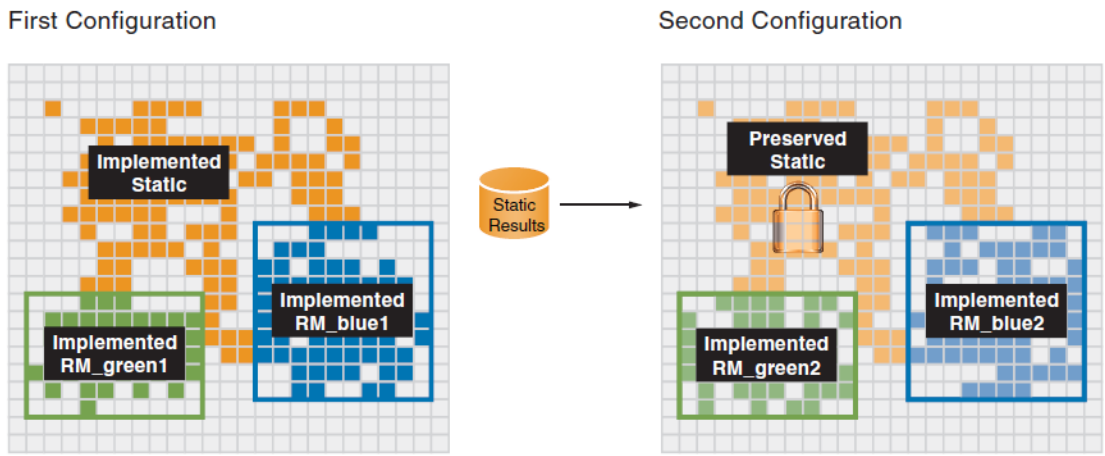


Figure 3.7: *Illustrative diagram showing the partial reconfiguration flow, highlighting the steps of creating multiple configurations based on different RMs (source ref. [40]).*

The FPGA is first configured with a full bistream using any configuration mode, then DPR is carried out using the partial bitstreams as needed. The most common modes for partially reconfiguring the FPGA are stated in Table 3.6 adapted from Xilinx documentations on PR, which shows that ICAP is the fastest method to reconfigure the FPGA. The designer determines the method for delivering the partial bitstream for the initiation of the partial reconfiguration, the two possible delivery methods to initiate the reconfiguration are: using off-chip processors such as a host GPP; using on-chip processor or state machine. When off-

chip processor is used, the possible configuration modes are serial and JTAG cables. On the other hand, when using on-chip processor, ICAP is used as configuration mode driven by a simple state machine or by soft/hard core processors such as MicroBlaze or Power PC which copies the bitstreams stored off-chip to the ICAP to reconfigure the RP, this is called self-reconfiguration, Fig 3.8 illustrates the reconfiguration process using both methods [40].

Table 3.6: The possible reconfiguration modes applicable to DPR and their performance details

Configuration Mode	Max Clock Speed	Data Width	Max Bandwidth
SelectMap/ICAP*	100 MHz	32-bit	3.2 Gbps
Serial Mode	100 MHz	1-bit	100 Mbps
JTAG	66 MHz	1-bit	66 Mbps

The smallest reconfigurable region forms the finest granularity for partial reconfiguration, which is referred to as reconfigurable frame. A reconfigurable frame spans the height of one clock region and width of one FPGA column. As explained earlier, Xilinx columnar architecture implies that a column consist of one kind of resources such as CLBs, Block RAMs (BRAMs), DSPs, or IOBs. Such columns are variable from one device family to another; Table 3.7 illustrates the resources reconfigured in each frame for various Xilinx device families [39].

Table 3.7: Reconfigurable frames of different Xilinx device families, illustrating the resources affected when reconfiguring a single frame in one column whereby a column contains only one kind of resources

Device Family	No. of CLBs	No. DSPs	No. BRAMS	No. IOBs
Virtex-4	16	8	4	32
Virtex-5	20	8	4	40
Virtex-6	40	16	8	80

Reconfiguring a FPGA involves reconfiguring the entire frames associated with the RP including the static logics placed in those frames, even if a frame is partially selected as shown in Fig. 3.9. Consequently, it is best practice when specifying a Pblock rectangle to attempt constraining the RP to a single full clock region instead of using multiple partial clock regions as this will involve double the reconfigurable frames of a single clock region.

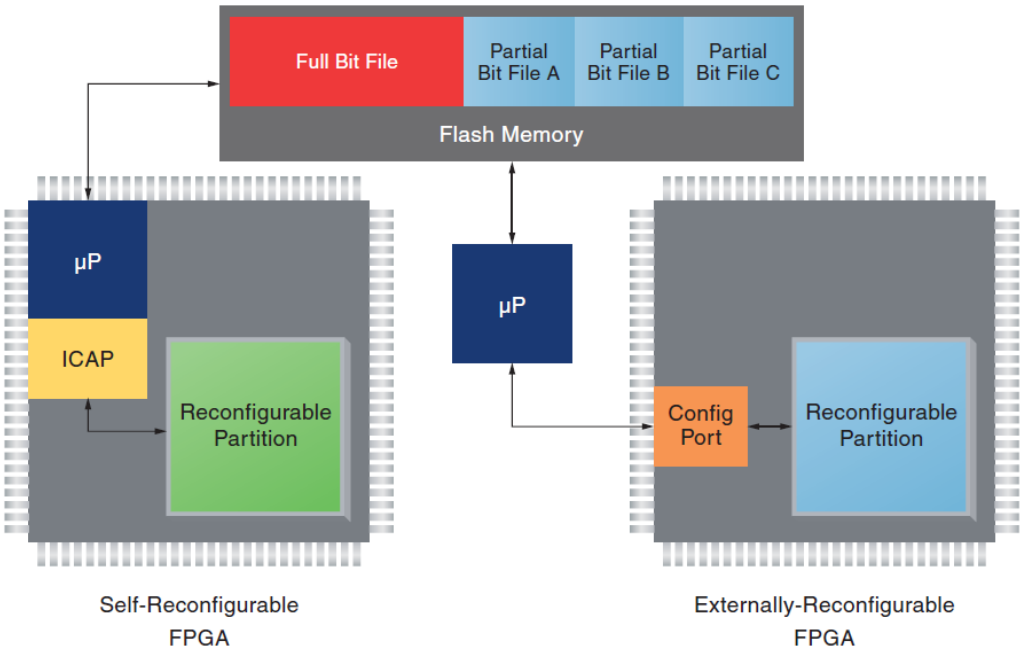


Figure 3.8: *Methods to reconfigure the FPGA (source ref. [40]).*

Another significant design consideration which will be of particular importance in subsequent chapters is to ensure that the I/O of all RMs associated with specific RP are consistent in number and in data width. Incompatibility of any RM with the one specified in the top design will result in failure of the implementations. There are many other design considerations and issues which are not detailed here, the reader is referred to vendor documentations on PR design flow for in depth information [38].

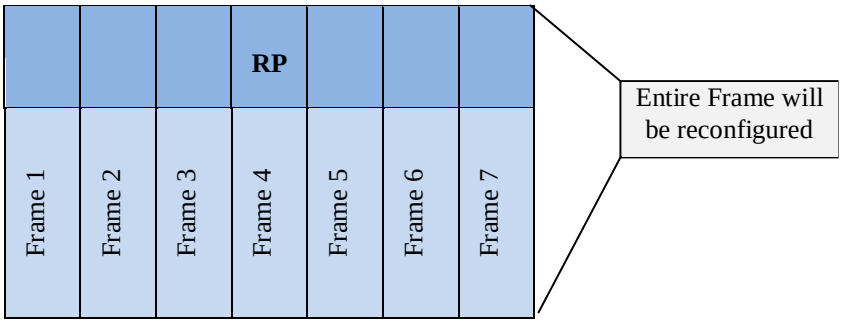


Figure 3.9: *Reconfiguring FPGA involves Reconfiguration frames spanning a clock region height, and a single columns width.*

3.4.2 Potential Advantages of applying DPR

The main features enabled by DPR are system flexibility in performing multiple operations, whereby the hardware resources within the FPGA can be time multiplexed to perform multiple operations. Consequently, enabling the implementation of large systems which otherwise cannot be implemented in a given device when hardware resources are limited. Additionally, time multiplexed tasks can result in small area footprint, allowing the integration of various tasks onto the same FPGA.

Beside gains in area footprint, DPR allows for shutting down power hungry tasks when they are not particularly needed. Some applications which use the FPGA as a communication hub among several systems such as video, audio, or network links do not permit the full reconfiguration of the FPGA; as such DPR is the only permissible option to modify specific hardware configurations without disabling the communication links or causing interruptions. Additional power saving can be achieved when replacing high power functions with low power functions when performance is not affected or if maximum performance is not required [40].

Over and above, the capability to reconfigure specific area within the FPGA while other tasks are running increases the fault tolerance of the system. For instance, when a fault occurs within specific RP region, reconfiguration is usually required to repair the fault and re-commission the system. Instead of reconfiguring the whole FPGA, DPR can be used to reconfigure the faulty region only while the un-faulty regions remain on operation, once the fault is repaired, the system will be back in full operation without having to re-boot [40]. Those advantages of DPR are applicable to the specific data mining implementations covered in the next three chapters; and additional benefits will be pointed out as they come.

3.5 The ML 403 Platform board

The ML 403 is a FPGA platform board which includes Xilinx' XC4VFX12-FF668-10 FPGA. The board includes plenty of off-chip resources such as memory, communication ports, general purpose I/Os (GPIO) e.g., push buttons and LEDs, 100 MHz clock source. Fig. 3.10 and Fig. 3.11 illustrate the main components and architecture of the ML 403 board. The off-chip memories available on the board are compact Flash (CF), 8MB Flash, 64 MB DDR SDRAM (based on two 32 MB), 8 Mb ZBT SRAM, and 4 Kb IIC EEPROM. Supported configuration modes are JTAG, master/slave serial, master/slave SelectMAP. The main

connection ports available on the ML 403 are the RS-232 serial, JTAG, PS/2 mouse and keyboard connectors, VGA output, tri-speed Ethernet transceiver, and host/peripheral USB.

Additional resources are available which include Microphone input, headphone output, LCD, and expansion headers. The ML 403 is used in testing the implementations covered in the subsequent chapters; a JTAG cable is used to establish the communication link with a host GPP. Xilinx ChipScope™ Pro is used to debug the designs and verify correct functionalities of the top design as well as the individual submodules.

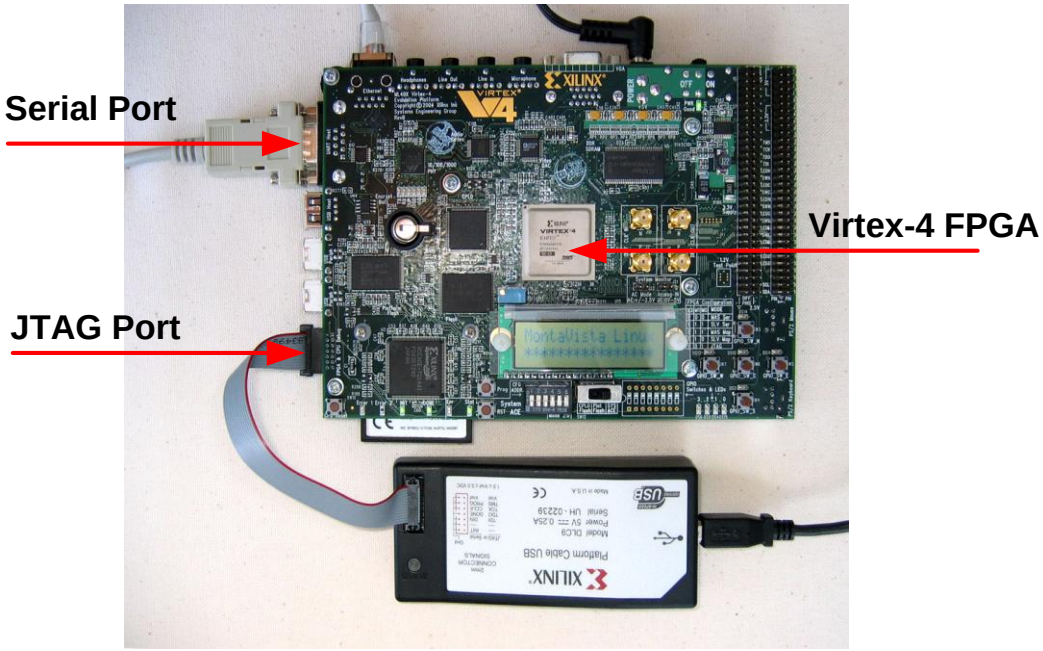


Figure 3.10: *Top View of the ML403 board highlighting the location of the FPGA, the serial and JTAG ports (source ref. [41]).*

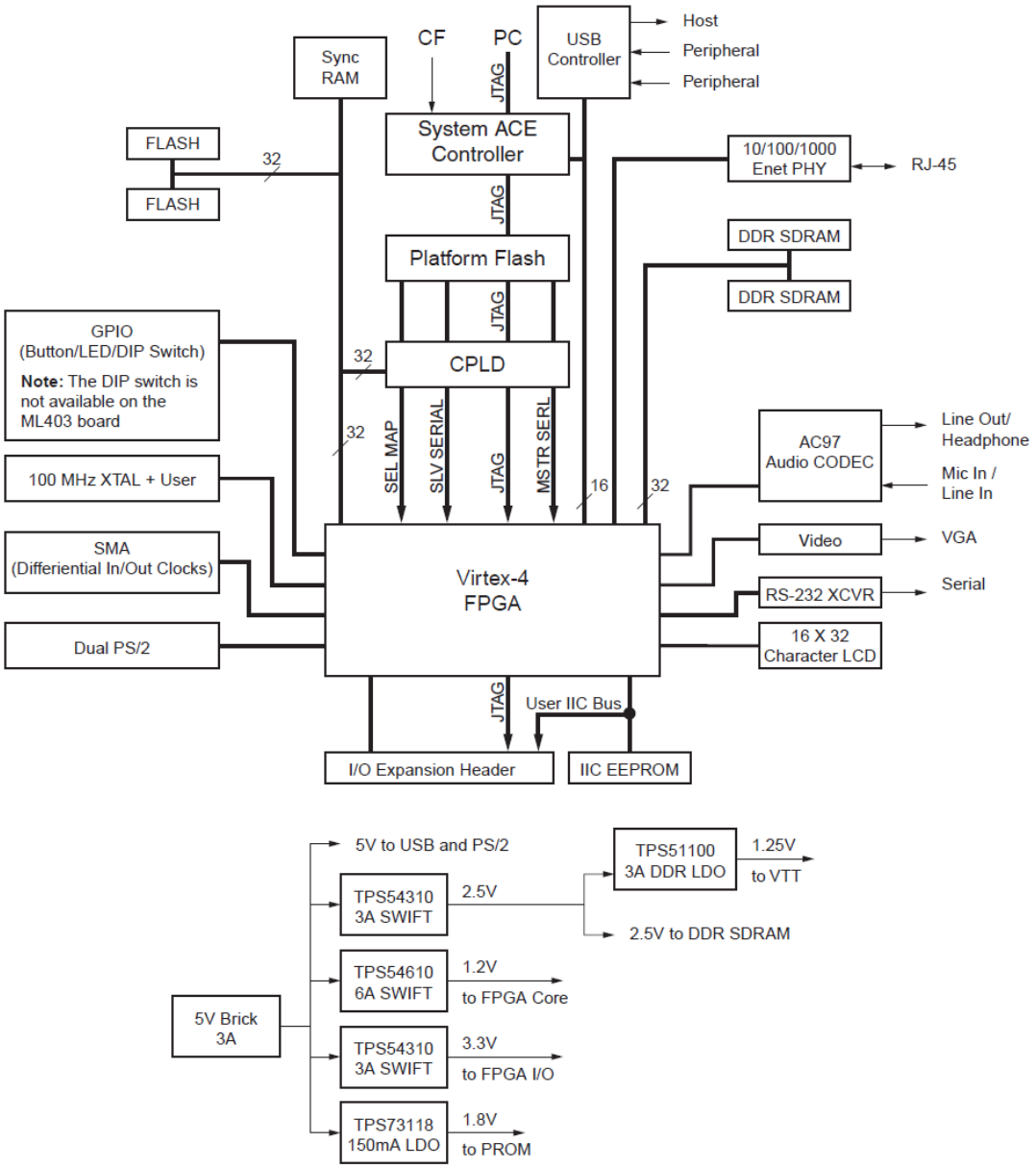


Figure 3.11: *Block diagram of ML403 board (source ref. [41]).*

3.6 Summary and Conclusions

A short overview about the history and circumstances which have led to the development of FPGAs was given in this chapter. Then a description of the ASMBL™ architecture of Xilinx' Virtex FPGAs was given. Following this, an overview about the main resources or device primitives was given highlighting their types, functions, and distribution within the FPGA. It has been established that the smallest logical component within a CLB is the LUT which can be configured to implement logical operations, shift register, or memory. Virtex-4 devices are based on 4-input LUTs while subsequent Virtex families built around 6-input LUTs. Advancements in semiconductors following Moore's law have been contributing to rapid evolution in FPGA technology enabling packing denser transistors within the FPGA and embedding hard IP cores within the FPGA. Today FPGAs include rich heterogeneous resources enabling their integration into wide range of applications, this have been catalysed by their on-the-spot re-configurability feature, short time to market, and high performance.

In addition, the FPGA design flow was covered briefly in this chapter, introducing design methodology and EDA tools required to carry out design entry, pre and post simulations, synthesis, implementation, bitstream generation, system configuration and design debug. Following this, a brief overview of current FPGA applications was given, including HPC. Leveraging the parallelism offered by the fine granularity of FPGAs and the high potentials of pipelining has made FPGA popular in a variety of applications and by various disciplines.

Furthermore, an overview of DPR technology was given highlighting its advantages and main features. Then, the design flow was laid out along with significant design considerations. In summary, DPR is an expert design flow which involves following stringent design requirements to create configuration files to be used to partially reconfigure FPGAs. The reconfiguration methodology is based on selecting the most appropriate configuration mode for the application in hand and on selecting the method for initiating the reconfiguration of the FPGA which is based on using off-chip or on-chip processors.

Lastly, the ML 403 FPGA board was briefly introduced highlighting the main off-chip components and connections to host. The board includes various off-chip resources such as memories and communication ports.

Chapter-4

Hardware Implementation of the K-means Clustering on FPGA

4 Hardware Implementation of the K-means Clustering Algorithm on FPGA

4.1 Introduction

The K-means clustering algorithm is one of the most widely used unsupervised data mining techniques to analyse large datasets and extract useful information from them. In bioinformatics, K-means clustering and hierarchical clustering are the most popular methods [42]. K-means is a parametric and iterative algorithm which involves grouping objects into smaller partitions called clusters, where objects in the same clusters are believed to share some form of similarity. The algorithm requires the pre-determination of the number of clusters beforehand and the initialisation of the cluster centroids. The computational complexity of the K-means clustering algorithm is dependent on the number of objects in the dataset (N), number of dimensionality (M), number of clusters (K), and number of iterations (t), such that the complexity is O(tKNM), with t typically much smaller than N [42].

Clustering Microarray data using K-means clustering has been integrated as part of many gene expression analysis software packages to measure relationships between genes or samples by grouping together genes sharing similar patterns or behaving in a coordinated manner [7]. K-means has been helping scientists to identify gene markers linked to some diseases, study the interaction between genes, regulations and pathways. In addition, K-means is used to discover gene functionality, and study the effect of specific treatments e.g., effect of some chemotherapy drugs on cancerous cells [7], [42]-[43]. Unfortunately, applying K-means clustering to Microarray data is computationally intensive and requiring long execution times when GPPs are used. However, large Microarray datasets contain a lot of hidden and unveiled information due to the insufficiency of current GPPs in exposing this information mainly as a result of the limiting number of expression profiles that can be analysed simultaneously. As a result, a lot of high potential benefits of Microarray datasets have not been utilised or fully discovered. The fact that GPPs have not been able to keep up with the high computational demands of most bioinformatics applications mainly due to reaching limits in terms of clock frequency and power requirement is discouraging scientists from attempting to ask complex biological questions. Consequently, accelerating K-means by means of hardware implementation is inevitable and is expected to have high impacts on

Microarray research allowing complex problems to be finally approached at acceptable execution time.

FPGAs have been successfully applied in the acceleration of many bioinformatics applications such as in bio-sequence alignment, phylogenetic analysis, and molecular dynamics simulation [26]-[27] and [44]-[46]. The reader is advised to consult the aforementioned references for details about those applications. Generally, FPGAs have proven to be effective in reducing the execution time of many bioinformatics applications. In addition, several numbers of groups have successfully accelerated K-means clustering using FPGAs for other applications such as hyperspectral imaging. Nevertheless, the acceleration of K-means in FPGAs to target Microarray datasets has not been reported in literature, although this application is highly candidate for such platform, and is due to greatly benefit from it. Furthermore, some specific applications of K-means clustering such as cluster ensemble which requires repeated runs is due to benefit from the parallelism offered by state-of-the-art FPGAs. Ensemble clustering based on combing several runs of the K-means has been associated with improved performance in terms of accuracy as reported in [47].

The remainder of this chapter will first present some background on K-means clustering algorithm followed by an overview of prior work in the area of hardware implementation of K-means clustering. Second, requirements for hardware design will be discussed detailing the pre-processing steps applied to Microarray datasets, the processes associated with converting the dataset and the GPP software to fixed-point format. Third, the design and architecture of the novel K-means clustering algorithm will be detailed, this part consists of three novel implementations of the K-means in FPGA harnessing DPR technology of current state-of-the-art FPGAs. Following this, the results of the various implementations will be presented, and the results of comparing the performance of the proposed implementation with equivalent implementations running on GPP will also be presented. Then, a comparative study of the performance of the proposed FPGA implementation of the K-means with GPU will be presented and analysed. Finally, the power consumption and energy efficiency of the K-means implementations in GPP, FPGA, and GPU will be compared. Following this, summary and conclusions will be laid out with plans for future work.

4.2 Background on the K-means Clustering Algorithm

K-means clustering has been helping scientists in many fields of studies in extracting relevant information from large datasets. The algorithm aims at minimising the within cluster

variance and maximising the intra cluster's variance to ensure best separation of objects. The first step in performing K-means clustering is to determine the number of clusters and initialise centroids for each cluster from the given dataset. The pre-determination of the number of clusters has been a complex task and almost impossible to predict, thus most works relied on trial-and-error to determine the number of clusters [43].

There are several ways for doing this initialisation, one way is by randomly assigning all points in the overall dataset to one of the pre-defined clusters, then calculating the means for each cluster and use them as the new centroids for these clusters. Another way to do the initialisation is by randomly selecting cluster centroids from the whole dataset. There have been several efforts reported in the literature to optimise the initialisation of the clusters to enhance the performance of the algorithm. Some optimisations have reduced the computation time due to reaching convergence at smaller number of iterations; however, the initialisation may take longer time in some of these techniques depending on the complexity involved. An example of such optimisations is based on running the clustering several times, and selecting the cluster centroids resulted in the best performance as the initial center for subsequent runs [47].

Another example is the use of sampling methods such as rejection sampling, which assists in cleverly selecting initial cluster centroids by selecting a point from the dataset randomly and computing the distances between this point and a K number of randomly selected cluster centroids. Then, the resulted K distances are checked against the values of the current K centroids to determine the new centroids to be used in the actual K-means clustering, the new centroids takes the coordinates of the selected point if the current centroids are smaller than the distances, otherwise that point gets rejected from being candidate for a new cluster centroid, the process gets repeated several times [48]. It is important to note that the process of random initialisation is very difficult to do in hardware since data are often represented in fixed-point format having specific precision. Therefore, initialisation is done as part of the off-chip pre-processing step which will be explained in upcoming sections.

After initialising the cluster centroids, the second step is to compute the distances between each point vector in the dataset and every cluster centroid using a distance metric such as Euclidean, Manhattan, Max, or any other one, this computation will result in number of distances equivalent to the number of clusters (k). The next step is to find the minimum distance among the K results and assign the point vector to the closest cluster; this step is called cluster assignment or cluster classification [48]-[49]. Once the point vector has been

assigned to one of the clusters, a counter associated with that particular cluster gets incremented and that particular point vector gets accumulated. When all point vectors in the dataset have been assigned to the clusters, the average is computed based on the total accumulated points and the number of points in each cluster. These results become the new centroids for the subsequent iteration, this step is referred to as *centroids update*. The process of re-assigning point vectors in the dataset to one of the clusters continues until an ending condition is met, which can be iterating for a fixed number of times or until the new cluster centroids become the same as the current centroids such that points stop moving to other clusters; this condition is called *convergence*. The above steps of the K-means clustering algorithm could be summarised in a flowchart as in Fig. 4.1.

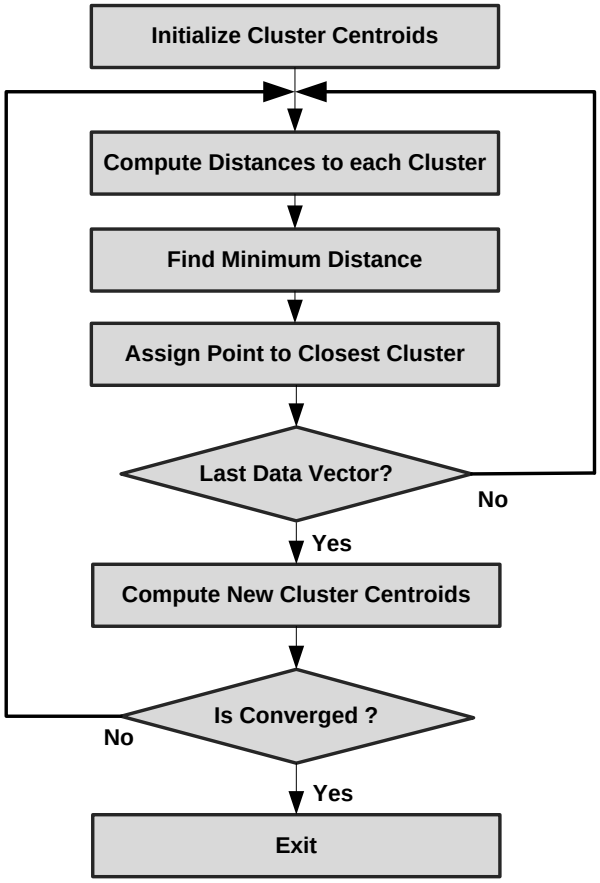


Figure 4.1: *Flowchart illustrating the steps of the K-means clustering algorithm when implemented in software, the same steps are followed for the hardware implementation.*

The distance computation kernel can be performed using several distance metrics, however Euclidean metric illustrated in equation (4.1) is considered one of the most widely used distance metrics in K-means clustering and one that provides good accuracy [50].

$$D(P_i, C_i) = \sqrt{\sum_{i}^{M} (P_i - C_i)^2}, \tag{4.1}$$

where P is the data point, C is the cluster centroid (both are vectors), and M is the number of features or dimensions. Euclidean distance has the advantage of minimising the distance and the within cluster variance. On the other hand, Euclidean distance consumes a lot of resources when implemented in hardware as a result of the multiplication operation used to compute the square which needs to be repeated many times reducing the amount of parallelism that can be exploited [50]. Consequently, previous groups working on hardware implementation of K-means clustering for image segmentation have experimented with using alternative distance metrics, a common choice was to use the Manhattan distance shown in equation (4.2) mainly due to its simplicity leading to lower implementation cost [11]-[13]. Furthermore, their results showed faster performance than the Euclidean metric, one reason was that it does not require multiplication offering better exploitation of parallelism and speed twice that obtained by Euclidean distance as reported in [50]. However, the same authors have reported that the accuracy of Manhattan distance was slightly inferior to the Euclidean metric, but were still within an acceptable error.

$$D(P_i, C_i) = \sum_{i}^{M} (|P_i - C_i|), \tag{4.2}$$

where P is a point vector, C is a centroid vector, and M is the number of dimensions.

The time needed to complete clustering a dataset depends on the size of data, number of dimensions, and the selected number of clusters, the larger they are, the longer it will take. Various works have profiled the computational times of the K-means clustering and identified the distance kernel as the most time consuming part. In [49], when clustering an image to 32 clusters, the author reported that the distance computational part of a single loop through the image pixels was consuming 99.6 % of the computation time. The fact that

distances computation is the most computationally demanding part and where most of the K-means processing time occurs has led to parallelising this part.

One reason which encouraged one of the groups who have worked in implementing K-means clustering on FPGA was the possibility of truncating the wordlength of the dataset and the cluster centroids without significant loss in accuracy. The group has conducted data dependent and data independent experiments to study the effect of truncating the input dataset on the K-means algorithm and investigate the quality of the clustering due to such truncation. They concluded that input data can be truncated by many bits and still have good quality clusters with the K-means algorithm [50]-[51]. To sum up, accelerating K-means algorithm can be reached by:

- *Accelerating the distance computation kernel using simpler distance metric such as the Manhattan distance requiring less time than the Euclidean.*
- *Moving the task of calculating distances to hardware such as FPGAs to benefit from parallelism offered by configurable logic.*
- *Carefully truncating the wordlength of the data without sacrificing accuracy.*

K-means has proven to be popular for clustering large datasets due to several reasons: first, is the time linearity of the algorithm with respect to the number of point vectors (N) when the number of clusters and iterations are fixed. Second, the possibility of storing the point vectors (or patterns) in secondary memory and access them on demand reducing the space complexity from O(K+ N) to O(K). Finally, due to being order-independent yielding the same partitions for the same initial cluster centroids regardless of the order in which objects are processed [10] and [52]. On the other hand, most other clustering methods were found to be suitable for small datasets only. For example, the hierarchical clustering has a time complexity of $O(N^2 \log N)$, and space complexity of $O(N^2)$ leading to more time consuming clustering than in the case of the K-means clustering. Although K-means has the disadvantage of yielding different clustering results at different runs due to the dependency and sensitivity of the algorithm on the initialisation of the cluster's centroids, it has gained a lot of popularity due to its capabilities to cluster large datasets [10] and [53].

The wide popularity of the K-means clustering algorithm in processing Microarray datasets and its candidacy to benefit from new frontiers in FPGA design have been the main drivers for this research. Recent advances in hardware design include EDK tools, denser logic resources, larger number of dedicated hardware resources, DPR technology, and larger off-chip resources in most medium and high end platform boards offering new frontiers to accelerate the K-means algorithm in one hand, and implementing new innovative design approach on the other hand. This research intends to implement a highly scalable and parameterised implementation of the K-means clustering algorithm on FPGAs using both Euclidean and Manhattan distance metrics. The reason to consider both distances as opposed to using Manhattan distance only as reported in the literature is that current state-of-the-art FPGAs contain abundant number of dedicated resources (Multipliers or DSP48 blocks) that render the Euclidean metric more affordable than it used to be with earlier FPGAs without compromising the execution time. In addition, three novel implementations of the K-means clustering which harness the DPR capability of recent FPGAs will be presented for the first time in literature.

4.3 Prior Work on Hardware Implementation of the K-means Clustering Algorithm on FPGAs

The work done in hardware implementation of K-means has been mostly in the area of multispectral and hyperspectral imaging with few in other areas such as document clustering and network anomaly detection. The following review is for most significant works on FPGA implementation of K-means clustering ordered chronologically.

In 2000, Dominique Lavenier at Los Alamos National Laboratory implemented systolic array architecture of K-means clustering algorithm on a number of FPGA boards [49]. The author of [49] aimed at parallelising the most computationally intensive part which is the distance calculation kernel by running the input through an array of Manhattan distance calculation units of numbers equal to the number of clusters. Each of the distance calculation units compute the distance between the data point and one of the cluster centroids, once the point finished propagating through the complete systolic array, the cluster assignment index is received at the end of the array. The remaining steps of the K-means algorithm were carried out by a host processor. The image stored in a host and streamed to the FPGA for distance computation, the results were then sent back to the host for accumulation, counting, and calculation of new cluster centroids, those new centroids were streamed back to the

FPGA for the next iteration. The process iterates until convergence is reached. This implementation was effective in allowing for any data size to be processed, however, the disadvantage was the communication overhead between the host and the FPGA. Lavenier tested his design on several processing boards and reported several speed-ups over pure software implementations based on MemMap and DMA transfers between host and FPGA boards, the speed-ups he obtained varied for different boards and different cluster numbers. For example, using 16 clusters resulted in speed-ups between 24x to 39x for different boards when using DMA as compared to speed-ups between 53x to 92x for the case of using 32 clusters which was also based on DMA transfer [49]. Higher speed-ups were obtained for larger number of clusters. On the other hand, when MemMap transfer was used the reported speed-ups were much smaller, for example for the case of the 16 clusters, the speed-ups were between 3.1x to 7.4x for different boards, and were between 6.3x to 16x for the case of 32 clusters. Lavenier concluded that the speed-up of the systolic array was function of the number of clusters and transfer rates between the host and the FPGA board, the latter depends on the capabilities of the available FPGA board [49] and [54].

In 2001, Michael Estlick *et al*., at Northeastern University implemented K-means in hardware using software/hardware co-design [50] and [55]. Their design was partitioned between hardware and host microprocessor where both distance calculation and accumulation were done in hardware in purely fixed-point, while new centroids were computed in the host to avoid consuming large hardware resources associated with the division operation. The design was tested on a platform board called the Wildstar which housed three FPGAs and had memory of 40MB ZBT SRAM. Before the clustering begins, the complete dataset was moved from the host and stored in the ZBT SRAM on board while the initial centroids were stored in registers within the FPGA chip. The point's vectors were then streamed to the FPGA for processing. The hardware implementation achieved a speed-up of 50x over the 500 MHz Pentium III host processor. This implementation benefited from two things: the first was using Manhattan distance metric instead of the commonly used Euclidean metric to reduce the amount of hardware resources needed, and the second was truncating the bitwidth of the input data to be able to process larger number of channels [50] and [55]-[57]. The authors have tested their design on two types of images; the first was to cluster 10 channels Multispectral Thermal Image (MTI) of 614 × 512 pixels which had 120 bits per pixel; and the second was 20 channels Aviris Image of 614 × 512 pixels with also 120 bits per pixel, the latter image benefited from wordlength truncation method which was developed and verified by the same group to ensure acceptable accuracy.

In 2002, Pavel Belanovic [58] who was working with the same group mentioned in [50] and [55] created a library of hardware modules for floating point arithmetic to effectively use floating point arithmetic in hardware instead of using fixed-point format. As an application to use the library created by Belanovic, the authors of [56]-[58] implemented K-means clustering using a hybrid implementation of fixed-point and floating-point arithmetic instead of the purely fixed point implemented previously in [50] and [55]. The design itself was the same as in [50] and [55] except that floating point library modules were used to implement the subtraction, absolute value and addition in the distance calculation part instead of the fixed-point format. The throughput of the distance calculation block in the hybrid design was eight pixels per clock cycle as compared to one pixel per clock cycle achieved in [50] and [55]. Results showed that the hybrid design occupied larger slices within the FPGA than in [50] and [55] in addition to having smaller throughput. As a consequence, no performance improvement was gained from using the proposed hybrid model. However, the authors stated that the implementation could be used for other applications that may require floating point implementation of the distance computation kernel.

In 2003, Bhaskaran [59] implemented a parameterised implementation of the K-means algorithm on FPGA. All of the K-means steps were executed in hardware with exception to the initialisation of cluster centroids that was done in a host. This was the first work to implement the division operation within the hardware to obtain the new centroids using dividers generated by the Xilinx' Core Generator tool. However, the design was tested only on three clusters and achieved a speed-up of 500x over Matlab implementation including the I/O overhead. One disadvantage of this implementation was that the board used did not have any memory capability which restricted the size of image that can be processed at one time to a size that can be accommodated by the FPGA Block RAMs. In addition, the implementation used three divider cores consuming a lot of hardware resources which drove the cost of the implementation high while just needed for a short period of time during the clustering iteration [59].

In 2003, Filho *et al.*, implemented a hybrid implementation of the K-means clustering using software and hardware co-design based on using the Euclidean metric for the distance computation kernel. The hardware implementation achieved speed-up of 2x over GPP implementation eventhough the former was running at 12.5x lower frequency than the FPGA [60].

In 2007, Xiaojun Wang [61]-[62] proposed a variable precision floating point divider for efficient FPGA implementation. The work was an extension of the work presented in [50],

[55] and [60]. As a case study, the author implemented the K-means clustering in FPGA and utilised the created floating point divider to calculate new centroids within the FPGA. This approach required the use of an extra block to convert the fixed-point data to floating point. Then, after the division was done, another floating to fixed-point converter was again needed. Results of clustering an eight-channel multispectral image of 614 × 512 pixels of eight bits each into eight clusters resulted in speed-up of the core computation of 2150x over an equivalent GPP implementation; and an overall speed-up of 11x over GPP when taking I/O communication into consideration. On the other hand, when comparing the speed-up of the hardware implementation using the floating point divider with a hardware implementation doing the division operation in host, the author found that no speed-up or advantage of implementing the division in hardware was gained from using the created floating point divider. The only advantage of such implementation according to Wang was to free the host to work on other tasks while K-means clustering was performed completely in hardware [61]-[62].

Additional work was reported in literature about the hardware implementation of the K-means clustering in other areas such as document clustering [63] and anomaly detection in computing networks [64] reporting encouraging speed-up results over GPP implementations. All previous works on K-means clustering have established a fact that it is not efficient to directly transform the standard software implementation of K-means to hardware and that several modifications were needed to simplify the computation and increase the parallelism [59] and [65].

4.4 Requirements for Hardware Design

Designing software to be used by GPPs is based on using floating point arithmetic, where dedicated floating point units are abundant. It is difficult to map floating point algorithms directly to hardware, one of the reasons is that floating point takes large resources when implemented in hardware. As such, to implement the K-means clustering on FPGA, fixed-point arithmetic was chosen by most previous groups [49]-[51] and [54]-[65]; and will be used in this work too. Fixed-point format requires the user to fix the precision of the computation in order to be able to represent real data points using specific number of bits. In the following subsections, the methodology used to convert the dataset to fixed-point format will be presented. Before converting the data to fixed-point, an additional pre-processing step was needed to filter out un-wanted data vectors; this step is usually applied before

processing Microarray data whether the implementation is carried out in software or hardware implementation.

4.4.1 Pre-Processing of Microarray Data

Microarray technology produces large amount of data which usually contain some noise or irrelevant information. Taking the time to process datasets containing large amount of noise is a waste of time and resources. Therefore, it is important to filter out noise from the dataset before performing clustering to reduce the amount of objects in the dataset to only those of significance. The gene expression profiles of 14 hour Microarray experiment of Yeast Saccharomyces Cerevisiae, commonly known as Baker's Yeast consists of 6400 genes of seven dimensions (6400 × 7) [66]. The rows of the matrix correspond to genes and columns to samples, features or dimensions whereby each gene is read as a vector of seven elements. Such data normally contain noise or irrelevant points (correspond to Microarray spots or samples) which include the following: empty cells or missing profiles indicating no expression profiles were measured for those samples; zero expression profiles; low variance over time, and low entropy profiles. Consequently, it is important to remove genes containing such noise. In this work, Matlab bioinformatics toolbox was used for filtering out noise in the used datasets. The toolbox includes a number of specific filtering functions for removing the genes having missing expression profiles or empty spots, low variance over time, very low absolute expression values, and low entropy values [67]. Performing filtering on the above 6400 × 7 using the toolbox reduced the size of the dataset to 415 × 7 simplifying the process and leading to time efficient clustering as the clustering is now limited to gene’s containing relevant biological information only. As part of automating the pre-processing of the Microarray dataset, a script was written to automate the call of all the filtering functions available in the Matlab bioinformatics toolbox making the task of preparing the dataset for clustering easier and faster.

A last pre-processing step before converting the dataset to fixed-point is to initialise the cluster centroids and supply them as seeds to the hardware core. To be able to validate the hardware clustering in this work, the K-means algorithm was run in Matlab and then the final cluster centroids were selected to be used as the initial cluster centroids for the hardware model. This is because the hardware design does not include circuitry for obtaining initial centroids. The next step is to perform range and error analysis on the given dataset to

determine the best wordlength for representing the samples to be used for the hardware implementation.

4.4.2 Converting Data to Fixed-point Format

Using fixed-point format involves representing a real number in the dataset using specific number of digits for both the integer part and fractional part. Care must be taken when attempting to represent a real number in fixed-point format to avoid conditions of overflow or underflow. Overflow occurs when the number of bits chosen to represent the real number is too small indicating that more bits are needed to represent that real number. On the other hand, underflow occurs when the real number is too small (close to zero) to be represented by the selected number of bits. The way a fixed-point number is described is using the notation shown in equation (4.3).

$$\textit{Wordlength in Fixed-point} = QI.QF, \tag{4.3}$$

where QI refers to the number of bits needed to represent the integer part, and QF refers to the number of bits needed to represent the fractional part. Fixed-point format is suitable only for use in hardware design when the dynamic range of the dataset is low because high dynamic range requires a larger wordlength causing the hardware implementation to be costly. To obtain QI and QF, one need to perform a range and precision analysis to carefully select the minimum wordlength and avoid using more bits than actually needed since cost will be directly affected by the chosen number of bits.

4.4.2.1 Range Analysis

The aim of this step is to obtain the minimum number of bits required to represent the integer part of the signed input using equation (4.4) [68].

$$QI \geq \left\lceil log_2 \left| Range \right| \right\rceil, \tag{4.4}$$

where QI is again the integer part, Range is the difference between the maximum and minimum values in the dataset. As an example to demonstrate this computation, the expression profiles of the filtered Yeast dataset mentioned in subsection 4.4.1 were found to

have a range of [-3.2780, 3.5460], therefore applying equation (4.4) results in a requirement of a minimum of 3 bits to represent the integer part of this dataset.

4.4.2.2 Precision Analysis

Having determined the required number of bits to represent the integer part, the aim is now shifts towards obtaining the number of bits required to represent the fractional part of the dataset. This requires the determination of the precision needed to sufficiently describe the samples in the dataset. The number of bits required to represent the fractional bits can then be obtained using equation (4.5) as stated in [68].

$$QF \geq \left\lceil log_2\left(\frac{1}{Precision}\right)\right\rceil \tag{4.5}$$

Where QF is the number of bits required to represent the fractional part. Back again to the Yeast example, by examining the dataset, three digits were found to be sufficient leading to precision of 10^{-3}. Applying equation (4.5) results in a minimum of 10 bits needed to represent the fractional part of the Yeast dataset. Consequently, the total wordlength (WL) needed for representing the dataset is QI + QF, being 13 bits for the Yeast example. WL can be obtained directly from equation (4.6) as stated in [68].

$$WL \geq \left\lceil log_2\left(\frac{Range}{Precision}\right)\right\rceil. \tag{4.6}$$

The K-means clustering involves computing distances and accumulating them, in addition to accumulating the objects of each cluster. Therefore, the WL resulted in every step of the algorithm will also change requiring care in accounting for such changes. The following example demonstrates the procedure followed to account for changes in WL of distance results and accumulators.

4.4.2.3 Wordlength of Intermediate Results (Distances and Accumulators)

Repeating the above range and precision analysis for every step in the K-means algorithm such as the distance and accumulation parts is very important to avoid having overflow

condition since distance computation and accumulation will result in a wordlength requirement above the 13 bits used for representing the input data. Choosing a high WL to represent these data at random may use more FPGA resources than actually needed driving the cost of the implementation up unnecessarily. Consequently, performing the above range and precision analysis for the distance and accumulation kernels will optimise the implementation and avoid the use of unnecessary extra bits.

To determine the wordlength of the intermediate results in the K-means algorithm the Matlab software of the K-means algorithm was run using the above mentioned Yeast dataset, then, the range and precision analysis were applied to the intermediate results. For this particular case, 15 bits (5 integers, 10 fractional) were found to be required to represent the distances, and 25 (15 integers, 10 fractional) bits for the accumulators to achieve a precision level of three fractional digits. The above process was automated in Matlab to provide an easy tool for performing this analysis on any datasets as part of an off-line pre-processing stage.

4.4.3 Error Analysis Associated with Conversion to Fixed-point

This step consists of two tasks, one is to convert the dataset to fixed-point format to be used in the hardware implementation; and the second task is to convert the Matlab K-means algorithm to fixed-point to be used for comparison with the hardware implementation. Both tasks were performed using Matlab fixed-point toolbox. In summary, the task of converting the K-means algorithm from floating point format to fixed-point using Matlab fixed-point toolbox was automated and combined with the previous range and precision analysis in one program. The automated process is basically a Matlab program which takes the precision, and the Microarray dataset as inputs, then performs the filtering operation, computes the minimum wordlength required to represent the given dataset at the required precisions, compare the error involved in selecting the entered precision as well as different other precisions, and finally converts the given dataset to fixed point. The error analysis allows the user to select the best combination of integer or fractional bits resulting in optimum trade-offs between performance and accuracy.

4.4.3.1 Accuracy Analysis

Converting data to fixed point is always associated with some degradation depending on the range and precision used. It has been established that appropriate trade-off between range

and precision is important to avoid any inaccuracy or unnecessary use of hardware resources. To provide an easy tool to realise these trade-offs, the previously mentioned Matlab program performs accuracy analysis using different precisions revealing the associated fractional number of bits for each case along with the associated relative error between the floating point and the different fixed-point implementations, the relative error is computed as shown in equation (4.7).

$$\text{Relative Error} = \frac{|\text{Floating point data} - \text{Fixed-point data}|}{\text{Floating point data}} \tag{4.7}$$

The program outputs graphical results highlighting the required wordlength and the associated error for different precisions as shown in Fig. 4.2, which helps in deciding on the best WL especially when bit truncation or lower precisions are considered for minimising cost.

In an attempt to demonstrate the effect of the selected precisions on the performance of the K-means clustering, Narayanan et al., implemented the K-means clustering in hardware using different fixed-point representations as an application to his data mining benchmark suite called MineBench which incorporated most data mining algorithms. He demonstrated that different implementations resulted in different speed-ups over pure floating point implementations when tested in Power PC processor. When he used different combinations of integer and fractional bits for the same WL, he found that the performance of the K-means clustering over floating point varied in terms of speed-up and accuracy. The accuracy was higher for the case of larger fractional bits (higher precision), however the speed-up was lower. He concluded that there is a trade-off between accuracy and speed-up, and one must select the best trade-off for a given dataset. For example, when testing the hardware implementation of the K-means clustering for the following WL cases: Q16.16, Q20.12, Q24.8, he reported speed-ups of 9.06x, 8.80x, and 11.59x, respectively, while relative errors for the case of 11 clusters where 1.54%, 1.62%, and 18.71%, respectively. This shows that the worse accuracy was with the case of Q24.8 having eight fractional bits. Lastly, the author reported that 12 bits were needed to represent the fractional part based on the precisions required for the dataset and he concluded that using smaller fractional bits would normally result in significant membership errors in the clustering [69]-[70]. The aforementioned analysis illustrated the importance of carrying out this precision analysis and choosing the appropriate binary representations of the dataset.

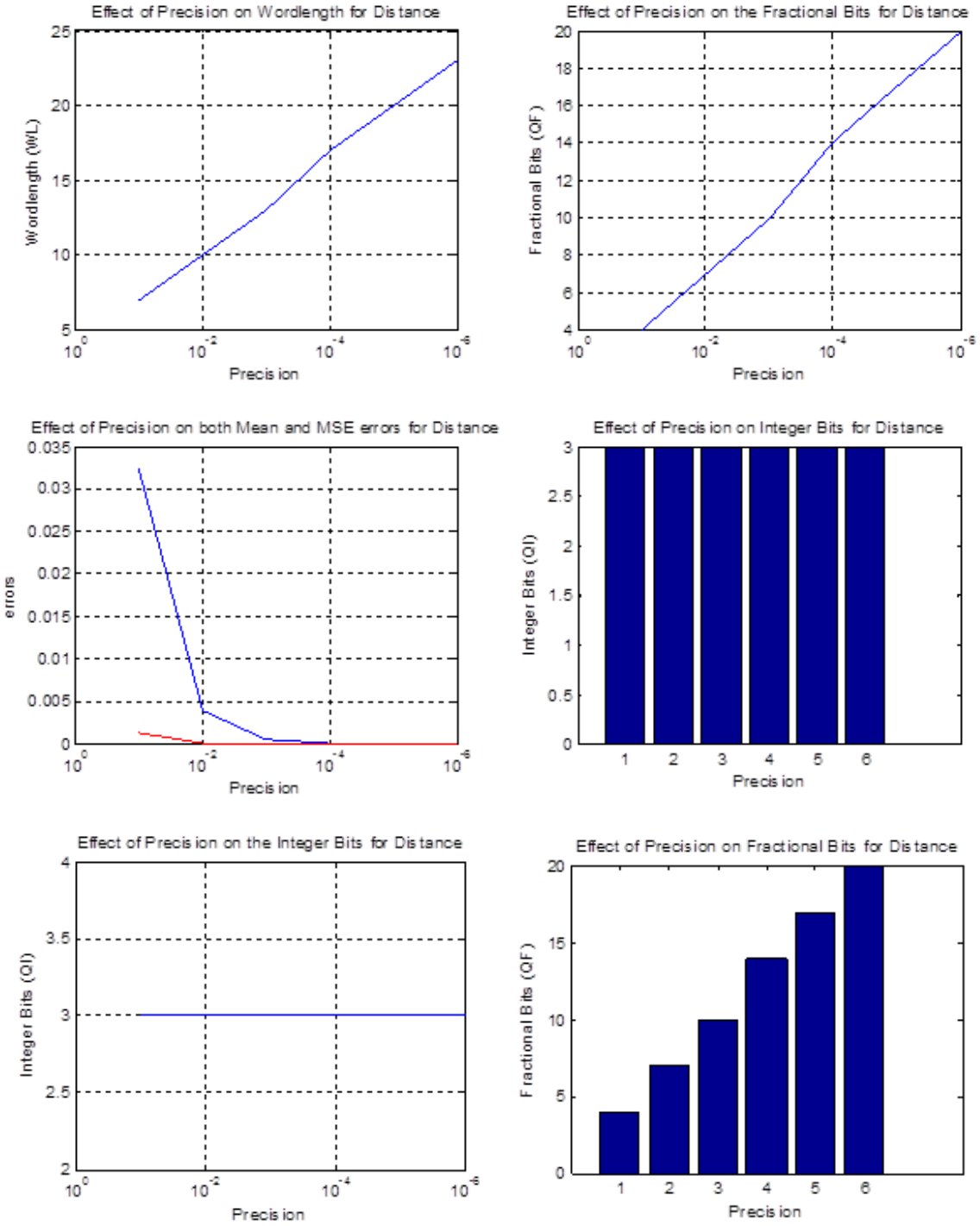


Figure 4.2: *The graphical result from the automated Matlab pre-processing program, which estimates the wordlength requirement for different precisions and the associated relative and mean square errors for each of the considered precisions.*

4.5 Novel Hardware Implementation of the K-means Algorithm on FPGAs

A highly parameterised K-means clustering algorithm is implemented on FPGA; the design is captured in Verilog HDL and parameterised in terms of user parameters. The following subsection illustrates the main blocks of the K-means clustering algorithm implemented in hardware as shown in Fig. 4.3. Subsequent subsections will then cover various novel implementations, two of which will be based on DPR feature.

4.5.1 Hardware Architecture of a Parameterised Single-core K-means Clustering Algorithm

The aim of the presented K-means-core is to carry all the steps of the K-means clustering on FPGA using fixed-point arithmetic eliminating the need for communication with a host. The architecture consists of a number of blocks which execute kernels in the K-means algorithm. The design generates the required hardware resources and logics based on the parameters entered by the user at compile time. These parameters are the wordlength of the input data (B), number of clusters (K), number of point's vectors (N) and number of dimensions (M). The design intends to perform all the K-means kernels within the FPGA including the division operation and avoids directing any task to an off-chip resource although this capability can be exploited whenever it is needed. In the following subsections, an overview about each block will be presented along with the timing for executing each kernel.

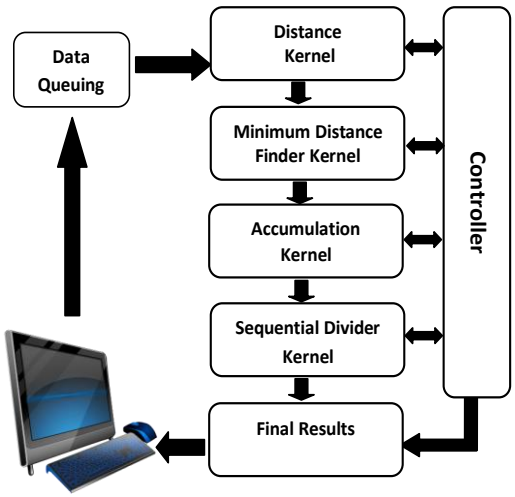


Figure 4.3: *Block diagram illustrating the main kernels of the K-means clustering algorithm.*

4.5.1.1 Distance Computation Kernel Block

Two versions of this block were constructed; one was to compute the Manhattan distances, while the other was to compute the Euclidean distances, both distances were computed between the input vectors and all the vectors of the cluster's centroids. The acceleration in distance computation is arrived at by parallelising the distance computation kernel that is achieved through the instantiation of multiple distance computation units, namely, distance processors (DPs) of a number equal to the number of clusters (K) allowing for parallel computation of distances between the vectors and the K clusters. The architecture of a single Manhattan DP configured to process single dimension consists mainly of B bits subtractor and the logics to obtain the absolute value of the subtraction result, as shown in Fig 4.4.

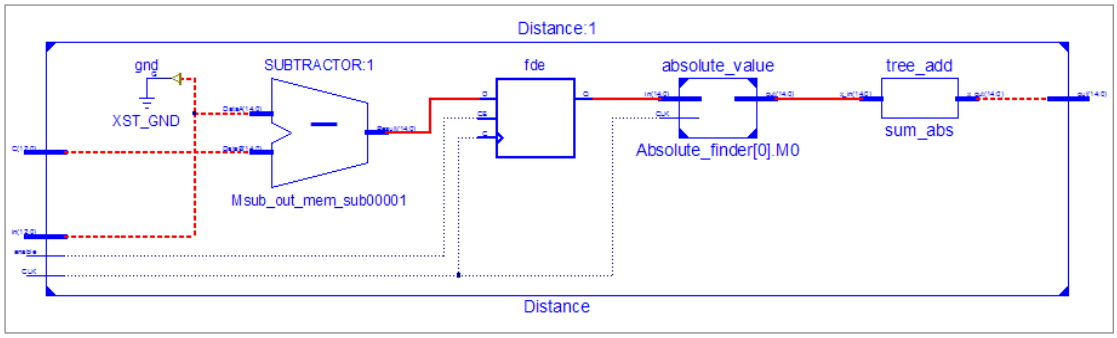


Figure 4.4: *The RTL schematic of a Manhattan DP configured to process single dimension illustrating the main building blocks of a single DP as inferred by the synthesis tool.*

For the case of M-dimensions, each DP would now have M number of subtractors, M absolute-finder units, and one adder tree having a number of levels equal to:

$$Ceil\lfloor log_2(M)\rfloor,$$

which finds the summation of the resulted M absolute values corresponding to the summation:

$$D(P_i, C_i) = \sum_{i}^{M} (|P_i - C_i|).$$

Since there are K clusters involved in the clustering operation, K number of DPs will be inferred each having M subtractors, M absolute-finders and one adder tree. Consequently, the K number of DPs will have a total of (K × M) subtractors, K × M absolute-finders and K adder trees.

The block receives streaming inputs stored in on-chip Block RAMs or from off-chip memory along with the initialised or updated cluster's centroids, and computes the distances between each received point and all cluster's centroids simultaneously. The hardware resources inferred by the synthesis tool to generate the distance processors (DPs) are based on the number of clusters, wordlength of the data (B), and the number of dimensions (M). Each DP is responsible for the computation of the distances between all the dimensions of the input and all the dimensions of one of the cluster centroids. Therefore, one DP is associated with each cluster as can be seen in Fig. 4.5; having K clusters would require the utilisation of K number of DPs by the hardware design. These K DPs work simultaneously such that the distances between every point to all clusters are computed in one clock cycle, hence fully exploiting all possible parallelism associated with the distance calculation kernel.

The datapath of this block depends on the number of dimensions (M) in the dataset such that the number of stages is given by equation (4.8).

$$Datapath\ of\ Distance\ Kernel = Ceil\ [\log_2 (M). \tag{4.8}$$

As for the computation time of a single pass through the whole dataset, it was found to be a function of the number of points (N) which is basically the depth of the memory holding the dataset as shown in equation (4.9).

$$Distance\ Computation\ Time = 2\ CLK\ cycles + N \tag{4.9}$$

Consequently, the block has a throughput of K distances per clock cycle and a latency of 2 clock cycles only.

As for the Euclidean distance version, K number of DPs are utilised but each having different resources from those used in the Manahan distance version. The main resources

include (M × K) number of subtractors, and (M × K) DSP blocks to perform the multiplication needed for the computation of the square.

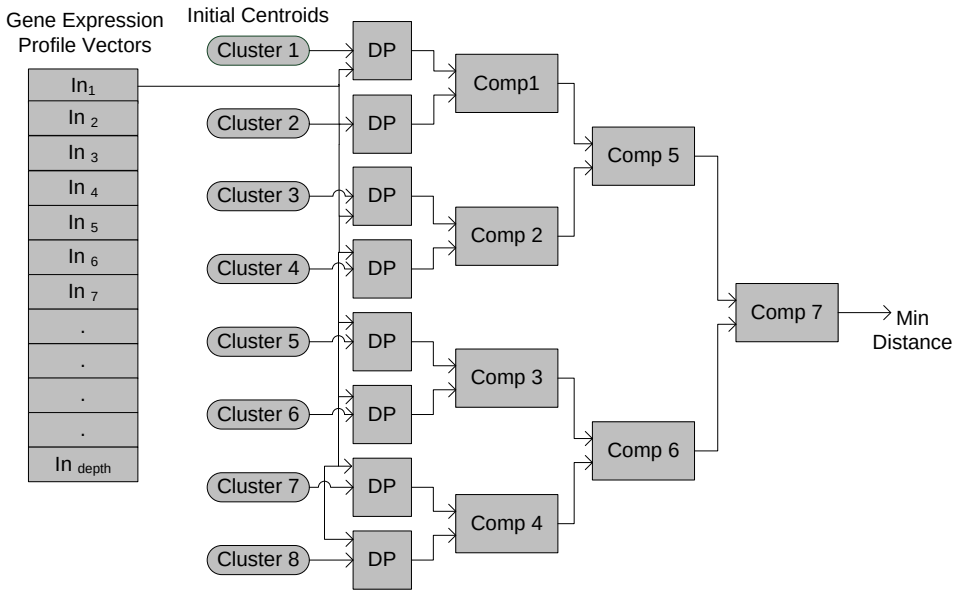


Figure 4.5*: The datapath of the distance kernel block and the minimum distance finder block, highlighting a comparator tree having a number of levels specific to the case of eight clusters.*

4.5.1.2 Minimum Distance Finder kernel Block

This block has the role of comparing the K distances received from the previous block to determine the minimum distance and the associated index which correspond to the ID of the cluster closest to the point vector being processed. The block consists mainly of a comparator tree as shown in Fig. 4.5, which has number of stages related to the number of clusters (K) given by equation (4.10). Fig. 4.6 illustrates the logic circuit of a single comparator.

$$Comparator\ Tree\ Stages = [Ceil[\log_2(K)]] \tag{4.10}$$

Each comparison unit within the tree utilises 17 CLB slices, corresponding to the logic inferred by the synthesis tool based on the behavioural description of the unit captured in the Verilog. The number of clusters alone affects the number of comparators inferred by the design. In addition, the datapath of this block is the same as the number of stages needed to

complete the comparison shown in equation (4.10). For instance, when eight clusters were used, three comparison levels were needed consisting of a total of 7 comparators to obtain the minimum distance and the index of the cluster associated it. The combined execution time of this block and the previous block could be now summarised in equation (4.11).

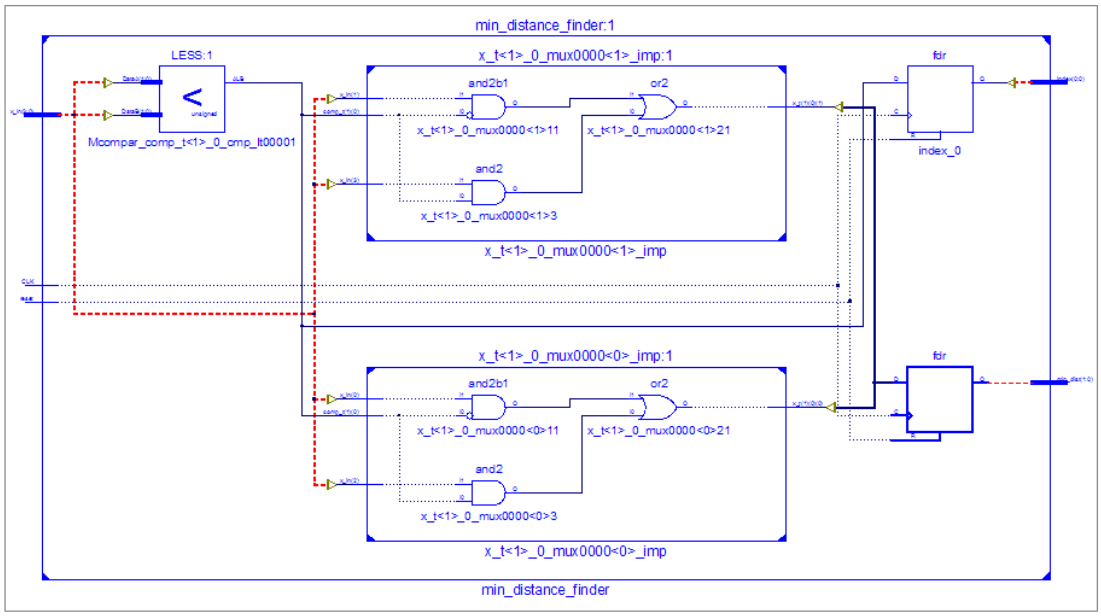


Figure 4.6*: The RTL schematic of a single comparator unit with two inputs of 2 bits each and single dimension as inferred by the synthesis tool.*

$$\textit{Execution Time} = (\textit{Latency of 2 clock cycles} + \textit{Ceil}[\log_2(M)] + \textit{Ceil}[\log_2(K)]) \qquad (4.11)$$

4.5.1.3 Accumulation kernel Block

This block consists of K number of accumulators and K number of counters in addition to some other logic; this is for the case of single dimensional dataset. Once the first minimum distance has been computed for the first gene, the block gets signalled to start receiving two inputs every clock cycle, these inputs are the index of the cluster closest to the point vector and the point vector itself. Then, the block performs two operations: the first is checking the received index (cluster assignment), to direct the received point vector to the accumulator having the ID that matches the received index. The second operation is to increment the counter associated with the same cluster. As such, the block is actually keeping track of the number of points in each cluster along with the values of these points.

For the case of multi-dimensional datasets (M>1), the required number of accumulators will increase to M accumulators for each cluster while the number of counters remains the same (K counters). Consequently, the total number of accumulators in this block is summed up in equation (4.12).

$$Accumulator\ numbers = K \times M, \tag{4.12}$$

where each of those M accumulators is responsible for accumulating the values of one of the M dimensions of the received point vector.

As a result of utilising fixed-point arithmetic in this design, extra care was taken in choosing the wordlength (WL) for each accumulator and counter to minimise hardware resources and avoid overflow as was detailed in subsection 4.4.2.3. Based on the WL of the input selected initially to represent the data and the number of points in the dataset to be processed, the accumulator and counter WLs were calculated as shown in equations (4.13) and (4.14), respectively.

$$Accumulator\ WL = [\log_2[(the\ maximum\ range\ that\ can\ be\ represented\ by\ the\ input\ WL) \times N]] \tag{4.13}$$

$$Counter\ WL = \log_2[N] \tag{4.14}$$

As an example based on 13 bits per sample to represent single dimensional dataset of 25,000 data points, (usually N <= 25,000 point vectors for Human Microarrays), 15 bits were found to be sufficient for each counter and 32 bits for each accumulator.

Once all the point vectors in the dataset have been processed and assigned to one of the clusters, the accumulations-counting operations stop and the block outputs a finish signal to the subsequent block to start the computation of the new means (or new centroids).

4.5.1.4 Sequential Divider Kernel Block

The divider kernel block is responsible for receiving results from the accumulation kernel block and calculating the new cluster centroids corresponding to the means of the accumulators computed using a divider generated by Xilinx' Core Generator tool. Similar to the previous block, care was needed in specifying the WLs of the divider inputs and outputs.

In the case of Microarray data for instance, 32 bits were determined to be sufficient to represent the dividend (correspond to WL of an accumulator), and 15 bits for the divisor (correspond to WL of a counter). These values were chosen based on the WLs of the previous block because the role of the divider is to actually perform the division operation on each of the accumulator results over the corresponding counter results to obtain the new cluster's centroids.

Once signalled to start, this block starts scheduling the data received (total of (M × K) accumulator's results and K counter's results) to be serviced by the divider core serially as illustrated in Fig. 4.7. The scheduling circuitry consists of a multiplexer and two arrays of shift registers which work together in receiving the data from the previous block and sending them sequentially to the divider core. The total number of sequential division operations performed by the divider core is:

$$M \times K$$

operations. The division results are then used as the new centroids for the subsequent clustering iteration. The number of clock cycles taken by the divider to complete its work is a function of the divider latency and the number of clusters as well as the number of dimensions as shown in (4.15).

$$\textit{Divider time} = (\textit{core latency}) + (K \times M) \tag{4.15}$$

The core latency changes if the divider core is changed to reflect different WLs for the dividend or divisor. In the case of Microarray datasets examined during the pre-processing stage which resulted in 32 bits for the dividend and 15 bits for the divisor, the divider core has a latency of 84 clock cycles.

The decision of using a pipelined divider generated from core generator as opposed to a serial HDL divider was arrived at after comparing the performance of both dividers in terms of area and timing. The serial divider was found to process one bit of information at a time, thus for a 32 bit dividend and 8 clusters, the number of clock cycles that were needed was 256 clock cycles as compared to 92 for the pipelined divider. These results were based on single dimensional data. However, this timing difference amplifies when more dimensions are used, for instance using ten dimensions and eight clusters causes the serial divider to take 2560 clock cycles while the pipelined divider to take 164 clock cycles only. On the other

hand, the number of CLB slices consumed by the serial divider is a lot less than the pipelined divider, where the latter consumes 1389 slices as compared to 91 slices for the former. Since one of the aims of this implementation is to accelerate the K-means clustering, the pipelined divider was favoured over the serial one. However, the design could be adopted to work with the serial divider if area was found to be a bottleneck at specific circumstances.

The combined blocks detailed in this subsection constitute a complete single K-means core that will be used as the building block of the subsequent novel implementations including a multi-core K-means clustering implementation and three DPR implementations. The proposed three DPR implementations are: a reconfigurable distance kernel of the K-means core; a reconfigurable complete K-means core; and a reconfigurable multi-core K-means implementation. In total, five implementations of the K-means clustering are proposed in this chapter, three of which are based on DPR technology.

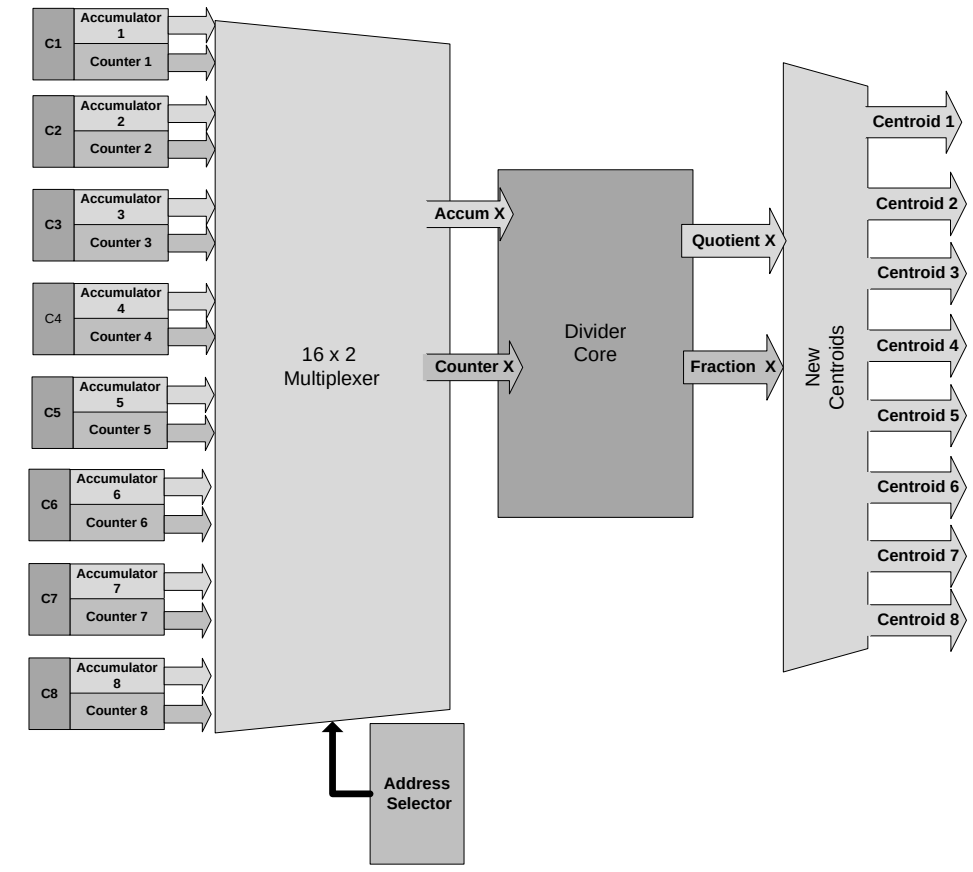


Figure 4.7: *The architecture of a sequential divider kernel block for the case of eight clusters and single dimension; the block receives 16 inputs corresponding to the results of eight accumulators and eight counters; each pair is associated with one cluster. Then, the block schedules one pair at a time to be serviced by the divider core and outputs the new centroids for each cluster sequentially.*

4.5.2 Multi-core Implementation of the K-means Clustering Algorithm

Current state of the art FPGAs have resources that permit multiple tasks to be fitted onto the same FPGA. This implementation aims at harnessing the abundant resources available in state-of-the-art FPGAs to increase the acceleration of the K-means clustering through the implementation of multi-core architecture. Additionally such architecture aims at targeting special applications such as server solutions. A multi-core implementation of the K-means clustering is implemented based on the single K-means core detailed in subsection 4.5.1. The new implementation is based on replicating the K-means core consisting of all the processing blocks including the divider a number of times (as permitted by the available resources) allowing the cores to execute the clustering in parallel. Each core executes the K-means clustering independently as illustrated in Fig. 4.8 for the case of five-core implementation. Final results from each core are then sent to the host. The number of cores that can be included in any multi-core implementation is based on the user specific requirement and the available resources of the given FPGA.

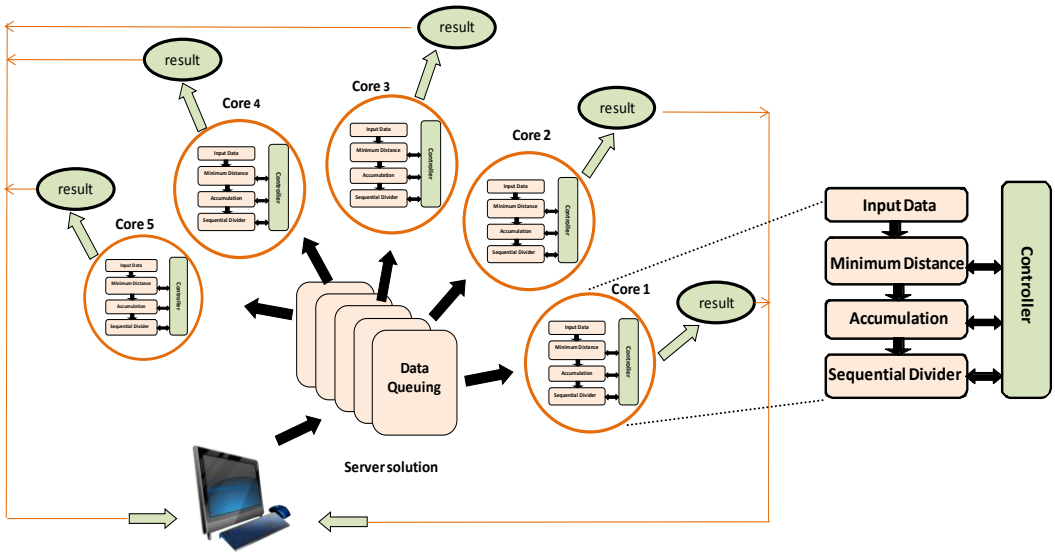


Figure 4.8: *An illustration of the five-core implementation of the K-means clustering.*

The advantages of such implementation are: first, the capability of processing different datasets simultaneously by allowing different users to send data to the same FPGA for clustering forming a server solution. Second, the same dataset could be clustered by multiple cores and results are combined using ensemble method to enhance the clustering accuracy.

Finally, the same dataset could be clustered by different cores using different parameters such as the number of clusters, different distance metrics, or different initial centroids allowing users to quickly compare clustering results performed using different parameters and asses the quality of each. In addition, the latter implementation could be sued as ensemble by combining the results of the different cores.

4.5.3 DPR Implementation of the Single-core K-means Algorithm

The DPR capability of modern state-of-the-art FPGAs allows for partially reconfiguring specific parts of the FPGA during run time without affecting the configuration of the other parts. Consequently, some tasks remain uninterrupted while others are being modified elsewhere in the same device. This technology opens the horizon for a wider spectrum of applications of the K-means clustering serving different purposes. In the following subsections, three different implementations of the K-means clustering algorithm based on DPR are presented. The first DPR implementation is based on reconfigurable distance kernel while the second is based on a reconfigurable single-core of the K-means clustering. The last DPR implementation is based on reconfigurable eight-core implementation.

4.5.3.1 Reconfigurable Distance Kernel

This implementation is based on identifying a specific kernel within the K-means clustering that is candidate to benefit from DPR. DPR is used to modify specific kernel of the K-means clustering, namely, the distance computation kernel allowing for the partial reconfiguration (PR) of this kernel to reflect new distance metric. In this implementation, the Manhattan and Euclidean distances are chosen, however additional distance metrics could be included in the future if needed. The original modular K-means core is slightly modified to allow the replacement of the distance computation kernel with another block based on the Euclidean distance rather than the Manhattan distance. As such, a new Verilog code was written to implement the distance computation kernel in Euclidean metric making use of the abundant DSP48 blocks available in Virtex-4 devices. The availability of such hardened IP blocks on modern FPGAs simplified the multiplication operation associated with using the Euclidean distance, something that used to be more difficult and costly in previous generations of FPGAs. Using DPR to reconfigure the distance kernel provides a solution to a user who wants to alter the distance metric kernel only without changing other blocks in the design or interrupting other running processes on the device.

The DPR implementation of the K-means clustering based on setting the distance kernel as reconfigurable partition (RP) is created using Xilinx' hierarchical design methodology and PR design flow that were explained in chapter 3 [38]-[39]. The RP region is configured with one of the two possible reconfigurable modules (RMs) which are variations of the RP region corresponding to the logic resources required to implement the distance kernel with either the Manhattan or the Euclidian distances as illustrated in Fig. 4.9. More RMs could be created in the future for other distance metrics such as the Hamming, Cosine, Canberra, Pearson or Rank correlation coefficients if required by a specific application.

This design is suitable for implementation in most modern FPGAs, however the location of the RP region may be different depending on the FPGA device used as different FPGAs have different number of DSP blocks and have different spatial arrangement. Consequently, additional care is needed to ensure that enough DSP blocks are included within the RP region in order for the implementation to be successful.

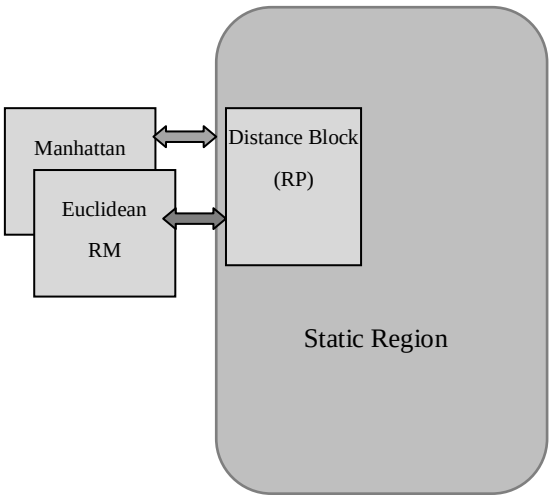


Figure 4.9: *Illustrative diagram of the DPR implementation of the single K-means core based on setting the distance kernel as RP, and providing two RM corresponding to the Euclidean and Manhattan distance kernels.*

A limitation of this implementation is that it is targeting eight clusters and single dimension. Using more clusters or dimensions is also possible, but it will contribute to a significant increase in the amount of resources. However, such increase may be possible to cater for in some circumstances if the FPGA resources were sufficient. Furthermore, additional RMs for each of the distance metrics could have been added to reflect variations

in the number of clusters or dimensions. This capability could be useful to alter the implementation to work with different dimensions or number of clusters. On the other hand, this latter implementation is very complicated as care must be taken to meet the design consideration of having the same exact number of I/Os in all RMs, which means that some RMs must be fitted with additional I/Os not really needed by the design to act as dummy I/Os.

4.5.3.2 Reconfigurable Single-core

The second DPR implementation of the K-means clustering is based on reconfiguring the complete K-means core following the DPR design methodology used in subsection 4.5.3 which was also outlined in chapter 3. However, a new design Wrapper is created to instantiate the complete K-means core to be able to set the whole K-means core as RP.

The advantages of this approach is the capability of swapping an existing K-means core with any other cores (RMs) having different initialisations such as different cluster centroids, different Microarray dataset, different distance kernels e.g., Manhattan or Euclidean, different parameters e.g., B, K, M or N. Consequently, the memory contents of the K-means core or any of its parameters are updated during run-time without disrupting other processes running on the device. As a result, the RMs created to modify the K-means core are variations of the original configuration that are created to reflect desired functionalities.

4.5.4 DPR Implementation of Multi-core K-means Clustering

This DPR implementation is based on reconfigurable multi-core implementation of the K-means clustering, whereby multiple K-means cores placed onto the same FPGA are all made reconfigurable following the same methodology used in the previous two implementations.

The purpose of making those cores reconfigurable is to allow cores to be activated on demand by different users in a network allowing the FPGA to be partially reconfigured at run-time when a user initiates a request for the configuration/reconfiguration of the specific core owned by the user. The advantages of this implementation could be summarised as follows:

1) *Server solution, cores get activated on demand without interrupting any running applications residing on the same chip or altering their configurations.*

2) *Fault tolerance, in case of an error calling for reconfiguration, the affected core gets partially reconfigured without disrupting other cores or other applications.*

3) *Fault tolerance, if the FPGA fabric of an existing core becomes faulty for any reason, the user can dynamically configure another region of the device to run the same core without interrupting other applications on the chip, basically this is relocating a K-means core from a defected RP region to another healthy location within the same device.*

4) *Re-locatable, in addition to use it for fault tolerance, this feature allows cores to be re-located around the FPGA to serve different purposes such as when tasks need to be compacted to make space available for newly introduced tasks onto the FPGA.*

5) *Conduct special studies, such as the implementation of K-means ensemble clustering.*

The number of complete K-means cores that can be placed in a single device is a function of the available resources. For instance, using *XC4VFX12* allows for one core only to be placed in the FPGA, using *XC4VLX60* allows for placing 8 cores, using *XC4VLX80* allows for placing 11 cores, and using *XC4VLX100* allows for placing 16 cores. For demonstrating a proof of concept, the eight-core approach is implemented in this chapter.

To proof the above concept, a DPR implementation based on using eight K-means cores is created using Xilinx' *XC4VLX60*. In this implementation, the single-core K-means architecture presented in subsection 4.5.3.2 is modified to remove the divider block for simplifying the implementation. The rationale behind the removal of the divider is that it is not used all the time during the clustering, and it occupies the largest amount of resources among other kernels of the K-means clustering. Consequently, using eight dividers would waste a lot of the CLB slices. Other options to compensate for the removal of the divider is to use the small serial divider presented in subsection 4.5.1.4, use one divider shared among the eight cores, or utilise Power PC or MicroBlaze for the division operation. However, in the meantime those options are not implemented in this research and are considered in future work if this implementation is proved to be useful and feasible. The new K-means architecture is referred to as the K-means core in this subsection.

Based on the newly modified K-means core which excludes the divider, a new design Wrapper compatible with Xilinx' hierarchical design methodology and PR flow is created which instantiates eight of the newly modified K-means core. The general steps for implementing the eight-core DPR design are summarised as follows:

1) *Create a new Wrapper which instantiate eight K-means cores.*
2) *Set each of the eight cores as RP.*
3) *Create multiple RMs for each RP. (RMs reflect K-means cores with different datasets, cluster's centroids or distance metrics).*
4) *Create several configurations based on different RMs for each RP.*
5) *Run the implementations.*
6) *Create the bitstreams for the different implementations.*
7) *Verify the implementations.*

The three aforementioned DPR implementations presented in this chapter were based on storing the configuration/reconfiguration bitstream files in a host and downloading them via JTAG cable to the FPGA. The following section will present the implementation results of the five implementations of the K-means clustering.

4.6 Implementation Results

4.6.1 Single-core Implementation

At first, the design was captured in Verilog hardware description language (HDL) and configured with the following parameters: (B=13, K=8, M=1, N= 2905). Second, it was functionally simulated using Mentor Graphics ModelSim SE 6.0 software. When the design was simulated using the pre-processed Yeast dataset mentioned in subsection 4.4, simulation results showed that 2971 clock cycles were needed for one complete iteration to cluster the 2905 points (415 × 7 gene vectors) with dataset already initialised in the Block RAMs. The algorithm converged after 25 iterations taking a total of 74275 clock cycles. This result does not take into consideration the time needed to write data to the FPGA Block RAMs since in this implementation the Block RAMs were initialised from file. However, if data were to be

written to the Block RAMs, 2905 clock cycles are additionally required. The final results corresponding to the cluster indices for each point vector were written to the FPGA's Block RAMs. The complete execution time of the hardware implementation is computed as in equation (4.16):

$$\frac{Hardware}{Execution\ Time} = \frac{\#clock\ \ cycles\ \ per\ iteration \times \#\ iteration}{Clock\ Frequency}, \tag{4.16}$$

where "clock cycles per iteration" refers to the number of clock cycles to complete one iteration, and the clock frequency is the frequency obtained from the hardware implementation results as shown in Table 4.1. Since simulation results showed that it took about 2971 clock cycles to complete single iteration and 25 iterations to converge, the hardware execution time was just 742.75 µs as computed from (4.16), given that the clock frequency was 100 MHz. This result was verified by comparing the actual number of clock cycles recorded by Xilinx ChipScope Pro.

Having simulated the design successfully, the design was next synthesised, translated, mapped, placed and routed using Xilinx ISE 12.2 to target Xilinx' *XC4VFX12* FPGA. Finally, various bitstream files were generated and downloaded to Xilinx ML 403 platform board which houses the *XC4VFX12* via a JTAG cable. The generated bitstreams were for the complete K-means core and for all of its individual blocks which were used for testing the functionality of each block at a time using Xilinx' ChipScope Pro 12.2. Table 4.1 summarises the place and route results of the single-core FPGA implementation of the K-means clustering based on configuring the core with the aforementioned parameters.

The results were compared against the fixed-point Matlab model using the Yeast Microarray datasets and were found to be consistent. Another implementation of the single core was implemented based on a larger Xilinx' FPGA, namely, the *XC4VLX25*. The new implementation achieved a frequency of 126 MHz, consequently the execution time of the new implementation was found to be 589 µs according to equation (4.16). The reason for implementing the design on a larger FPGA is to allocate larger area footprint for the multi-core K-means implementations requiring more resources than the *XC4VFX12*, whose results will be presented in the next subsection.

Table 4.1: Place and Route Synthesis Results for a Single-core K-means Implementation

Device	Xilinx' *XC4VFX12*	
Parameters	(B=13 bits, K=8, M=1, N =2905)	
	Used/Available	**Utilisation Ratio (%)**
CLB Slices	2,866/5,472	52
Slice Registers	4,314/10,944	39
4 input LUTs	2,670/10,944	24
Block Rams	5/36	13
Clock Frequency	100 MHz	

Furthermore, the distribution of CLB slices across the K-means blocks has been investigated and reported in Table 4.2 indicating that the divider kernel utilised the largest amount of resources among other blocks accounting for approximately half the size of the device floor area.

Table 4.2: Distribution of CLB Slices across all the K-means Blocks

Device	Xilinx' *XC4VFX12*		
Parameters	(B=13 bits, K=8, M=1, N =2905)		
K-means Kernel's Blocks	**Used Slices**	**Utilisation (%)**	**Clock Frequency (MHz)**
Distance +Minimum Distance Finder	429	7	110.6
Accumulation and Counting	492	8	348.8
Divider	1,438	26	160.5
Controller	369	6	186.6
Input RAM	4	11	459.9
Results RAM	1	1	-
Total	2,866	52	100 MHz

4.6.1.1 Comparison with GPP Implementation

The comparison of the aforementioned FPGA implementation with an equivalent implementation running on GPP has been carried out in this subsection to study the performance of FPGAs over GPPs. Matlab's Statistical toolbox (Matlab 2008a), which has an optimised K-means function was used to model the GPP implementation of the K-means algorithm based on using Manhattan distance. The GPP implementation was tested on 3.0 GHz Intel Core2 Duo E8400 GPP running on Windows XP Professional operating system, with 3 GB RAM. The average execution time of 10,000 runs of the GPP implementation of the K-means core was 0.0241 ± 4.49e-4 s (24.1 ms), with minimum execution time of 23.4

ms and maximum execution time of 31.3 ms. These results were based on initial centroids being pre-defined beforehand and given as seeds to the K-means core (the core includes the divider). Consequently, the speed-up achieved when implementing the K-means clustering algorithm on FPGA as compared to Matlab is computed as in equation (4.17).

$$Speed\ up = \frac{Matlab\ \ Execution\ \ Time}{FPGA\ \ Execution\ \ Time} \tag{4.17}$$

The speed-up of the FPGA implementation was ~41x over the GPP implementation for the case when using *XC4VLX25* (FPGA execution time was 589 µs); and 32x when using *XC4VFX12* (FPGA execution time was 742.75 µs). The difference in the execution times between the two devices is as a result of the two achieving different maximum clock speed using the same parameters, where the *XC4VLX25* achieved 126 MHz while the *XC4VFX12* achieved 100 MHz. In summary, the FPGA implementation of the K-means clustering outperforms GPP implementation in terms of execution time. The maximum clock speed of the FPGA implementation was the determinant factor of the speed-up and was influenced by the selected device.

4.6.1.2 Comparison with other FPGA Implementations

In general, it is difficult to compare like for like FPGA implementations because of the use of different FPGA families and chips, as well as different design parameters. Nonetheless, a comparison has been attempted here between the FPGA implementation of the single-core K-means implementation presented in this chapter with the closest implementation reported in literature, namely, the one reported in [56]. The parameterised K-means core presented in this chapter is modified and excluded from the divider to make it compatible with the FPGA implementation reported in [56], where the latter was based on performing the division operation on a host. Both implementations were based on data size of 1024 × 1024, 10 dimensions, 12 bits per dimension or feature, and 8 clusters. In both cases, data were stored off chip and streamed on to the FPGA. Comparative results are shown in Table 4.3. Obviously, the implementation presented in this thesis is faster because it is based on a more recent FPGA technology, but it is also more compact using normalised slice/LUT count.

More importantly, it is more flexible as it has a higher degree of parameterisation compared to the implementation reported in [56].

Table 4.3: Comparison with another FPGA implementation of the K-means Clustering

Parameters	(B=12, M=10, K=8, N= (1024 × 1024))	
Device	**Xilinx' XCV1000 [56]**	**Xilinx' XC4VFX12**
Slices	8,884 (out of 12288)	5,107 (out of 5549)
LUTs	17,768	10,216
Max Clock Frequency	63.07 MHz	100 MHz
Single loop processing time	0.17 s	~ 0.07 s

4.6.2 Multi-core Implementation

The Xilinx' *XC4VLX25* FPGA was used in the five-core implementation of the K-means clustering. Synthetic Microarray dataset was used having a depth of 14,525 genes and a single dimension; those genes were partitioned among the five cores such that each core was responsible for clustering 2095 genes. All the cores were configured with the same parameters (B=13, K=8, M=1, N=2905). Consequently, the speed-up of the multi-core implementation over the single-core can estimated as in equation (4.18).

$$\textit{Multi-core speed-up} = \textit{Single-core speed-up} \times \textit{number of cores} \tag{4.18}$$

Since the single-core implementation achieved ~41x speed-up over GPP using *XC4VLX25*, the estimated speed-up for the five-core implementation was ~205x. As for the hardware resources inferred by this implementation, they were about five times those of the single-core implementation as illustrated in Table 4.4. The reported results of the single-core implementation shown in Table 4.4 varies from those reported in Table 4.1 because the two implementations were based on different dividers, where the former was a 32 bit divider while the latter was a 25 bit divider.

Table 4.4: Place and Route Synthesis Results for the five-core K-means Implementation as compared with a single-core implementation

Device	Xilinx' *XC4VLX25*			
Parameters	(B=13 bits, K=8, M=1, N =2905)			
	Used/Available Single-core	**Utilisation Ratio (%)**	**Used/Available Five-core**	**Utilisation Ratio (%)**
CLB Slices	2,208/10,752	20	10,750/10,752	99
Slice Registers	3,022/21,504	14	15,120/21,504	70
4 input LUTs	2,948/21,504	13	14,705/21,504	68
Block Rams	5/72	7	25/72	32
Clock Frequency	126 MHz		124 MHz	

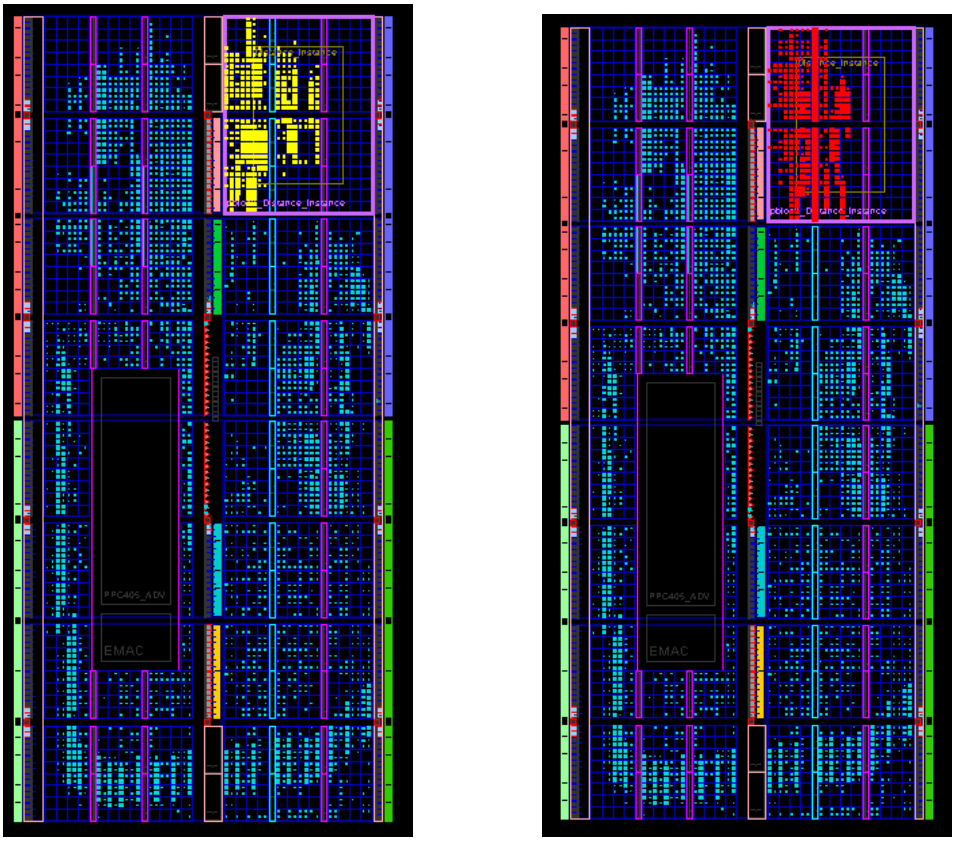
4.6.3 DPR Implementation of Partial/Single-core

4.6.3.1 Reconfigurable Distance Kernel

This DPR implementations was created using Xilinx PlanAhead 12.2 tool where the distance block was set as an RP as explained in subsection 4.5.3.1. The core was configured using the same parameters used in the aforementioned implementation (B=13, K=8, M=1, N=2905). Fig. 4.10 illustrates the floorplan of the two implementations highlighting the amount of resources within the RP regions for each distance metric.

One of the reasons for selecting the Manhattan and Euclidean distances beside their popularity in clustering Microarray data, is that the logic resources of the two metrics were found to be comparable in terms of the number of CLB slices and LUTs, with an exception that the Euclidean metric requiring DSP48 blocks to implement the multiplication operation. However, this additional requirement was possible to cater for in this implementation and did not impose shortage in resource as most modern FPGAs nowadays come with heterogeneous hardware resources that usually include DSP blocks. The close area footprint and similar requirement of resources between the two distance metrics led this application to be candidate for DPR implementation, otherwise it would not have been possible to implement this unique architecture. Furthermore, for the case when DSPs are not abundantly available or not available at all, the multiplication operation could still be performed using more CLB slices and LUTs, but this approach will significantly increase the size of the RP region more than the case of using dedicated DSP block. Based on using eight clusters, 13 bits wordlength, depth of 2905, and single dimension, the CLB slices for the Manhattan

distance block were found to be 277 as compared to 246 for the Euclidean distance both occupying only 4% of the total Xilinx' *XC4VFX12* floor area, with the Euclidean distance additionally requiring eight DSP48 blocks. The implementation of the Euclidean distance presented in this subsection is based on using DSP48 blocks rather than building multipliers from LUTs.



(a) Manhattan (b) Euclidean

Figure 4.10:*Floorplan image resulted from the DPR implementation of the K-means clustering based on using two distance metrics, with the rectangular area corresponding to the RP region enclosing the resources of the distance kernel block for the cases of: (a) Manhattan distance, (b) Euclidean distance.*

Two main configurations were generated based on the two RMs (Euclidean and Manhattan), the associated full and partial bitstreams for each configuration were also generated. The size of the full bitstream was found to be 582 KB while the partial bitstream was 61 KB for both configurations. In this work, JTAG cable was the configuration mode having a bandwidth (BW) of 66 Mbps. Consequently, when configuring the FPGA with a full bitstream for any of the two cases whether the Manhattan or the Euclidean distance, the

configuration time was found to be 70.55 ms as computed from equation (4.19). On the other hand, when using the partial bitstream to partially configure the FPGA with any of the two distance metrics, the reconfiguration time was found to be 7.39 ms. This implementation concludes with the fact that DPR offers significant time saving when the distance metric need to be changed leading to ~10x speed-up in reconfiguration time over full device reconfiguration.

$$ConfigurationTime = \frac{Size\,of\ bitstream}{BW\ of\ Configuration\,Mode} \tag{4.19}$$

Furthermore, using ICAP as a reconfiguration mode will reduce the reconfiguration time significantly over JTAG as the former has a bandwidth of 3.2 Gbps compared to 66 Mbps for the latter. However, the speed-up ratios remain the same for the two configuration modes.

According to Fig 4.10, the DPR implementation was found to have a disadvantage associated with wasting some of the CLB slices within the RP region due to having to enclose enough DSP48 blocks needed for the Euclidean metric. The image shows that ~72% of the RP's CLB slices are unused for in the Manhattan distance case and ~69.5% for the case of the Euclidean distance. Such issue is expected to amplify as a result of increasing the number of clusters or the dimensionality of the dataset as both will contribute to increasing the requirement of DSP48 blocks. Consequently, the DPR implementation must be justified for the application in hand and its benefits must outweigh non-DPR implementation.

4.6.3.2 Reconfigurable Single-core

This DPR implementations was created using Xilinx PlanAhead 12.2 tool where the complete K-means core (including the divider) was set as RP as explained in subsection 4.5.3.2. The implementation targeted Xilinx' *XC4VFX12*. The place and route results of the implementation showed that the K-means core occupied 1,178 CLB slices (~22%) of the FPGA floor area, as seen in Fig. 4.11. The partial bitstream was found to be 121 KB while the full bitstream was 582 KB. As such, the partial reconfiguration time was 14.67 ms as compared to 70.55 ms for the full reconfiguration time which lead to the DPR implementation being ~5x quicker in reconfiguration time than non-DPR implementation. This result implies that in addition to being able to partially reconfigure the K-means core without interrupting the operation of other cores, the partial reconfiguration is also faster than the full reconfiguration.

A successful attempt was made by a colleague in the SLIg group at Edinburgh University to reconfigure the K-means core (without the divider) presented in this thesis internally as part of an Internal Reconfiguration Engine (IRE) system developed by the same colleague. The IRE system was based on using a PicoBlaze processor as custom ICAP controller to read the partial bitstreams stored in SRAM located off-chip to reconfigure the FPGA via ICAP. The custom ICAP controller achieves higher performance than Xilinx' HWICAP IP core, the latter is provided by Xilinx controller for its ICAP. The reconfiguration time of a single K-means-core was actually measured on the ML 403 board while using the IRE and found to be 360 µs as compared with 350 µs estimated using equation (4.19). This indicates that actual reconfiguration using reliable reconfiguration system such as IRE is associated with negligible overheads. Therefore, estimating the reconfiguration time using equation (4.19) adopted in this thesis yields valid results.

Additionally, the speed-up in reconfiguration time is a function of the FPGA floor area which means that the proposed DPR implementation leads to higher speed-up in reconfiguration time compared to non-DPR solution. Indeed the above DPR implementation of the single K-means-core was implemented on a large FPGA namely, the *XC4VLX60* which has a full bitstream of 2,163 KB as compared with 582 KB for the *XC4VFX12*. The full reconfiguration time of the *XC4VLX60* was 262.2 ms as compared with 70.55 ms for the *XC4VFX12*, with the partial reconfiguration time being 14.67 ms (based on 121 KB partial bitstream and JTAG mode) in both devices leading to speed-up in reconfiguration time of ~5x for the small FPGA and ~18x for the large FPGA. This finding is of particular importance to server solutions applications, where larger FPGAs are usually deployed to facilitate the placement of multiple cores on demand.

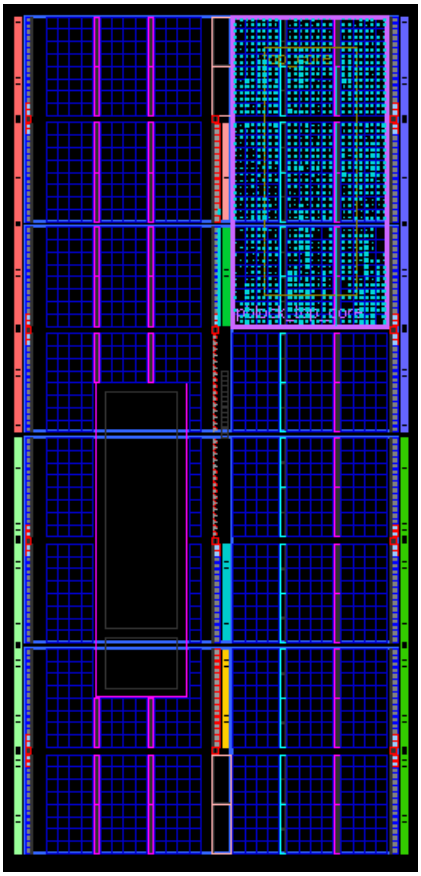


Figure 4.11: *Floorplan of the DPR implementation of the single-core based on Manhattan distance.*

4.6.4 DPR Implementation of Multi- core K-means Clustering

The results of the DPR implementation of a multi-core K-means clustering based on using eight K-means cores are presented in this subsection. The fitted cores were based on new modular architectures of the K-means core that exclude the divider block due to its large size. The purpose of this architecture was mainly to demonstrate if there are any advantages gained in terms of power or execution/reconfiguration time from using DPR over non-DPR implementation.

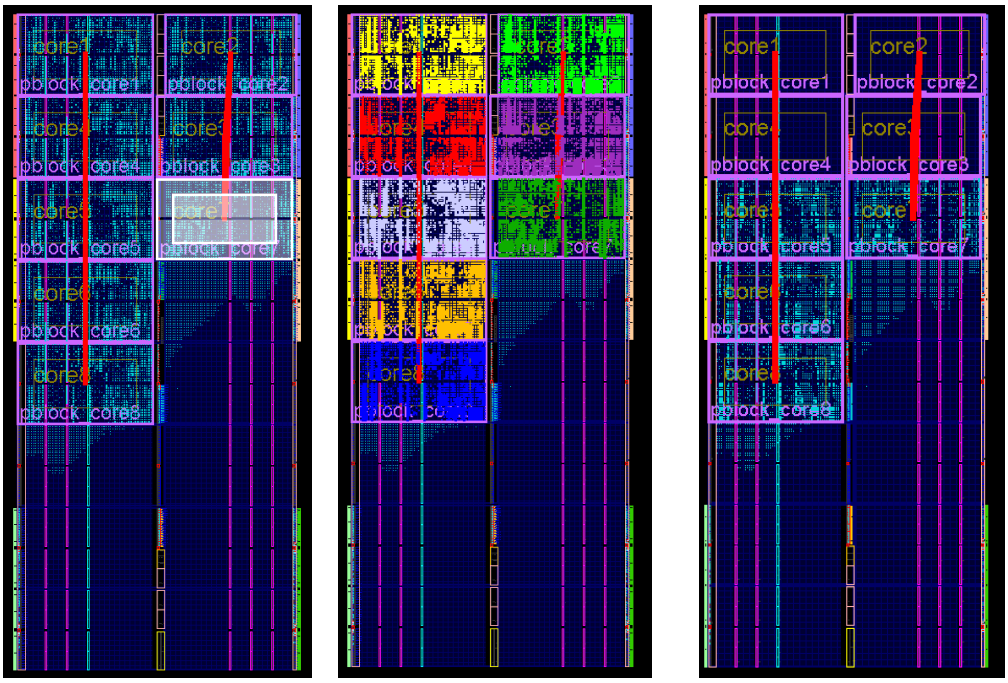
In this implementation, eight cores were fitted in a large FPGA, namely, the *XC4VLX60* which has abundant CLB slices and Block RAMs making it possible for setting all the eight K-means cores as reconfigurable partitions (RPs). The sizes of the RPs were made identical to add an advantage of making them re-locatable. The resources of the eight-core implementation with respect to the used single core are shown in Table 4.5.

Table 4.5: Place and Route results of the single-core and eight core implementation of the K-means clustering (cores exclude divider block)

Device	Xilinx' *XC4VLX60-12ff668*	
Parameters	(B=13, K=8, M=1, N=2905)	
Architecture	**Single-core**	**Eight-core**
Slices	1,178/26,624 (4%)	12,960/26,624 (48%)
4 input LUT's	2,111/53,248 (3%)	15,792/53,248 (29%)
FF	1,189/53,248 (8%)	12,312/53,248 (23%)
Block Ram's	5/160 (3%)	40/160 (25%)
Clock Frequency (MHz)		109.155

Several RMs were added for each of the eight cores corresponding to cores having different datasets, distance metrics, wordlength (B's) or of empty blocks (black boxes). Those RMs reflect variations in any of the core parameters B, M, N, or K, however the selected parameters do not exceed those used to set the pblock. Various configurations were implemented and the associated bitstreams were generated. The full bitstream was 2,163 KB while the partial bitstreams were all of 128 KB in size. According to (4.19), the reconfiguration time for the case when the FPGA is fully configured/reconfigured using the full bistream is 262.2 ms as opposed to 15.5 ms for the case when the FPGA is partially reconfigured. Consequently, in addition to the advantage of being able to dynamically alter the specifications of any K-means core without harming the integrity of the other cores or configurations running on the same FPGA, DPR offers ~17x speed-up in reconfiguration time over full chip reconfiguration. Furthermore, being able to re-locate any one of the eight cores within the same FPGA adds some level of fault tolerance to the implementation as well as increases the flexibility of the implementation. Fig. 4.12 illustrates the floorplan of three implementations based on different configurations, consequently the logic within each core have different density reflecting the preset parameters.

Lastly, the K-means-cores could be turned off when un-used or not-needed as a power reduction measure as shown in Fig. 4.12(c). Fig. 4.13 shows the placed and routed image of the eight-core implementation shown in Fig. 4.12(a).



(a) (b) (c)

Figure 4.12: *Floorplan image of the eight-core DPR implementation illustrating the difference in logics density corresponding to different configurations :(a) and (b) configurations based on RMs of different wordlengths and distance metrics, and (c) illustrates a case where some cores are set as black boxes corresponding to un-used cores.*

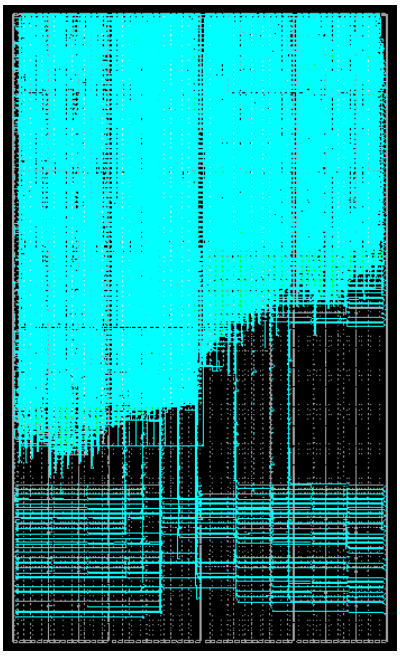


Figure 4.13: *The FPGA Editor image of the routed DPR implementation of the eight K-mean's cores, illustrating compact placement and routing.*

4.7 Comparison between GPU and FPGA-based implementations of the K-means Clustering

Graphics Processing Units (GPUs) have been gathering a lot of attention and popularity in the computing community as being affordable hardware accelerator to many applications. Initially, the use of this technology was confined to the gaming industry, however in recent years this has changed where GPUs established themselves as accelerators to many algorithms. K-means clustering is one of the most popular supervised data mining techniques, attempts to accelerate the performance of this application were not limited to FPGAs, GPUs have also reported promising results especially when compared with GPPs. In the following two subsections, a review about most recent implementations of the K-means clustering on GPUs will be presented followed by a comparison of the FPGA implementation presented in this thesis with few recent GPU implementations.

4.7.1 Prior work on GPU Implementations of K-means Clustering

Several implementations of the K-means clustering GPUs have been reported in the literature. In 2008, Fairvar implemented K-means on GPU and achieved speed-up of 13.57x (the GPU implementation took 0.724s compared to 9.830s on GPP) when clustering one million data points into 4000 clusters using Nvidia's GeForce 8600 GT and a 2 GHz GPP host, respectively [53].

In [70], good results were reported when K-means was implemented on two different types of GPUs using different datasets. The reported speed-ups for clustering 200K to 1M on the Nvidia's GeForce 5900 GPU were between 4x to 12x, over equivalent GPP implementations running on 1.5 GHz Pentium IV GPP. A higher speed-up of 30x was reported when using Nvidia's GeForce 8500 GPU and 3 GHz Pentium IV GPP.

In [74], the authors reported that the performance of their GPU implementation was less affected by the size of the dataset as compared to GPPs. Another result reported by the same authors was related to the effect of the number of clusters on the speed-up figures. The GPU implementations achieved speed-ups between 10x and 20x over GPP's for clusters less than 20 using the Nvidia's GeForce 5900 GPU, and more than 50x when more than 20 clusters were used. Additionally, when 32 clusters were used, the reported speed-up was 130x on the Nvidia's GeForce 8500 GPU.

In 2010, Karch reported a GPU implementation of K-means clustering for accelerating colour image segmentation in RGB space [75]. This work reported extensive comparison between GPP and GPU implementation of the K-means clustering for different number of clusters and dimensions. As such, the next subsection will be based on comparing the FPGA implementation presented in this chapter with the one in [75].

Moreover, Choudhary *et al.* reported in 2011 the GPU implementation of the K-means clustering using Nvidia's GeForce 8800GT that achieved speed-ups between 9x and 40x for datasets ranging in size from 10,000 to one million data points that were clustered into 20 clusters [76]. In addition, the group has stated an interesting finding that speed-ups of the GPU implementations over the equivalent GPP's increase as datasets grew in depth (more data points or patterns).

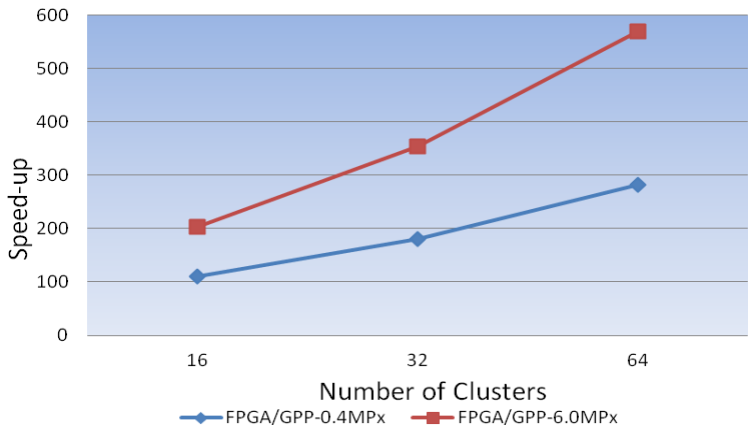
4.7.2 Comparative results: FPGAs vs. GPUs

When comparing the performance of the FPGA implementation of the K-means clustering reported in this work with the recent GPU implementation presented in [75] for an image processing application, the FPGA solution was found to excel that of GPU in terms of speed as shown in Table 4.6. The results shown are based on two different datasets; one is 0.4 Mega Pixel (MPx) in size while the other is 6.0 MPx. Both datasets were processed for 16, 32, and 64 clusters and were for single dimension. The GPP and GPU results were based on 2.2 GHz Core 2 Duo, with 4 GB memory and Nvidia's GeForces 9600M GT GPU running Microsoft Windows 7 Professional 64-bit. On the other hand, the targeted FPGA device was Xilinx' *XC4VSX35*, based on implementing the design using 13 bits to represent each point within the dataset running at a maximum clock frequency of 141 MHz. The Virtex device used in this comparison is not a top of the range FPGA, the latter can achieve higher clock speeds, however an attempt was made to limit the choice to a reasonable size that can accommodate the design and be reasonable for comparing with the abovementioned GPU device. The two images were too large to be stored within the FPGA, therefore, off chip memory is needed to store data and stream them to the FPGA pixel by pixel, and one data point was read every clock cycle. The processing times reported in [75] do not include the initialisation of cluster's centroids and the input/output overheads, similarly with the FPGA times reported in Table 4.6.

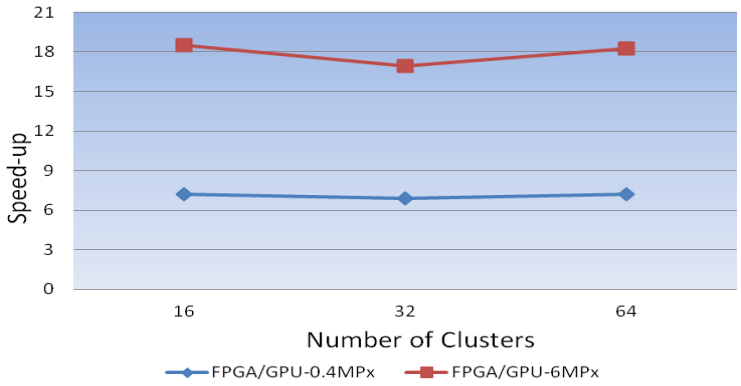
Table 4.6: The Execution Result of K-means in GPP, GPU, and FPGA for data of single dimension

Clusters	GPP Avg time per iteration (sec) [40]	GPP Avg time for complete execution (sec)[40]	GPU Avg time per iteration (sec) [40]	GPU Avg time for complete execution (sec)[40]	FPGA per iteration (sec)	FPGA Complete execution (sec)
0.4 MPx						
16	0.269	4.314	0.021	0.443	0.0028	0.039
32	0.516	7.637	0.020	0.421	0.0028	0.042
64	1.004	12.78	0.023	0.508	0.0028	0.045
6 MPx						
16	4.279	67.07	0.256	5.176	0.0425	0.723
32	8.144	110.7	0.247	4.439	0.0425	0.638
64	15.86	208.2	0.270	5.220	0.0425	0.723

The speed-up results of the FPGA implementation over the GPP and GPU results are illustrated in Fig. 4.14 for both images (0.4 MPx and 6 MPx) against three different number of clusters (16, 32 and 64). From Fig. 4.14(a), it can be stated that the FPGA implementation excel the equivalent GPP as the number of clusters were increased. Similar observation was found when comparing the GPU implementation with the equivalent GPP as reported in Table 4.6. As such, it can be inferred that the FPGA and GPU implementations outperforms the equivalent GPP implementation as the number of clusters was increased. This is due to the fact that both FPGA and GPU apply parallelism to the distance computation part while GPP performs this computation sequentially. Over and above, the FPGA/GPU acceleration is not greatly affected by the number of clusters (up to 64 clusters in the shown experiments) as illustrated in Fig 4.14(b) which shows nearly constant performance with increased number of clusters. This behaviour can be attributed to the fact that both technologies scale well with increased number of clusters.



(a) FPGA/GPP



(b) FPGA/GPU

Figure 4.14: *Performance of the FPGA implementation of the K-means clustering presented in this thesis as compared with the GPP and GPU implementations presented in [40], where: (a) illustrate the performance of FPGA versus GPP for different clusters, and (b) FPGA versus GPU.*

As for the device utilisation, the *XC4VSX35* FPGA used in this comparison has 15,360 CLB slices which were enough to implement the logic required to accommodate the number of clusters shown in Table 4.6. For the case of 16 clusters, the FPGA implementation occupied 5,177 CLB slices (33%), and with 32 and 64 clusters, the implementation occupied 8,055 (52%) and 13,859 (98%) CLB slices, respectively.

Additionally, the effect of data dimensionality on the performance of both FPGA and GPU implementations was investigated in this work based on comparing the FPGA implementation presented in this work with the GPP/GPU implementations reported in [77]. The authors of [77] reported results of clustering Microarray Yeast expression profiles in GPP and GPU as shown in Table 4.7, where GPU achieved speed-up of 7x to 8x over an equivalent GPP implementation for the case of four and nine dimensions, respectively; while the FPGA implementation achieved speed-up of 15x to 31x for the same dimensions. The three implementations, namely, GPP, GPU and FPGA were based on a dataset of 65,500 vectors clustered to three and four clusters. The FPGA implementation presented in this thesis was compared with the GPP and GPU implementations reported in [77] and found that the FPGA implementation achieved speed-up between 2x to 7x over the equivalent GPU implementation as shown in Fig. 4.14. The results shown in Table 4.7 were based on Xilinx' *XC4VSX35* FPGA, and Nvidia's 8600 GT GPU.

Over and above, when specifically comparing the performance of the FPGA implementation with the equivalent GPU for different dimensions, FPGA demonstrated superior performance to GPU as shown in Fig. 4.16. The performance drop in GPU as the

dimensions were increased is thought to be as a result of the way the implementation utilises resources within the GPU when computing specific kernels as data increase in size, particularly with relation to memory bottlenecks. GPUs suffer from limited access to local memory as data grow in size or dimensions leading to the requirement for accessing global memory, whereas FPGA benefits greatly from parallel access to Block RAMs which are abundant in recent FPGAs. Furthermore, Table 4.7 reports the estimated speed-up results of the five-core FPGA implementation of the K-means clustering which was presented in subsection 4.5.2, 4.6.2, and [72]. In summary, the five-core FPGA approach clearly outperforms the GPU implementation as shown in Fig. 4.15 for problems requiring small or reasonable number of clusters and dimensions that can be mapped easily onto commercially available FPGA devices. In addition, the FPGA implementations based on larger number of cores are due to benefit further from higher performance compared to GPPs and GPUs.

Table 4.7: The Execution Result of K-means in GPP, GPU, and FPGA for Multi-dimensional data

Clusters DIMENSIONS=4	GPP time per iteration (sec) [77]	GPU time per iteration (sec) [77]	GPU/GPP Speed-up	FPGA time per iteration (sec)	FPGA/GPP Speed-up	FPGA/GPU Speed-up	Five-core FPGA/GPU
3	0.0495	0.00623	8x	0.0019	26x	3.2x	16x
4	0.0652	0.00902	7.2x	0.0042	15.5x	2x	10x
DIMENSIONS=9							
3	0.1031	0.0125	8.2x	0.0019	54x	6.7x	33.5x
4	0.1333	0.01589	8.4x	0.0042	31.7x	3.8x	19x

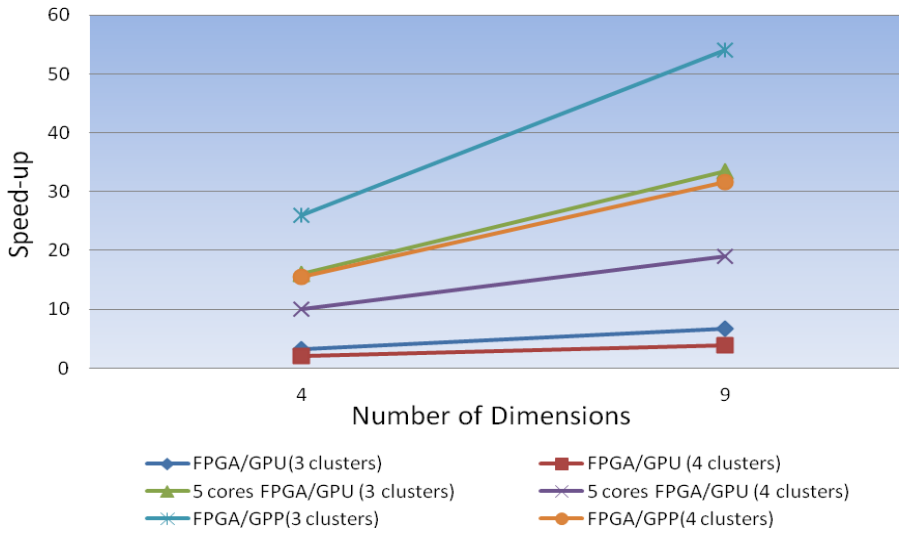


Figure 4.15: *The effect of data dimensionality on the speed-up performance of FPGA with respect to GPP and GPU for three and four clusters. The figure implies that FPGA outperforms GPP and GPU as the number of dimensions increase.*

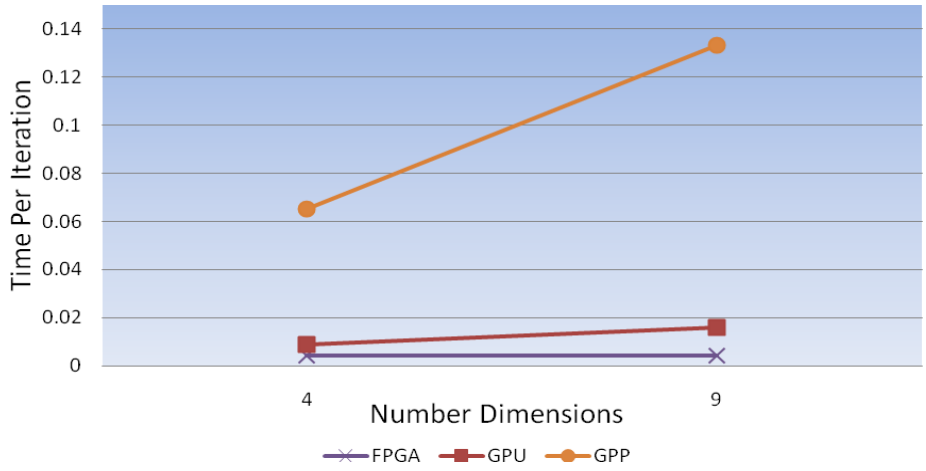


Figure 4.16: *The timing performance of the K-means clustering for a single iteration using FPGA, GPU and GPP with respect to different dimensions. The figure suggests that FPGA maintains superior timing performance with respect to GPP and GPU as the number of dimensions increase.*

However, the above findings are device specific and are not generalised unless fair comparison is made using higher end GPUs and FPGAs. Even then, large variations in the size of FPGAs within the same device family makes it difficult to assess the performance of the two technologies, especially that variations in GPUs within the same family range are much smaller than FPGAs. Furthermore, other issues will arise when high end devices are used such as cost of purchasing with FPGAs being more expensive than GPUs; and power consumption issue where GPUs and GPPs consume more power than FPGA. Nevertheless, the above comparative study was an attempt to highlight main performance bottlenecks such as memory limitations in GPUs and GPPs.

4.8 Comparative Power and Energy Consumptions: FPGAs vs. GPPs vs. GPUs

The power consumption of the K-means implementation in both FPGA and GPP were actually measured and compared. The power consumption of the single core implementation of the K-means clustering algorithm presented in subsection 4.5.1 was measured while running on Xilinx' ML 403 board and found to be 15 W only. Similarly, when an equivalent K-means implementation was running on 3.0 GHz Core2 Duo E8400 GPP, with 3 GB RAM, the measured power was 90 W. Consequently, a single core K-means implementation consumed six times less power than an equivalent implementation running on GPP while

being 32x faster than GPP as stated in subsection 4.6.1.1. Furthermore, the FPGA implementation was found to be 192 times more energy efficient than the GPP as deduced from (4.20):

$$\textit{Energy Efficiency} = \textit{Power Efficiency} \times \textit{Speed-up}, \tag{4.20}$$

where power efficiency is the power consumed by the GPP divided by the power consumed by the FPGA which was 6x in the above case.

With respect to the power consumption of the five-core implementation presented in subsection 4.5.2, the implementation was based on simulation results only as the larger device used for the implementation, namely, *XC4VLX2*5 was not available for actual measurement, a projection of the power consumption and energy efficiency is attempted in this subsection. The five-core implementation was 205x faster than GPP as stated in subsection 4.6.2; the single core GPP consumed 90 W and this figure will also be used for the five-core case; as for the FPGA power consumption for the five-core implementation, 30 W is estimated since most medium FPGA boards consume no more than 30 W [39] and [79]. Consequently, the FPGA implementation is estimated to consume around 3x less power than the equivalent GPP. As a result, the total energy of the five-core implementation is estimated to be 615x less than GPP as obtained from (4.20) based on speed-up of 205x of the five-core implementation over the equivalent GPP implementation reported in subsection 4.6.2. Please note the small GPP power is used due to difficulty to estimate the GPP power for the five-core implementation.

Additionally, the power and energy consumption of the FPGA, GPP, and GPU implementations of the K-means clustering were compared and reported in Table 4.8 based on the data reported in [75]. Both GPP and FPGA power figures were actually measured for the dataset reported in [75] based on using 16 clusters which fit into the available *XC4VFX12* FPGA. On the other hand, since it was not possible to actually measure the GPU power used in [75], the GPU power was obtained from the Nvidia's GeForce 9600 GT datasheet reflecting the power rating of the device [78]. The results in Table 4.8 are based on using thirteen bits to represent the 0.4 MPx image with the image being stored in an off-chip memory.

From Table 4.8, it can be stated that FPGA is ~8x more power efficient than GPP, and ~4x more power efficient than GPU. In addition, the FPGA implementation is ~615x more

energy efficient than the GPP, and ~31x more energy efficient than the GPU. Note that this implementation utilised 4,909 CLB slices (89%) of the targeted device.

Table 4.8: Comparison of Power and Energy consumptions of different K-means implementations based on 0.4 MPx image and 16 clusters presented in [75]

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

Note that the GPP power was taken while the K-means clustering algorithm was running in Matlab in a loop, and GPP power was 70 W when idle.

4.9 Summary and Conclusions

In this chapter, a detailed design and implementation of the K-means clustering algorithm in FPGA was presented and labelled as a single-core implementation of the K-means clustering. The core was captured in Verilog HDL and was parameterised in terms of number of clusters, wordlength, data depth and dimensions. The architecture composed of various kernel blocks performing specific computations of the algorithm in parallel exploiting the possible parallelism within the K-means algorithm. This implementation outperformed equivalent implementations in GPP and GPU by two orders and one order of magnitude, respectively. Based on this single-core, four more FPGA implementations of the K-means clustering were presented. The first is a multi-core implementation while the remaining three are based on DPR.

The first novel DPR implementation of the K-means clustering was based on setting a specific kernel within the K-means algorithm as reconfigurable partition, namely, the distance kernel block. This implementation allows swapping out the distance kernel and replacing it with another one that is based on different distance metric. The exchangeable distance kernels correspond to the Manhattan and Euclidean metrics allowing the user to choose one or the other based on the application in hand. In addition to this newly added flexibility, the reconfiguration time of the DPR implementation was found to be ten times faster than a non-DPR implementation. The second novel DPR implementation was based on partially reconfiguring a single K-means core, the implementation was five times faster in partial reconfiguration time than full chip reconfiguration. The third DPR implementation was based on a reconfigurable multi-core architecture whereby multiple K-means cores

(eight-core implementation was actually presented) can be partially reconfigured. The reconfigurable eight-core implementation was found to be seventeen times quicker in partial reconfiguration than full chip reconfiguration.

A relationship between the size of the FPGA used in any DPR implementation and the speed-up in reconfiguration time was observed, the larger the FPGA device, the higher the speed-up in partial reconfiguration time compared with full device reconfiguration given that the comparison is with respect to identical reconfigurable partitions (RPs).

The main advantages of the DPR implementations presented in this chapter are the possibility to alter individual kernel or cores at run time without interrupting other processes running onto the same FPGA, turn off un-used cores to save power, re-locate cores when faults occurs in the FPGA fabrics leading to increased immunity of the implementation to faults, re-locate cores when tasks need to be re-arranged to allow for other tasks to be added onto the device. Additionally, the multi-core implementation particularly targets server solutions where cores are inserted or altered on demand during run time without disrupting other tasks placed onto the FPGA.

Lastly, the power consumption and energy efficiency of the FPGA implementation were compared with GPP and GPU. The FPGA implementation was eight times more power efficient than GPP and four times more power efficient than the equivalent GPU implementation. As for the energy efficiency, the FPGA implementation was estimated to be 615 times more energy efficient than GPP and 31 times more energy efficient than the GPU implementation.

In conclusion, modern FPGA technology render the implementation of the K-means clustering in FPGA faster, flexible, scalable, dynamically reconfigurable, more efficient in terms of power and energy consumptions.

Future work includes applying embedded processors to dynamically reconfigure the FPGA for all of the aforementioned DPR implementations and test all of them on high end FPGAs if they become available. In addition, the main architecture of the K-means clustering algorithm could be modified to implement the division operation in hard or soft core processors available in most modern FPGAs. This last approach may help reducing the area occupied by the K-means core allowing for more cores to be integrated into the same chip.

Chapter-5

Hardware Implementation of K-NN Classification on FPGA

5 Hardware Implementation of the K-Nearest Neighbour Classification on FPGA (K-NN)

5.1 Introduction

The K-nearest neighbour (K-NN) classifier is one of the widely used supervised classification algorithms in pattern recognition and data mining. In many applications such as bioinformatics, image processing of satellite and medical images, data retrieval, and many others, K-NN is used to classify an unknown sample or a query of multi-dimensions (features) to a class label, using samples with known class labels which are referred to as the training set. The classifier requires high computational power when the number of features and samples in the training set are high. Due to such high computing demands, the classifier renders itself candidate for hardware acceleration exploiting the parallelism and pipelining opportunities inherent in the classifier computations [80]-[82].

In bioinformatics, the K-NN classifier is a non-parametric technique commonly used in the analysis of Microarray data such as in class discovery to define new un-recognised cancer subtypes, or in class prediction to assign unknown samples to known class labels [18], [22] and [24]. The classification of highly dimensional Microarray data using K-NN is a time-consuming task when implemented on GPPs, and such it can benefit greatly from a parallel hardware implementation. K-NN classification requires the computation of distances between a query and all members of a training set which can be very time consuming when implemented on GPPs for high dimensional data. Therefore, to actually be able to exploit the high potentials of Microarray in the clinical field, K-NN classification methods needs to be accelerated through the use of parallel hardware such as multi-core GPPs, GPUs or FPGAs.

The aim of the work presented in this chapter is to design and implement a highly scalable and parallel K-NN classifier on FPGA that is adaptive to the number of training features or dimensions (M), number of training samples or vectors (N), class labels (C), number of neighbourhoods (K), and number of bits per sample or feature (B). In addition, the role of DPR is investigated in partially reconfiguring the K-NN core and in reducing the reconfiguration time of portion of the K-NN core or the complete K-NN with respect to full chip reconfiguration.

The remainder of this chapter will first present background on the K-NN classification followed by an overview of prior work in the area of hardware implementation of the K-NN

classifier. Eight proposed FPGA architectures of the K-NN classifier will then be presented; two variable single-core implementations will be presented based on different architectures. The details about the two architectures will be given including the kernels constituting the K-NN classifier. Second, a multi-core architecture based on combining multiple K-NN classifiers will be presented and its advantages are highlighted. Third, the role of applying DPR is investigated through the proposal of five DPR based implementations of the K-NN classifier. The first is based on partially reconfiguring specific kernel within the single-core K-NN classifier to provide a mean of altering the logic associated with changing a specific parameter affecting the logics inferred for that kernel. Building on the same concept, the whole single-core is then made reconfigurable. Third, a DPR implementation of a multi-core K-NN classifier is proposed. Then, a DPR implementation of the K-NN ensemble classifier based on the multi-core approach is presented and discussed, this implementation is an extension of the multi-core implementation and includes an additional circuitry to combine the results of multiple classifiers. The last DPR implementation is based on reconfigurable memory block, and application applicable to large FPGA only. Following this, the results of the aforementioned implementations are outlined and discussed. Finally, summary and conclusions are laid out along with plans for future work.

5.2 Background on K-NN Classification

The KNN classifier aims to identify the class label of an unknown sample or query based on the class labels of the training set, the latter is a form of a large matrix where rows are called training samples (N) and columns are called features (M), with an additional column representing the class labels of each of the training samples (L's). Therefore, samples in the training set are vectors of M dimensions having known class labels. The way the classifier works is by first calculating the distances between a query (unknown sample) and all of the training samples using a particular distance metric e.g., Euclidian or Manhattan distance. If a query with M dimensions is known as Y= {y_1, y_2, y_3, y_4,. . .,y_M}; N training samples in M-dimensional feature space, each is known as X_{Ni}={x_{N1}, x_{N2}, x_{N3}, x_{N4},. . ., x_{NM}}; and the class labels are L= {L_1, L_2, L_3 ,. . .,L_N}, then the distance between the query Y and one training sample X using Euclidean metric shown in equation (5.1).

$$D(X_i, Y_i) = \sqrt{\sum_{i}^{M} (X_{Ni} - Y_i)^2}, \tag{5.1}$$

where D(X, Y) is the distance between the two vectors X and Y. Although the Euclidean distance is commonly selected for computing distances in many data mining applications, another metric called the Manhattan or City Block is chosen in the implementations presented in this work as shown in equation (5.2) for its simplicity and lower cost compared to the Euclidean [72]-[73].

$$D(X_i, Y_i) = \sum_{i}^{M} \left(| X_{Ni} - Y_i | \right) \tag{5.2}$$

At the end of the distance computation phase, N distances corresponding to the accumulative distances between a query and all features of each of the training set are compared and sorted according to a user parameter called the number of neighbourhoods (K). The classifier obtains the K-minimum distances and sorts them in a descending order along with the class labels associated with the corresponding samples; those K-minimum distances are known as the K nearest neighbours or KNNs. Finally, the classifier performs a voting on those KNNs to assign the query to the most encountered KNN. The steps of the algorithm are very simple in that no complicated arithmetic operations are involved other than the additions and subtractions associated with the distance calculation and voting. However, similar to K-means clustering, the distance computation is the most time consuming part as it involves repeating the computation involving subtractions and additions between the query and all the samples in the training set. Thus accelerating K-NN classification can be achieved mainly through parallelising the distance computation part and pipelining all the other operations.

5.3 Prior Work on Hardware Implementation of the K-NN Classification on FPGAs

Most recent works on hardware implementation of the K-NN classifier were reported in [7] and [8]. The authors presented two different architectures of the K-NN classifier in the form of parameterised IP cores captured in VHDL. The two IP cores were adaptable in terms of the number of neighbourhoods (K), training samples (N), features (M), class number (C), and number of bits per feature (B). The two IP cores were designed as linear systolic arrays which vary in the number and arrangement of the processing elements (PEs) in the array.

Such variation aimed at allowing the user to select the most appropriate architecture for the targeted application based on the available hardware resources, cost, and desired performance. For instance, some classification problems use high dimensional data (large M) and small number of training samples (N) while others may have the exact opposite case. The authors of [83] and [84] referred to the first K-NN architecture as (A1) and to the other as (A2). A1 has a total number of PEs of M+K+1, while A2 has a total number of 2N+1 PEs, thus the inferred logic for A1 is affected mainly by M while those of A2 are affected by N. Such flexibility in choosing between A1 or A2 allows the user to trade-off performance with area depending on the classification problem in hand and the available hardware resources, where A1 is used to target applications whereby N>>M, whereas A2 is used to target applications whereby N<<M. The two architectures presented in [83] and [84] are illustrated in Fig. 5.1 and Fig. 5.2. The authors tested the two architectures using real SPECT dataset of M=44, N=80, B=9 bits, K=5, and C=2 and found that A2 outperformed A1 by almost two times, however A2 consumed three times more logic than A1 when using the same FPGA device [83], this example demonstrated the trade-off between area and performance.

In addition, the authors of [84] compared the performance of several A1 implementations using Xilinx' Virtex-II Pro *XC2VP30-6* FPGA with the Nvidia's GeForce 8800GTX GPU implementations presented in [85] and found that the FPGA implementation outperformed the GPU by between 1.5x to 3x depending on the parameters used.

Another FPGA implementation of a special case of the K-NN classifier known as the nearest neighbour (1-NN) was presented in [86] which achieved relevant speed-up over GPP. As the name implies this implementation was for the case of K=1, which assigns the query to the class label of the closest training vector. The architecture was based on instantiating distance PEs of a number equal to the number of features M to calculate the distances between all features of the query and a training sample in parallel. These PEs are used to calculate the Euclidean distances, they iterate N/M times to cover all the samples in the training set as illustrated in Fig. 5.3. This design is obviously suitable for the case of large N and small M, as a result, the size of this hardware architecture is dependent on M, similar to the A1 architecture reported in [83] and [84].

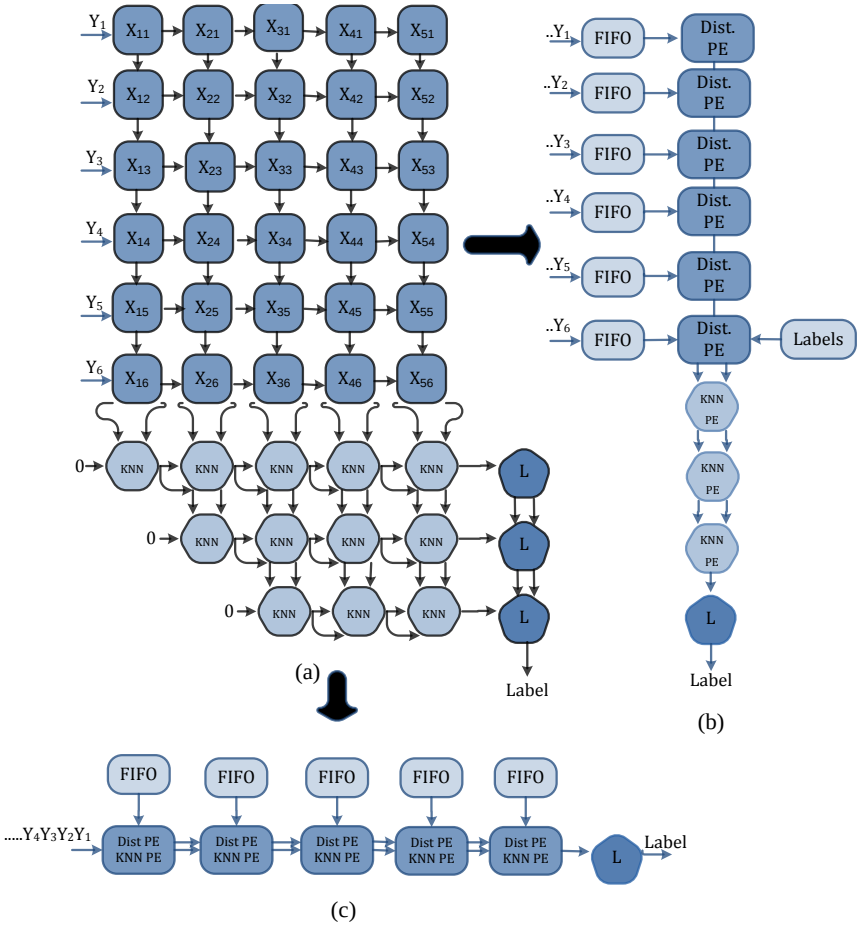


Figure 5.1*: (a) The architecture of a complete K-NN classifier based on M=6, N= 5, and K=3, illustrating the number and arrangement of three types of processing elements (PEs), where (b) A1 architecture (N>>M), and (c) A2 architecture (M>>N).*

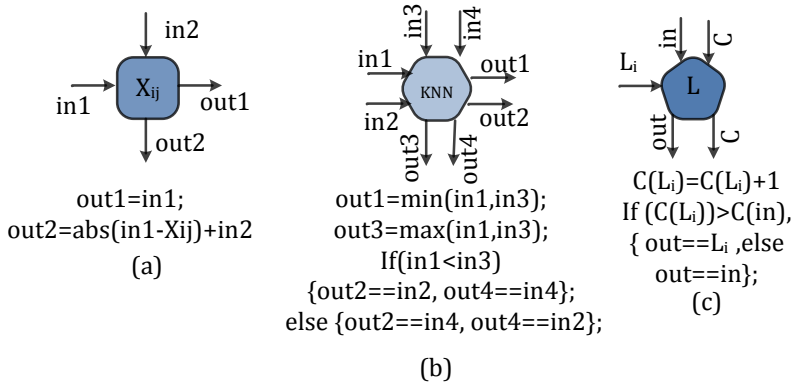


Figure 5.2: *The functionality of three different types of PEs involved in the architecture of the K-NN classifier which perform the: (a) subtraction-addition, (b) comparison, and (c) voting.*

When M is too large, the hardware resources may be too large to fit into small or medium size FPGAs, thus restricting the use of this architecture to specific applications having small M. The design works by first loading the query (vector having M dimensions) into off-chip memory and the training samples (N) into on-chip memory, then starts computing the distances between the query and all training samples. A set of comparators were then used to obtain the minimum distance and its class label, known as the 1-NN. Finally, the query gets assigned to the class labels of the 1-NN. The design was captured in Handel-C, and implemented using Xilinx' Coregen. The authors validated their design using two medical datasets: one was for Breast Cancer and the other for Prostate Cancer and compared their results with GPP. The FPGA implementations were performed using three FPGAs: Xilinx' Virtex-E, Virtex-II, and Virtex-IIP FPGAs. Speed-ups reported by the authors for the Breast Cancer dataset using 14 bits over a Pentium IV GPP implementation were 15x, 38x and 47x using Virtex-E, Virtex-II and Virtex-IIP, respectively. As for the Prostate Cancer dataset, speed-ups reported were 6x, 17x and 19x using Virtex-E, Virtex-II and Virtex-IIP, respectively [86].

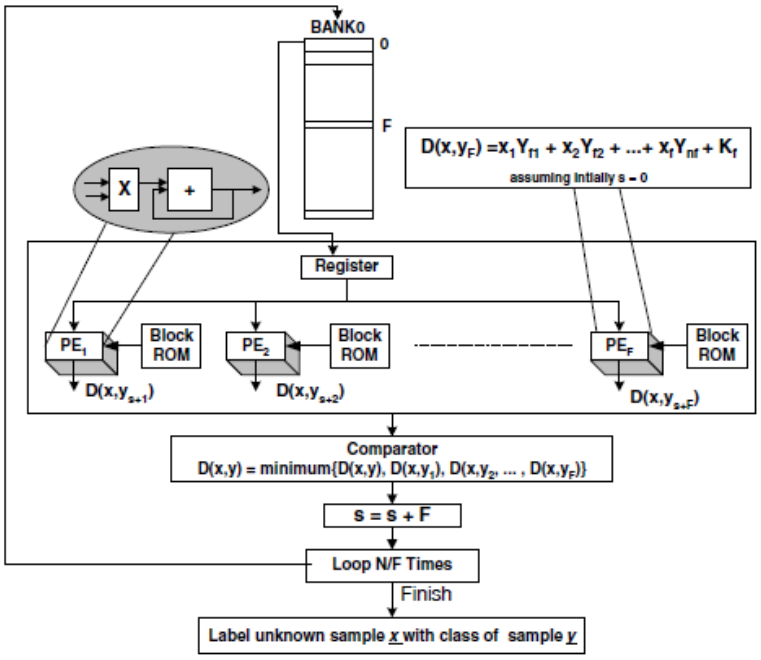


Figure 5.3: *The architecture of the 1-NN proposed in [10] suitable for the case of N>>M. The architecture is based on having a number of processing elements (PEs) equal to the number of features F (or M in A1 architecture), where each PE is responsible for computing one distance and have a Block RAM associated with it for supplying the PE with one feature at a time (source ref. [86]).*

5.4 Novel Hardware Implementations of the K-NN Classifier on FPGAs

The subsequent subsections will present the architectures and theory of eigh different implementations of the K-NN classifier including the followings: two single-core architectures, namely, A1 and A2; multi-core K-NN classifier, DPR implementation based on a reconfigurable kernel within the K-NN classifier; DPR implementation based on a reconfigurable single-core K-NN classifier; DPR implementation of multi-core K-NN classifier; a DPR implementation of multi-core ensemble K-NN classifier; and finally a DPR implementation based on reconfigurable memory block. The DPR implementations are built on the bases of the kernels constituting the A1 and A2 architectures of the K-NN classifier.

5.4.1 Single-core Architecture

The proposed two architectures of the single-core K-NN classifier are based on the architectures presented in [83] and [84]. Here, a modular architecture has been constructed separating the main K-NN classifier kernels into blocks as shown in Fig. 5.4. The first block is the memory responsible for storing the training data. The second block constructs the distance computation systolic array having the role of receiving the query and the training samples and computes the distances between them. The third block is also a systolic array pipelined with the distance computation block, and has the role of comparing the incoming distances as they arrive to obtain the KNNs, hence named the KNN finder. The last block is the class label finder responsible for performing the voting among the KNNs to obtain the most frequently encountered class label and assign it to the query. The following subsections will provide more details on each block.

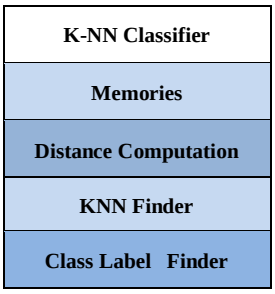


Figure 5.4: *The main blocks of the K-NN classifier.*

5.4.1.1 Memory Block

A) A1 Architecture

This block stores both the complete N by M training set, and the corresponding N labels in two different types of FIFOs (the first to store the training set while the other is to store their corresponding class labels). This block is made adaptive to the parameters B (feature's wordlength), M (number of features or dimensions), and N (number of training samples), where M-FIFOs are instantiated by the core design (written in Verilog) to store the complete training set each having a width of B and a depth of N. Each of these FIFOs is associated with one of the distance Processing Element (PE) and supply it with one feature every clock cycle as can be seen in Fig 5.1(b), therefore allowing M distance PEs to receive data simultaneously and process them in parallel in a pipelined manner.

On the other hand, the features of the query get streamed to the distance PEs one by one where they get stored locally inside registers as they are required every clock cycle until all the training samples are completely processed. As for the single FIFO used to store the class labels, it has a depth of N and width corresponding to the wordlength required to represent the largest class label (C). The role of this FIFO is to supply the PEs which are responsible for finding the KNNs with class labels every clock cycle, it starts operating only when the first distance result is ready that is after a latency of M clock cycles as will be discussed in subsequent subsections.

B) A2 Architecture

The function of the memory block is similar to that of A1 architecture; however the specifications, arrangement, and number of FIFOs are different. In A2 architecture, there are N-FIFOs each having a depth of M used to store the training set, where each FIFO is associated with a distance PE as shown in Fig 5.1(c). Since the number N is usually small in A2 architecture, the class labels are usually stored in registers, while the query gets stored in a FIFO having a width of B and depth of M.

In summary, the size of the memory used to store the training set in both architectures is based on the wordlength of each feature, the total number of training samples (N), and the number of features (M). The depth of memory in each architecture is different: in A1 the depth is N while in A2 it is M, however the total number of Block RAM needed to store the

training set will be the same for both architectures. When using Xilinx' Virtex-4 FPGA, the number of 18K Block RAMs needed to form this FIFO is given by equation (5.3).

$$Block\ RAMs\ required = [B \times M \times N]/18K \tag{5.3}$$

For instance, when 18 bits are used to represent each feature of a dataset containing 8192 training samples having a total of sixteen features, 128 of (1K x 18) Block RAMs would be needed to store the training set, in addition to two Block RAMs used to store the labels associated with the 8192 samples given that three bits were needed to represent each label (based on C=4).

5.4.1.2 Distance Computation Block

A) A1 Architecture (N>>M)

The number of distance PEs is different in the two architectures as well as the way they operate. For the case of A1 architecture shown in Fig. 5.1(b), the block consists mainly of a systolic array of M distance PEs which have the role of receiving M features corresponding to all dimensions of one training sample along with the associated features of the query every clock cycle. These M PEs are pipelined to compute the Manhattan distances between the features of the query and all the sample's features in parallel. This approach parallelises the distance computation whereby M distances get computed and accumulated simultaneously, the architecture of the systolic array is illustrated further in Fig. 5.5 for the case of M=4. Each of the PEs has a localised FIFO associated with it as described previously to supply each PE with one feature every clock cycle in a pipelined manner to ensure efficient utilisation of the hardware resources. The latency of this block is M cycles and the throughput is one distance result every clock cycle. Consequently, the time needed for computing the distances between a query and the complete training set is given by equation (5.4).

$$Distance\ computation\ time\ (clks) = M + N \tag{5.4}$$

The hardware resources inferred by this block are adaptable to the parameter M, whereby M PEs get generated automatically by the core based on the value of M specified by the user. Fig. 5.5(b) simplifies a single distance PE which consists mainly of a unit responsible for obtaining the Manhattan distance and an adder for accumulating the results.

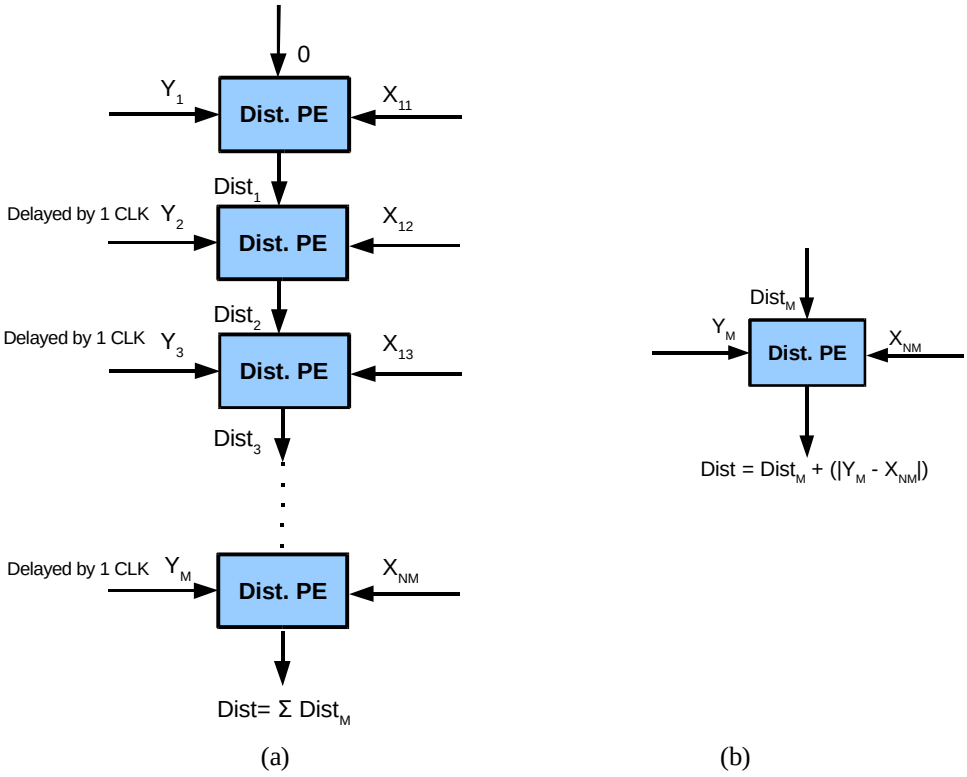


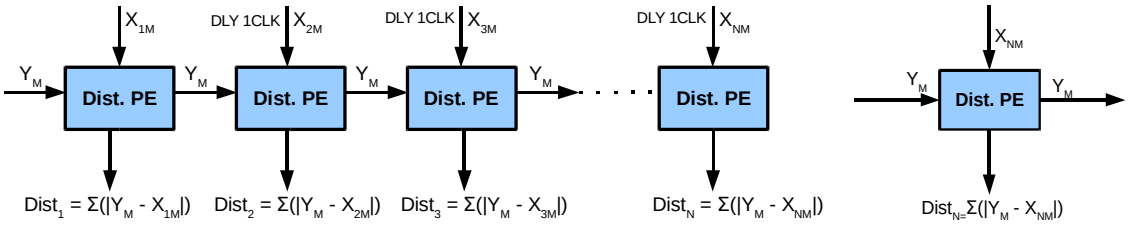
***Figure 5.5**:(a) Systolic array of M distance PEs of the A1 architecture, the result of the last PE is the sum of distances between a query and a training sample across all features which gets connected to the top KNN PE, and (b) the functionality of the single distance PE is illustrated.*

B) A2 Architecture (M>>N)

In contrast to A1 architecture, the systolic array consists of N distance PEs each having its own local FIFO, which supply each PE with one feature every clock cycle. Additionally, the query FIFO supply the systolic array with one feature every clock cycle, which gets pipelined to the adjacent distance PE. Consequently, the systolic array receives N training features every clock cycle and one query feature as shown in Fig. 5.1(c); and processes them simultaneously in a pipelined manner to ensure efficient utilisation of the hardware resources. It can be stated that in this architecture each PE process a complete training sample consisting of the computation of distances between the received features and the accumulation of those distances, hence no involvement of adjacent PEs in processing a single sample as was the case in A1 architecture, this functionality is illustrated further in

Fig. 5.6. As a result, the systolic array has a throughput of one distance result every clock cycle and a latency of M clock cycles. Similar to A1 architecture, the time needed to complete the computation of distances between a query and the complete training sample in A2 architecture is M+N clock cycles.

In contrast to A1 architecture, the hardware resources inferred by this block for architecture A2 are adaptable to the parameter N instead of M, whereby N distance PEs get generated automatically by the core based on the value of N specified by the user. Fig. 5.6(b) simplifies a single distance PE which consists mainly of a unit responsible for obtaining the Manhattan distance and an adder for accumulating the results. From comparing the functionality of this block in the two architectures, it can be concluded that A2 architecture involves significantly larger number of arithmetic additions than A1 architecture when processing one training sample, which are required to perform the accumulation of distances across the M features.



(a) (b)

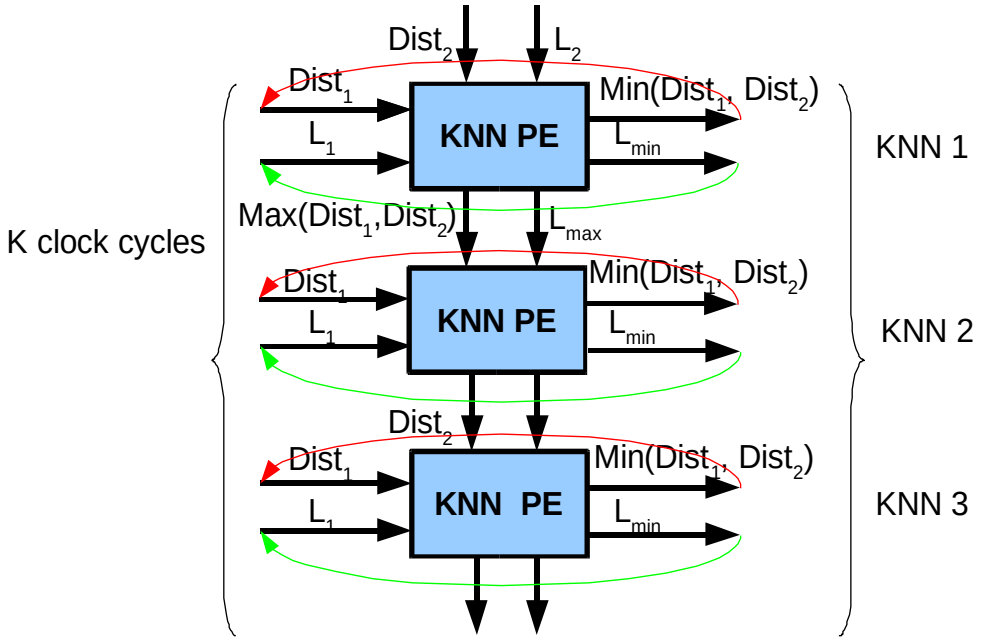
Figure 5.6: *(a) The systolic array of the distance kernel in A2 architecture, (b) the functionality of a single distance PE.*

5.4.1.3 K-Nearest Neighbour Finder Block (KNN Finder)

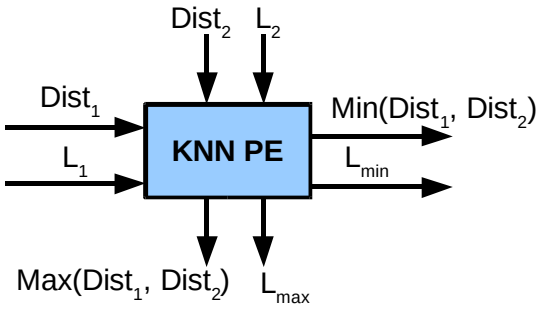
This block becomes active once the first distance result becomes ready, that is after latency period of M clock cycles in both architectures. Then it starts receiving the N distances coming from the previous block along with the N class labels supplied by the Label's FIFO every clock cycle. The block consists mainly of a set of comparators called KNN PEs used to compare the received distances and obtain the KNNs as shown in Fig 5.1; and works in parallel with the distance computation in a pipelined manner. The functionality of a single KNN PE is realised in Fig. 5.2(b). The following subsection will provide more details about the arrangement, functionality, and interconnectivity of this block for A1 and A2 architectures.

A) A1 Architecture (N>>M)

In A1 architecture, the block consists mainly of a systolic array of K number of KNN PEs arranged vertically as shown in Fig. 5.7(a). The top PE is directly connected to the distance PE above it as illustrated in Fig 5.1(b) supplying the KNN finder array with one distance every clock cycle after an initial latency of M clock cycles, which is the latency required to fetch the first result from the distance systolic array. The functionality of each KNN PE is illustrated in Fig. 5.7(b), where each KNN PE receives four inputs corresponding to two distances along with the labels associated with them, and outputs four results corresponding to the minimum and maximum distances along with their labels where the minimum distance and its label get fed back as an input to the same PE in the next clock cycle as illustrated earlier in Fig. 5.1(a), while the maximum distance and its associated label get propagated downward to the PE below it in the systolic array. Each PE is responsible for obtaining one of the KNNs, where the rightmost labels of each PE at the end of the processing time of this block would be the KNNs sought after. Those KNNs are sent sequentially to the next block as they become ready. The function of this block is completed after an N + K clock cycles.



(a)



(b)

Figure 5.7: *(a) The systolic array of the KNN finder block for A1 architecture for a case of K=3, (b) the functionality of a single KNN PE.*

B) A2 Architecture (M>>N)

In A2 architecture, the block consists of a systolic array of N number of KNN PEs arranged horizontally as shown in Fig. 5.8. Each KNN PE is directly connected to the distance PE above it as shown in Fig. 5.9 supplying the KNN array with one distance every clock cycle after a latency of M cycles. The functionality of each KNN PE is same as in A1 architecture which was shown in Fig. 5.7(b), where each KNN PE receives four inputs corresponding to two distances along with the labels associated with them, and outputs four results corresponding to the minimum and maximum distances along with their labels. However, the maximum distance and its label get fed back as an input to the same PE in the next clock cycle while the minimum distance and its associated label get propagated rightward to the adjacent PE in the systolic array as shown in Fig. 5.8. After N clock cycles, the KNNs are received sequentially at the rightmost PE one every clock cycle; which get fed sequentially to the next block to find the most frequently encountered label. Therefore, the complete processing time of this block is same as in A1 architecture being N+K clock cycles.

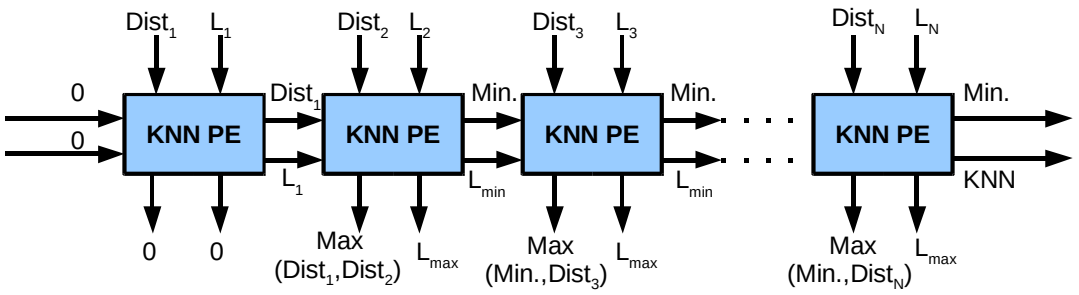


Figure 5.8: *The systolic array of the A2 architecture.*

5.4.1.4 Class Label Finder Block

This block gets signalled to start once the first KNN becomes ready from the previous block that is after N+M clock cycles and starts processing each of the received KNNs one by one taking K clock cycles to complete its operation. The functionality and resources of the block are identical in A1 and A2 architecture.

The block consists mainly of C counters each associated with one of the class labels, whereby a counter gets incremented by one when the incoming KNN being processed matches the ID of the counter. In addition to counting the received KNNs, the contents of the C counters are compared to determine the ID of the counter having the largest number of members. One comparator is used to compare the contents of the currently incremented counter with the result of the previous comparison, and then outputs the larger result along with the ID of the corresponding counter as illustrated in Fig. 5.2(c). The compare PE takes three inputs one is the KNN which is basically a class label while the other two inputs are the counter ID and contents from the previous comparison with these two set to zero for the initially received KNN. Accordingly, the top Label PE on Fig. 5.1 is only receiving one input instead of three (appearing in Fig. 5.2(c)) since there is no previous comparison result available at the start of this block. The two outputs of the compare PE are the maximum value along with the corresponding counter ID, which both get fed back to the compare PE when a new KNN is received as illustrated previously in Fig. 5.1(a). Lastly, the query gets assigned a class label equivalent to the ID of the counter having the largest value corresponding to the most encountered class label. The functionality of this PE is illustrated in Fig. 5.2(c).

Using single compare PE repeatedly reduces the overall logic utilisation and maximises the efficient use of the block. As for the hardware resources inferred for this block, they are mainly dependent on the user defined parameters K and C; the presented core is adaptable to both of them. This block is small in size when compared with the first two blocks. For instance, for the case when K=5 and C=4 only 28 CLB slices are consumed, for K=10, 47 CLB slices are consumed, and when K=19, 59 CLB slices are consumed. This small change can be attributed to the fact that when C is fixed to 4, only four counters are needed whose size depend on K. For example when changing K from 5 to 30, the wordlength of the counters change from five to seven bits only since this size is equal to the $\log_2$ (K). For a particular problem, C is usually fixed at the beginning of the implementation as it is a representation of the given class labels. However, K is a user defined parameter that is not

influenced by the supplied data and can be variable for a particular problem having the same training set and labels.

The above blocks are all controlled by one simple FSM, controlling the start and finish of the three main processing kernels: the distance computation, the KNN finder, and the label finder.

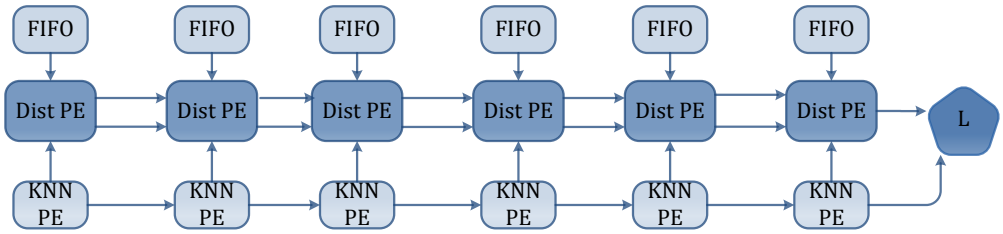


Figure 5.9: *The A2 architecture illustrating the number of PEs for the case of N=6, and the interconnectivity of the PEs of the three main blocks constituting the K-NN classifier.*

5.4.2 Multi-core Architecture of the K-NN classifier

The performance of the FPGA implementation of the two K-NN classifiers illustrated above can be enhanced through the implementation of multi-core K-NN classifiers. Several applications can harness multi-core K-NN classifiers. For instance, multi-core classifiers are capable of processing different queries using the same training set, or partitioning the training set among the multi-cores to further accelerate the classification of a query. Moreover, multi-core classifiers can be particularly useful in server solutions similar to that applied to the K-means implementation discussed in the previous chapter. In addition, the multi-core approach can be used to create an ensemble K-NN classifier which will be discussed in subsequent subsections. Based on the available resources and desired performance, a multi-core implementation based on A1 architecture will be presented in the following subsection.

5.4.2.1 Multi-core implementation based on A1 Architecture

A1 classifier consumes reasonable amount of resources even for large values of K commonly used in many classification problems. Consequently, this architecture can benefit from implementing multi-core classifiers. In this work, a quad-core K-NN classifier is created to implement an ensemble model that combines the results of the four classifiers. The four classifiers access the same data stored in the Block RAM's which has the training set as well

as the associated class labels. The ensemble classifier improves the prediction of a class label as a result of combining the outcome of several classifiers through voting; in this implementation those runs are based on each classifier being configured with different value of K. This implementation will be further implemented in DPR and the two will be compared in terms of configuration time and area footprint.

5.4.3 DPR Implementation of Partial/Single-core K-NN classifier

In the subsequent subsections, two novel implementations of the K-NN classifier based on DPR will be presented. The first implementation is based on identifying specific kernel in the classifier which could benefit from DPR, and make that portion of the classifier reconfigurable. As a result of investigating the kernels constituting the K-NN classifier and identifying the resources that are affected by the only user parameter involved in the classifier, namely, K, a decision was made to specifically investigate the benefits of dynamically reconfiguring the portions of the classifier that are affected by K. The second implementation is based on making the whole K-NN core reconfigurable rather than portion of it.

5.4.3.1 DPR Implementation of the K-NN Classifier based on Reconfigurable KNN core

Given the fact that K-NN performance is affected by the chosen number of neighbourhoods (K) [11], and that a classification problem may need to be repeated for different K's to ensure accurate classification, a novel architecture to investigate the benefits of applying DPR to the K-NN classifier is presented in this subsection. The aim is to set the portion of the K-NN classifier sensitive to the value of K to be dynamically reconfigurable taking advantage of the fact that other parameters e.g., (N, M, B and C) are fixed for a given classification problem. The advantage of this architecture is allowing the user to dynamically alter the portion of the classifier sensitive to K only without affecting other parts of the classifier. To apply this idea, a new modular architecture based on A1 architecture of the K-NN classifier presented in subsection 5.4.1 is formed by re-arranging the K-NN classifier blocks such that the kernels that are sensitive to the parameter K are grouped together into one block, now named as the KNN core, while others are slightly modified to compensate for the changes made in grouping some blocks. The newly created top design consists of

three blocks or cores only: the distance computation core, the KNN core and the memories. The KNN core is formed by combing the KNN finder and the class label finder blocks of the original A1 architecture described in subsection 5.4.1 since those two blocks are the ones that were found to be sensitive to K given that C is kept constant.

The new K-NN architecture is then used to create the DPR implementation based on the DPR methodology described in chapter 3 whereby the KNN core is set as the RP, while the other blocks of the classifier are kept in the static region. Next, RMs are created based on the .NGC files of several implementations of the KNN core reflecting different chosen K values. RMs are basically several copies of the functional block contained in the RP region that will be swapped in and out of the device as shown in Fig. 5.10 to allow the user to choose the RMs associated with the desired K during run-time, without interrupting other processes running on the same chip. RMs are different from each other in the amount of logic inferred since they reflect the same functional block generated with different values of K.

To meet design considerations, the size of the RP region is made large enough to accommodate the logic resources required by the largest K. To demonstrate the effect of changing K on the CLB slice requirement of the RP region, K was increased from 2 to 7 which resulted in CLB slice count to increase from 113 to 405 i.e., by 292 CLB slices; when K was increased to 19, the CLB slice count became 1093. Consequently, it is important to predefine the largest K and know the incurred CLB resources beforehand to be able to account for the required resources when sizing the RP region. However, if the changes in the amount of resources were too large, it may not be possible to cater for such variations on a particular chip making DPR unsuitable for the classification problem in hand. This is mainly because too much FPGA area would be reserved within the RP for the largest possible configuration driving the cost of the implementation considerably high while being needed for particular cases only. As such, the DPR implementation involving wide range of K's must be justified as trade-offs in terms of FPGA area will be involved.

Additionally, another significant design consideration is proper interface matching between the static region and the RP for all possible RMs. Interface mismatch between the static and dynamic regions will lead to implementation failure. In all the DPR implementations presented in this thesis, care was taken to account for this design consideration through introducing dummy I/Os when needed.

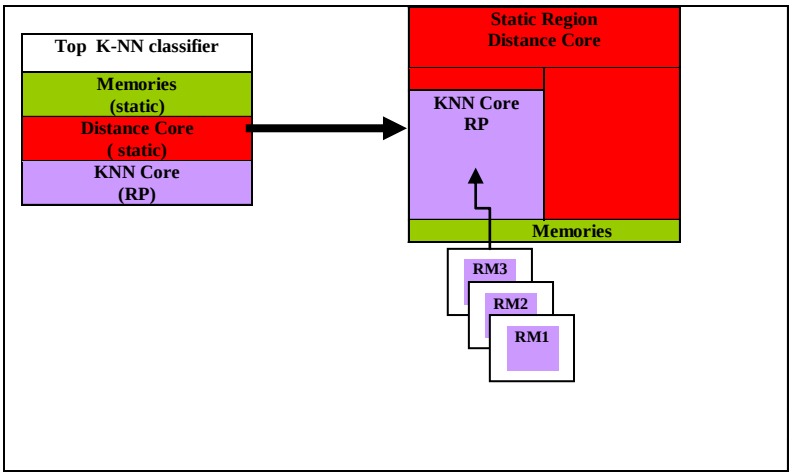


Figure 5.10: *Illustrative diagram of the DPR Implementation of the single-core K-NN classifier based on setting the KNN finder block as RP and creating several RMs to replace the RP.*

Following the definition and verifications of all RMs, several configurations were created reflecting variants of the RP region (RMs based on different K values) with the first configuration taking longer as it creates a full bitstream of the whole implementation consisting of both the static logic (memories and distance core) and the reconfigurable logic (KNN core). After this implementation is completed and the bitstreams (full and partial) are generated, the implementation was promoted so that subsequent implementation using different RMs copy the static logic instead of having to re-run the whole implementation again reducing the overall development time considerably. The implementation is repeated for all possible RMs.

Once all configurations are run successfully, full and partial bitstreams for each configuration become available for each particular configuration reflecting the chosen RM (specific to one value of K). The process is then considered complete and files are ready for actual download to the FPGA. The initial download must be with a full bitstream reflecting a full chip configuration including logic of the static and RP regions. Then, for any desired K one would only download the partial bitstream of the configuration associated with the desired K to reconfigure the RP region without affecting the configuration of the static region. The main advantage of this approach is that reconfiguring a specific portion of the FPGA is faster than reconfiguring the whole device if only a portion of the design need to be altered, or when fault occurs within the RP region which calls for reconfiguring that region only. As a result, configuration time is saved considerably when using DPR in addition to

power saving since specific regions within the FPGA are modified. Furthermore, the operations of other processes running on the same FPGA are left uninterrupted.

5.4.3.2 DPR Implementation of the K-NN Classifier based on Reconfigurable Single-core K-NN classifier.

Instead of dynamically reconfiguring just a specific segment within the K-NN classifier, as in the aforementioned subsection where the KNN core was only reconfigured, the complete classifier is made reconfigurable here just as was done with the K-means core in chapter 4. A new design Wrapper is created which instantiates the original single-K-NN classifier presented in 5.4.1. Then, the DPR methodology and PR flow is followed to set the complete K-NN core as RP rather than setting the KNN core only. Following this, the size of the RP is determined to account for the resources required for the maximum intended change in any of the parameters e.g., K, N, M and C. This implementation offer the flexibility to alter all blocks of the K-NN classifier simultaneously including the contents of the memories, the data dimensions, number of samples, and the number of neighbourhoods. This implementation is suitable when more modifications are needed than just changing K. The advantage of this implementation will be realised when dealing with multi-core classifier as will be described in subsection 5.4.4.

Generally, changing K has been found to be the most appropriate parameter that can benefits from DPR, while changing N or M would benefit the least. Changing N or M causes significant increase in CLB slices leading to trade-offs in area footprint when planning the size of the RP region. However, for small changes in N or M, the DPR implementation will still be feasible given that the RP region is sized according to the largest intended N, M and K.

5.4.4 DPR Implementation of Multi-core K-NN classifier

Multiple K-NN cores based on the single K-NN classifier shown in Fig. 5.4 were used to form a multi-core architecture of the K-NN classifier which allows those cores to run in parallel. Each core runs independent from the others having its own clock and memory. The system is used to partition a large dataset among the different cores thus can achieve significant acceleration over a single core implementation by a factor equal to the number of cores used. In addition, the result of the multi-core classifier can be combined to form an

ensemble K-NN classifier. Alternatively, the multi-core implementation is used as an on-demand server solution to process different queries coming from different users simultaneously where each core is configured or reconfigured according to user entries such as different training datasets or K values.

To enhance the above application and add flexibility to alter configuration of the multi-core architecture at run-time, a DPR implementation is formed based on setting all the cores in the above multi-core architecture as RPs and creating multiple variants of each core corresponding to different K's or different initialised memory contents, those variants are the new RMs. To complete the DPR implementation, all the RPs were constrained within specific regions on the FPGA containing all the necessary logics required by the corresponding RMs. Fig. 5.11 illustrates this process based on a system of three cores, where each RP is having four RMs associated with it. It is crucial that the size of each RP be big enough to contain sufficient number of logic resources required by the RM having the largest possible K, otherwise the process will fail if not enough resources are found to implement the required RM corresponding to the desired K. Lastly, multiple configurations were created corresponding to different combinations of the RMs for each core along with the full and partial bitstreams associated with each configuration that are used to configure/reconfigure the FPGA as illustrated in Fig. 5.12.

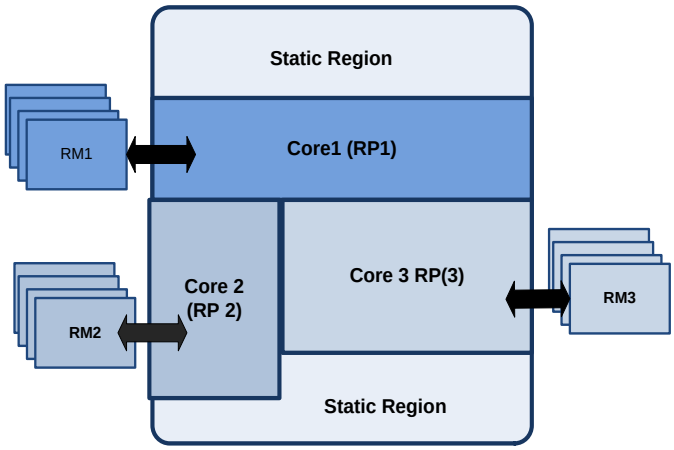


Figure 5.11: *A simplified layout of the DPR implementation of multi-core K-NN classifier based on three reconfigurable corse each having four possible RMs.*

In addition to the fact that DPR offers the flexibility to modify specific cores by partially reconfiguring the FPGA, advantages in terms of reconfiguration time will be discussed when

presenting the implementation results in the subsequent subsections. Note that the results of this implementation will be presented along with the results of the ensemble K-NN classifier presented in the next subsection as the two are similar in the architecture and characteristics except that the ensemble classifier includes an additional circuitry called the voter.

5.4.5 DPR Implementation of Ensemble K-NN Classifier

The multi-core architecture described above is extended to work as an ensemble classifier, where an additional block is added to combine the results from the individual cores forming the architecture of the multi-core, this new block basically performs voting to select the most commonly encountered class label. This additional block utilises small amount of resources based on the value of C and the number of cores constituting the ensemble classifier. Similar to the DPR implementation of the multi-core architecture presented in the previous subsection, all the cores within the ensemble classifier are set as RPs as in Fig. 5.11 and sized according to the maximum K expected to be used within each of the cores. The execution time of the architecture is based on the largest K selected among the cores as modelled in equations (5.5) and (5.6).

$$Execution\ Time = N + M + Max[K] + Ensemble\ Voter\ Time \tag{5.5}$$

$$Ensemble\ Voter\ Time = No.\ of\ K\text{-}NN\ Cores + C \tag{5.6}$$

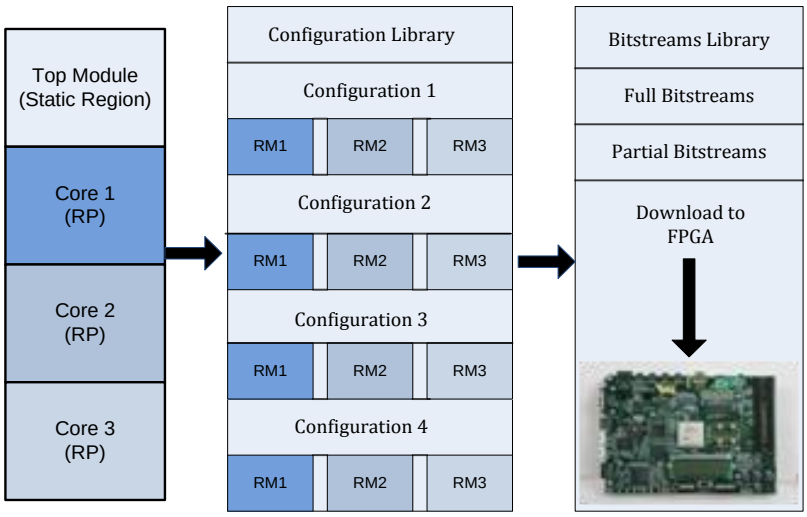


Figure 5.12: *The process of creating a configuration library containing the full and partial bitstreams for each of the three reconfigurable cores used to reconfigure the FPGA.*

5.4.6 DPR Implementation of Ensemble K-NN classifier based on Reconfigurable Memory Block

Given that the architecture of the DPR implementation of the K-NN classifier constitutes of three main cores or blocks, namely, the distance computation, the KNN and the memory block, any of those three blocks can be made reconfigurable. However, reconfiguring the distance computation core was not found to be viable in reducing the reconfiguration time or in adding any extra flexibility as changing this core will require changing the KNN core. The latter is similar to having to reconfigure the complete classifier. As such, using the reconfigurable single-core or multi-core implementations is more useful. On the other hand, investigating the benefits of reconfiguring the third core, namely, the memory block is attempted here.

Based on a ten-core ensemble classifier implemented using a larger FPGA than the one used for the triple-core ensemble implementation, it was possible to create a DPR architecture based on a reconfigurable memory block. The application of such implementation is to be able to change the contents of the memory dynamically without interrupting the operation of some tasks that might be busy processing other datasets from different memory blocks.

5.5 Implementation Results

This section presents the implementation results of all the aforementioned architectures including results from both hardware and software implementations of A1 and A2 architectures of the K-NN classifier. In hardware, synthetic data of different sizes were used for architectures A1 and A2 with chosen dataset that can be stored within the Block RAMs of the available device. The hardware implementation targeted the ML 403 platform board which has a Xilinx' *XC4VFX12* FPGA on it. The bitstreams for both architectures were generated, stored in host, and downloaded to the target device using a JTAG cable. On the other hand, the software implementation on GPP implementation was based on using matlab (R2009b) bioinformatics toolbox running on a 2.60 GHz Pentium Dual-Core E5300, with 3 GB RAM workstation. The reason for using this toolbox is that it includes an optimised K-NN classification function. The following subsections summarise the implementation results.

5.5.1 Single-core Implementation based on A1 Architecture

The test data used were for (B=16, M=16, N=1024, C=4, K=5). As stated earlier, the architecture was captured in Verilog HDL, synthesised, placed and routed using Xilinx' ISE 12.2. The place and route results are shown in Table 5.1:

Table 5.1: Place and Route Synthesis results of Single-core K-NN classifier based on A1 Architecture

Device	Xilinx' *XC4VFX12-10ff668*	
Parameters	(B=16, M=16, N=1024, C=4, K=5)	
	Used/Available	**Utilisation ratio (%)**
Slices	1,005/5,472	18
Slice FF	790/10,944	7
4 input LUTs	1,722/10,944	15
Block RAMs	17/36	47
Clock Frequency	138.882 MHz	

The results of the hardware implementation were thoroughly tested using Xilinx' ChipScope™ Pro Analyser 12.2 and checked against simulation results. The number of clock cycles to classify one query was found to be 1048, achieving an execution time of 7.55 µs based on the attained frequency reported in Table 5.1. On the other hand, the execution time of the GPP implementation was 571 µs based on taking the average of ten thousands runs. Consequently, the FPGA implementation of the A1 architecture outperformed the GPP implementation by approximately 76 times, as summarised in Table 5.2:

Table 5.2: Summary of Timing performance of the A1 architecture

GPP Software (µs)	**FPGA (µs) Based on 138.9 MHz clock speed**	**Speed- up**
571	7.55	~76x

5.5.2 Single-core Implementation based on A2 Architecture

The same tests were carried out for the A2 architecture of the K-NN classifier but with different datasets as using the above N would not fit into the available device. Alternatively,

a dataset having the following parameters was used: (B=16, M=1024, N=16, C=4, K=5). Table 5.3 shows the place and route results of this FPGA implementation.

Similar to the previous implementation, the hardware implementation was thoroughly tested using Xilinx' ChipScope^{TM} Pro Analyser 12.2 and checked against simulation results. The number of clock cycles to classify one query was found to be 1048. The execution time of the FPGA implementation was 5.8 µs based on the attained clock frequency shown in Table 5.3.

Table 5.3: Place and Route Synthesis results of Single-core K-NN classifier based on A2 Architecture

Device	Xilinx' XC4VFX12-10ff668	
Parameters	(B=16, N=16, M=1024, C=4, K=5)	
	Used/Available	**Utilisation ratio (%)**
Slices	2,745/5,472	50
Slice FF	3,376/10,944	30
4 input LUTs	4,811/10,944	43
Block RAMs	15/36	41
Clock Frequency	180.8 MHz	

On the other hand, the execution time of the GPP implementation was 396 µs based on taking the average of ten thousands runs. Consequently, the FPGA implementation of the A2 architecture outperformed the GPP implementation by approximately 68 times, as summarised in Table 5.4. Note that both implementations are based on the same parameters and distance metric.

Table 5.4: Summary of Timing performance of the A2 architecture

GPP Software (µs)	**FPGA (µs) Based on 180.8 MHz clock speed**	**Speed- up**
396	5.8	~68x

5.5.3 Effect of Data Dimensionality: GPP vs. FPGA

The FPGA implementation of the K-NN classifier was compared with equivalent GPP implementation (both based on A1 architecture) running on Matlab (R2009b) bioinformatics toolbox to particularly investigate the effect of changing the number of dimensions M on the

timing performance of the K-NN implementations on the two platforms. When carrying out this comparison, N was fixed at 1024, B at 16, and K at 13, and M was varied from 1 to 16. Fig. 5.13 shows that when M was increased, both implementations took longer times, however FPGA suffered less in classification time as compared with the GPP implementation. For instance, changing M from 5 to 16 features increased the classification time by 1.5% only for the case of the FPGA implementation as compared to 32% for the GPP implementation, and when changing the number of features from 5 to 77 the FPGA classification time increased by 7.3% only as compared to 337% for the GPP implementation.

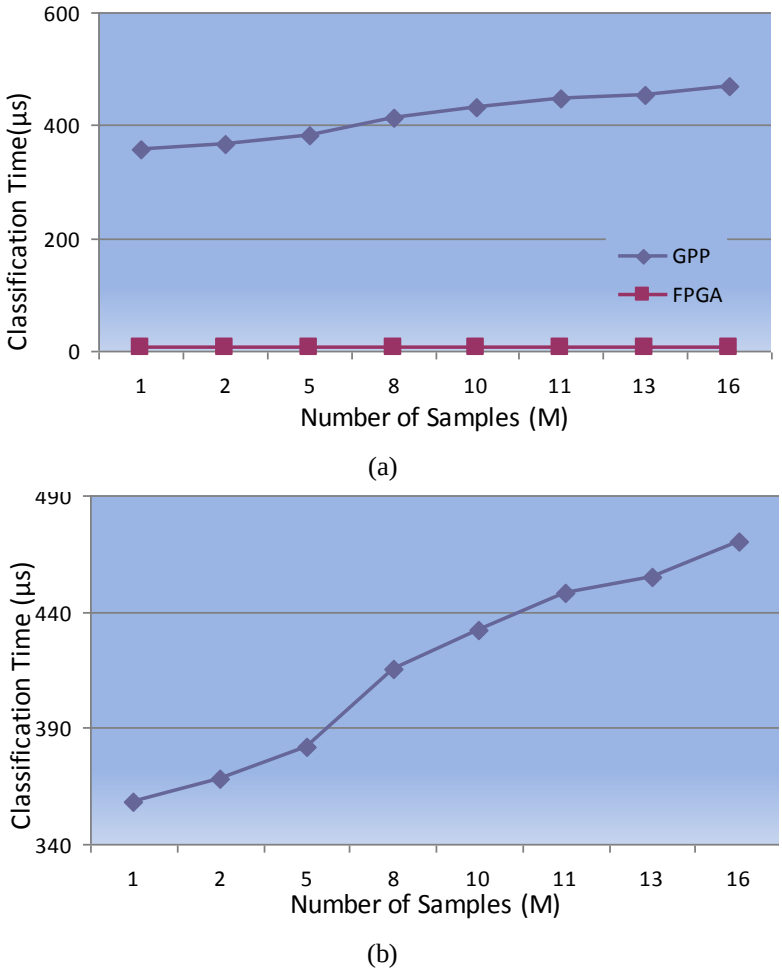


Figure 5.13: *(a) The effect of increasing the dimensions (features) on classification time for the FPGA and GPP implementations of the K-NN classifier, (b) enlarged GPP graph.*

The superior performance of the FPGA implementation is attributed to the high parallelism exploited in the systolic array architecture as opposed to the sequential behaviour

of the GPP implementation. This result shows that FPGA outperforms GPP in terms of timing performance when the dimensionality of data is increased. From examining the classification time in Fig. 5.13(b), it can be stated that the increase in data dimensionality results in significant increase in the classification time of the GPP implementation as compared to negligible increase for the FPGA case as seen in Fig. 5.13 (a). Additionally, the effect of increasing M on hardware utilisation of the K-NN classifier has been investigated, and found to increase linearly as M was increasing as shown in Fig 5.14.

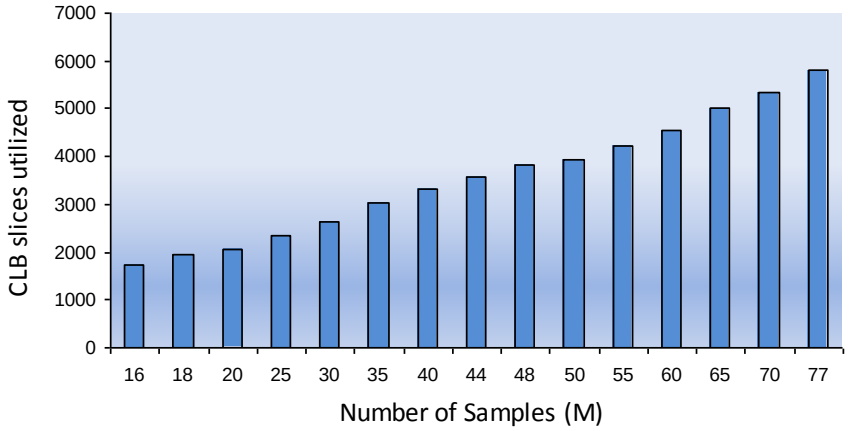


Figure 5.14: *The effect of changing the number of dimensions M on the CLB slices based on A1 classifier having: (B=16, N=1024, C=4 and K=13).*

In summary, it can be stated that FPGA scales better than GPP when the dimensions of data are large mainly due to the extensive pipelining and parallelism employed in the FPGA implementation of the K-NN classifier as compared to pure sequential behaviour in GPP.

5.5.4 Multi-core implementation of the K-NN classifier based on A1 Architecture

The proposed quad-core design was simulated first using synthetic data which mimic Microarray dataset, then synthesised, mapped, placed and routed using Xilinx' ISE 12.2 to target the *XC4VFX12* FPGA. The single core K-NN classifier was configured with the following parameters: (B=16, M=8, N=1024, K=13), the number of clock cycles required to classify a query was found to be 1048 clock cycles leading to execution time of 6.98 µs based on the attained clock frequency. The number of clock cycles was confirmed using Xilinx' ChipScope™ Pro and found to match that obtained from simulation. Then a multi-

core architecture constituting four cores based on K=9 was tested and found to take 1044 clock cycles to classify four queries leading to execution time of 6.82 µs. The resources of the multi-core implementation as compared to the single core architecture are shown in Table 5.5.

Table 5.5*:*Place and Route Synthesis results for the Single and Multi-core FPGA implementation of the KNN classifier

Device	Xilinx' *XC4VFX12-10ff668*			
Parameters	(B=16, M=8, N=1024, C=4, K=13(single-core) and K=9 (quad-core))			
	Used/Available Quad-core	**Utilisation ratio (%)**	**Used/available Single core**	**Utilisation ratio (%)**
Slices	4,733/5,472	86	1,470/5,472	26
Slice FF	3,488/10,944	31	1,124/10,944	10
4 input LUTs	7,045/10,944	64	2,155/10,944	19
Block RAMs	36/36	100	9/36	25
Clock Speed	152.99 MHz		150.320 MHz	

5.5.5 DPR Implementations of partial/Single-core K-NN classifier

The three subsections below present the results of two DPR implementations of the K-NN classifier based on the modified modular architecture having three blocks, memory, distance computation core, and KNN core, where the first subsection will be for an implementation based on A1 architecture while the second for A2 architecture. Both implementations are based on setting the KNN core only as reconfigurable partition (RP). On the other hand, the third subsection will present the results of the reconfigurable single-core implementation.

5.5.5.1 DPR Implementation of A1 architecture based on Reconfigurable KNN core

The DPR implementation of this K-NN classifier was implemented for the case of B=19, N=2048, M=1, C=4, and K was variable. Few K values were randomly selected to define the RMs to be used, choice was based on the fact that in many classification problems related to Microarray data K never exceeded 30, hence a maximum of 19 was selected to simplify the implementation and proof the concept. The selected K's are: 5, 9, 11, 15, and 19. Since K=19 is the largest value, the size of the RP was set to accommodate the KNN core for the case of K is 19 requiring 1094 CLB slices. Five configurations were successfully created and implemented, but three of them were selected to demonstrate the potential time saving in

reconfiguring the FPGA using DPR for this particular case study. These cases are K =5, 11, and 19. Fig. 5.12 referred to in subsection 5.4.4 illustrates the process followed to create the full and partial bitstreams to partially reconfigure the K-NN classifier. To investigate the benefits of this DPR implementation with respect to reconfiguration time, the sizes of the full and partial bitstreams of the three configurations were used to estimate the configuration time based on equation (5.7) for the case of full and partial reconfigurations using JTAG cable. The latter has a bandwidth (BW) of 66 Mbps as declared in [16].

$$ConfigurationTime = \frac{Size\,of\,bitstream}{BW\ of\ Configuration\ Mode} \tag{5.7}$$

The size of the full bitstream was 582 KB for all configurations as it is required for the configuration of the whole FPGA (being *XC4VFX12*), while the partial bitstream was only 150 KB. Applying these figures to equation (5.7) results in full reconfiguration time of ~70.55 ms and partial reconfiguration time of ~18 ms. As such, it can be stated that partially reconfiguring the FPGA is about four times quicker than reconfiguring the whole FPGA. Higher or lower timings could be achieved if the sizes of the bitstreams change or if the configuration mode changes e.g., ICAP has a maximum bandwidth of 3.2 Gbps as mentioned in [16] that is about 48 times higher than JTAG, using ICAP will reduce the reconfiguration time considerably. However, the reconfiguration speed-up is the same for the two modes since it is the ratio of the full reconfiguration time over the partial one. As for the partial bitstream, it will change according to the size of the RP determined based on the maximum chosen K value.

Despite the fact that the reconfiguration times were estimated and that actual measurements are usually associated with some overheads, the presented results are considered somehow reliable in predicting the performance of the DPR implementation with respect to the reconfiguration times as was discussed in chapter 4 subsection 4.6.3.2. Actual measurements obtained by a colleague in the SLIg group at Edinburgh University were compared with the presented estimates and overheads were found to be negligible [19]. The work presented in [19] illustrates an Internal Reconfiguration System (IRS) which is based on using ICAP to dynamically reconfigure the FPGA, the reconfiguration is controlled by the soft processor PicoBlaze, as a case study the author used the K-NN classifier presented in this work and measured the reconfiguration time. Consequently, estimating the

reconfiguration time based on equation (5.7) is considered viable approach for the implementation of the K-NN classifier.

Fig. 5.15 illustrates the floorplan of the three implemented configurations highlighting the area footprint occupied by the reconfigurable KNN core for different values of K, with the case of K=19 showing a fully utilised region while the K=5 and 11 cases showing smaller utilisation. When comparing the footprint of the implementation for the different K's, the KNN core utilised 5% of the resources within the RP when K was 5, 9% for K=9, 11% for K=11, and 15% for k=15. The place and route synthesis results of the DPR implementation are shown in Table 5.6.

Table 5.6: Place and Route Synthesis results for the DPR Implementation of the K-NN classifiers based on A1 Architecture which sets the K-NN core as reconfigurable partition

Device	Xilinx' *XC4VFX12-10ff668*			
Parameters	(B=19, M=1, N=2048, C=4, K=19)			
Blocks	**Slices**	**Slice FF**	**LUT's**	**CLK (MHz)**
Complete KNN	1,206 (22%)	1,304 (11)	2,119 (19%)	116.9
Distance	80 (1%)	19 (1%)	113(1%)	380.6
KNN Core	1,093 (19%)	1,222 (11%)	1,975 (18)	140.7

Note that the maximum clock speed of the DPR implementation was similar to that of the non-DPR. In fact when the clock speed of the DPR implementation of the implementation based on K=5 was compared with an equivalent non-DPR implementation, the maximum clock speed of the former was found to be slightly higher than the latter, with the former being 166.9 MHz as opposed to 121.3 MHz for the latter. The slight improvement in frequency for the DPR implementation is attributed to the fact that the DPR implementation was area constrained leading to more efficient placing and routing. On the other hand, the DPR implementation was found to occupy higher area footprint than the non-DPR, where for the aforementioned case the DPR implementation utilised 9% of the FPGA floor area as compared to 7% for the non-DPR.

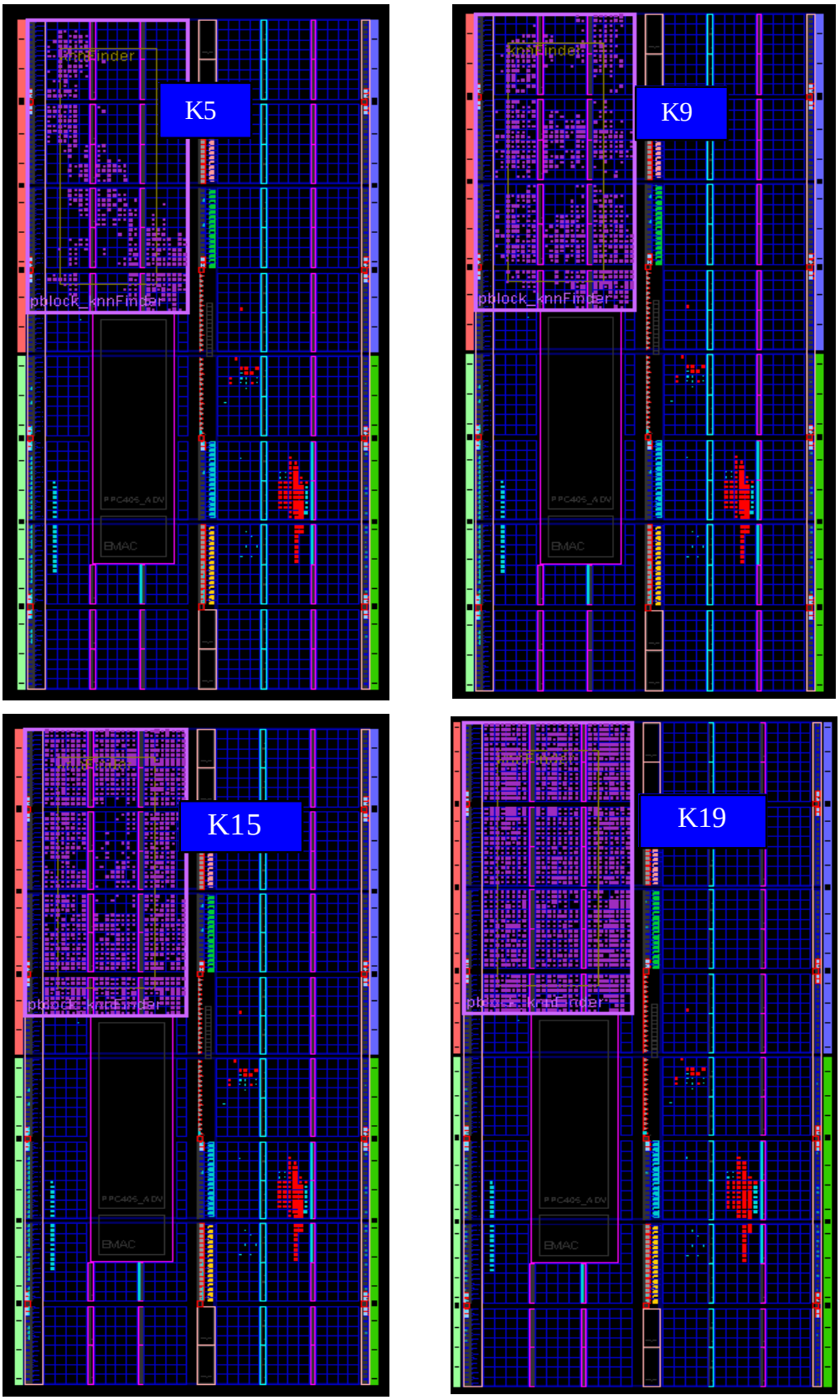


Figure 5.15: *Floorplan of the DPR implementation highlighting the difference in area footprint within the RP regions of four configurations based on using different Ks.*

5.5.5.2 DPR Implementation of A2 Architecture based on Reconfigurable KNN core

Based on A2 architecture, a modular design of three blocks: the memories, KNN core, and distance computation core was used to a create a DPR implementation based on B=16, M=256, N=7, C=4, and K=7, which sets the KNN core as RP and leaves the memories and distance core static similar to the implementation of subsection 5.5.4.1. Fig. 5.16(a) highlights the distribution of CLB slices when no area constraints were imposed and no partition blocks were created (non-DPR implementation) allowing the tool to randomly place the resources. On the other hand, as a result of applying area constraints for the DPR implementation, the static logic gets placed more efficiently and closer to the RP allowing better utilisation of the resources as illustrated in Fig. 5.16(b). Several configurations were created for different K values resulting in the configuration time of the full bitstream to be ~ 70.55 ms and for the partial bitstream to be ~15 ms, leading to a five times speed-up in reconfiguration time of the partial reconfiguration over full reconfiguration. These findings were based on full bitstream of 582 KB and partial bitstream of 121 KB. As for the hardware resources utilised by this implementation, Table 5.7 states the place and route results.

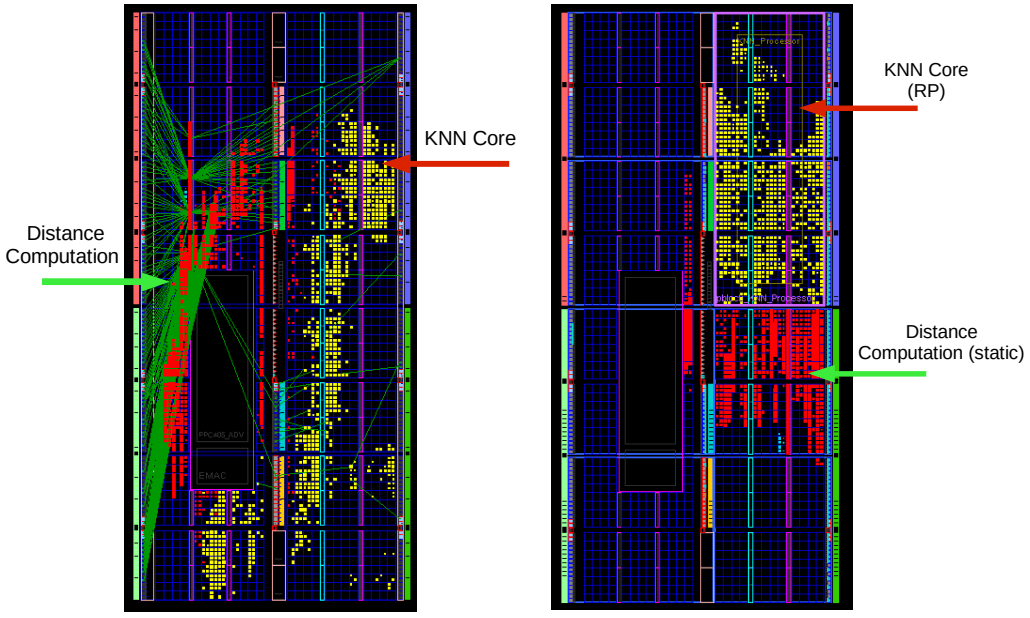


Figure 5.16:*(a) Floorplan of the normal flow implementation showing the area footprint occupied by the two main cores of the K-NN classifier. (b) Floorplan of the DPR implementation based on A2 architecture, with the RP sized for maximum of K=7 while the shown resources were for the case when the RP was configured with K=3.*

Table 5.7: Place and Route Synthesis results for a DPR Implementation of the K-NN classifiers based on A2 Architecture

Device	Xilinx' *XC4VFX12-10ff668*			
Parameters	(B=16, M=256, N=7, C=4, K=3)			
Blocks	**Slices**	**Slice FF**	**LUT's**	**CLK (MHz)**
Complete KNN	1,269 (23%)	1,501 (13%)	2,148 (19%)	195
Distance	452 (8%)	710 (6%)	607 (5%)	234
KNN Core	679 (12%)	754 (6%)	1207 (11%)	197

5.5.5.3 DPR Implementation of the K-NN classifier based on Reconfigurable Single-core

Three-core K-NN classifier was implemented using *XC4VFX12*, with one complete core set as RP while the other two were left static. The three cores are configured for (K1=5, K2=10, and K3=15, B=13, N=2048, M=16, and C=16). The implementation was based on A1 architecture, and on Core 1 being the reconfigurable core as shown in Fig. 5.17. Although, reconfiguring Core 1 can reflect changes in K, C, M and N, the size of the RP region was set to accommodate maximum K of 5, M of 16, C of 16, B of 13 and N of 2048.

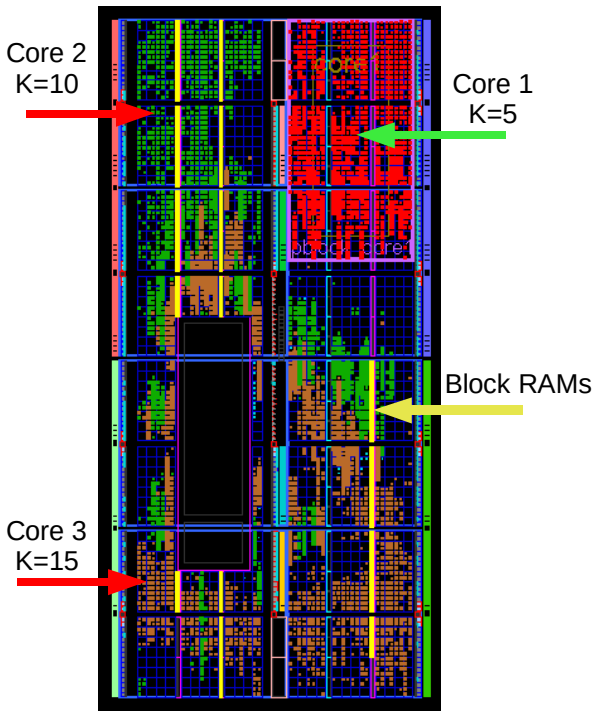


Figure 5.17: *The DPR implementation of a reconfigurable single-core K-NN classifier highlighting the area occupied by each core and the Block RAMs; with Core 1 being the reconfigurable core while the other two are static cores.*

The advantage of this implementation is mainly the flexibility to alter Core 1 by changing K without interrupting the operation of Cores 2 and 3. The advantage of reconfiguration time will be discussed in the following subsections as the implementation of the following subsection is based on having the three cores as RPs, as such the speed-up in reconfiguration time of the subsequent implementation will be similar to this implementation. The implementation consumed 52% of the CLB slices of the targeted FPGA and 69% of the Block RAMs. Note that the memory content is shared among the three cores and as such is part of the static region.

5.5.6 DPR Implementation of Multi-core KNN based on A1 Architecture

This implementation is similar in area footprint and in reconfiguration time to the DPR implementation of the ensemble K-NN classifier which is also based on three cores having an additional circuitry, namely, the voter occupying 22 CLB slices only. Therefore, the reader is referred to subsection 5.5.7 for the results of both: the multi-core and the ensemble K-NN classifiers to avoid repetition as they were found to have similar performance and occupy almost same area footprint.

5.5.7 DPR Implementation of the Ensemble K-NN Classifier based on A1 Architecture

The proposed ensemble classifier combines the classification result of the three K-NN cores which are reconfigurable with different values of K. The parameters used to configure/reconfigure the ensemble classifier are: (B=16, N=1024, M=5, K1=9, K2=7, K3=13), where K1, K2 and K3 reflect the maximum K's of Core 1, Core 2, and Core. The implementation was found to occupy the resources shown in Table 5.8 as compared to a single-core classifier configured for K of 13.

The implementation was tested in a similar way to the multi-core implementation described in 5.5.3, and the execution time was found to be 7.08 µs for this implementation when run as non-DPR. The DPR implementation of the ensemble classifier was based on the three cores being reconfigurable. The PR regions were drawn to accommodate the resources

required by the maximum K value expected for each core which are 9, 7, and 13 for Core 1, Core 2 and Core 3, respectively. The chosen RMs for each core were variant copies of each core configured with equal or smaller values of the maximum K specified for the core. Core 1 was set to accommodate the maximum allowable K value which was 9, the RMs created for this core were for K values of (9, 7, 5, 3), for Core 2 (7, 5, 3) and for Core 3 (13, 11, 9, 7, 5, 3) in addition to a black box core for each of the three cores.

Several configurations were then run constituting variable combinations of the RMs from each core allowing the user to initially select the desired combination and then dynamically replacing any individual core with another one having different K value. Several configurations were created along with the associated full and partial bitstreams for each of configurations permitting the user to swap cores in and out of the device as desired, or disabling one or more cores when they are not needed. Over twelve different configurations were created constituting different combinations of cores having variable K values. Fig. 5.18 illustrates the difference in logic utilisation within the RPs for three implementations corresponding to K9K7K13, K3K3K3, and K7K7K7, while Fig. 5.19(a) illustrate a non-DPR implementation highlighting the distribution of three cores across the FPGA.

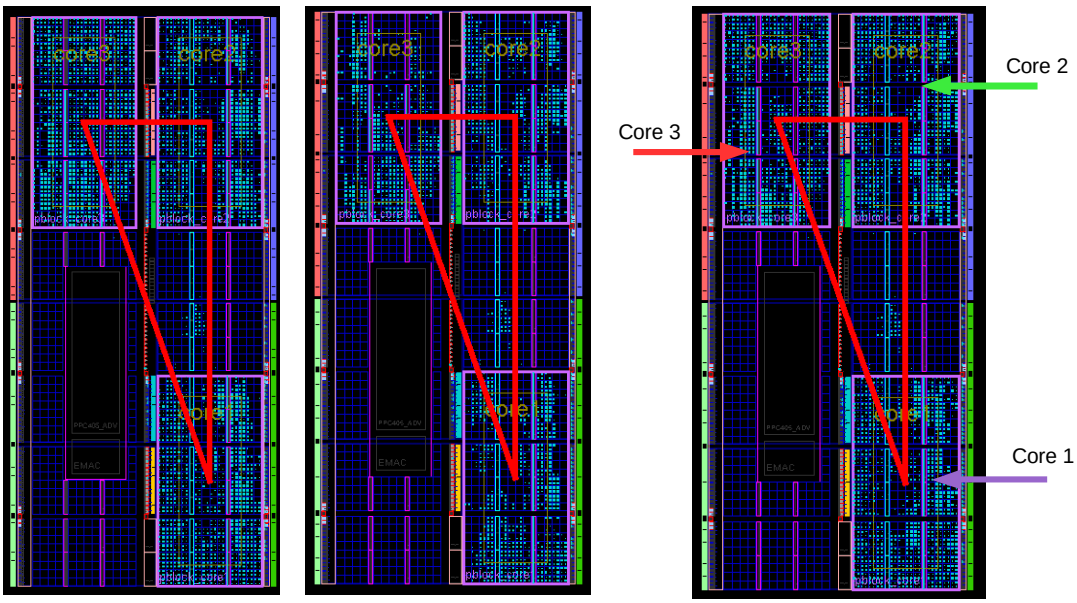
Table 5.8: Place and Route Synthesis results for the Single and Multi-core FPGA implementation of the K-NN classifier

Device	Xilinx' *XC4VFX12-10ff668*			
Parameters	(B=16, M=5, N=1024, C=4, K1=9, K2=7, and K3=13)			
	Used/Available Three cores	**Utilisation ratio (%)**	**Used/available Single core**	**Utilisation ratio (%)**
Slices	2,935/5,472	53	1,377/5,472	25
Slice FF	1,938/10,944	17	1,124/10,944	10
4 input LUTs	4,637/10,944	42	2,155/10,944	19
Block RAMs	18/36	50	9/36	25
Clock Speed	150.23 MHz		150.320 MHz	

From examining the size of the full and partial bitstreams of all configurations, the full bitstream for all configurations was found to be 582 KB in size, and the partial bitstreams were found to have sizes of 121 KB for Cores 1 and 2, and 140 KB for Core 3. The difference in the size of Core 3 as compared with Cores 1 and 2 is mainly due to the fact that Core 3 was set to accommodate a larger K than in Cores 1 and 2. Applying (5.7) based on using JTAG as configuration mode resulted in full reconfiguration time of 70.55 ms, and partial reconfiguration time of 14.67 ms for Core 1 and Core 2, while Core3 was 16.97 ms.

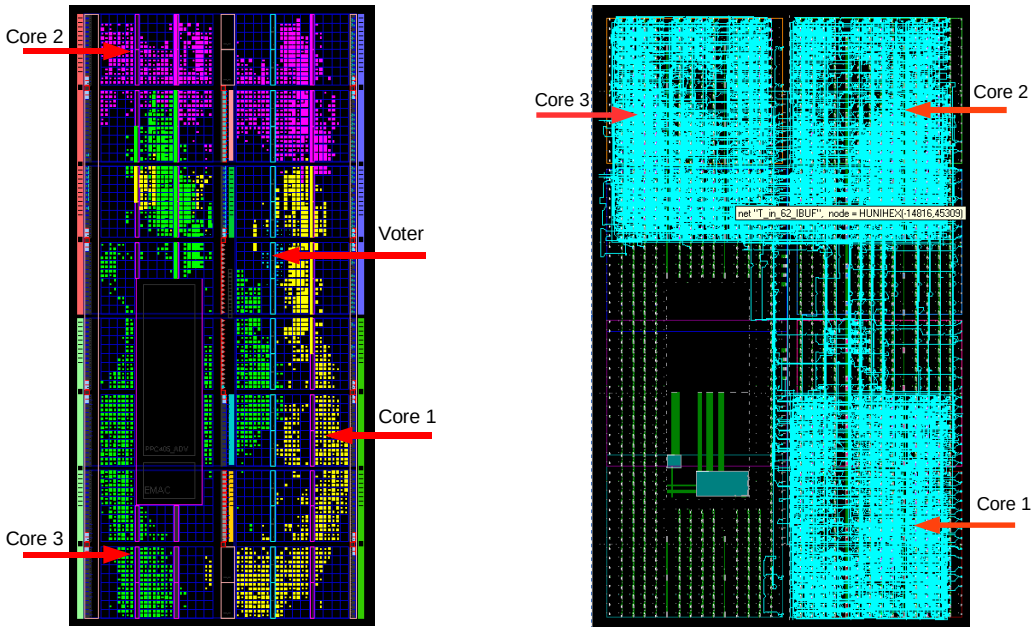
Consequently, it can be stated that for the ensemble K-NN classifier presented in this subsection, partially reconfiguring the FPGA with a core variant is at least four times faster than reconfiguring the whole device; the result is based on using the aforementioned Virtex-4 FPGA. On the other hand, the benefit of DPR in terms of quick reconfiguration time will be considerably higher when larger FPGAs are used as a result of large full configuration time associated with large FPGAs.

Despite the fact that DPR implementation of the K-NN ensemble classifier outperformed non-DPR in terms of reconfiguration time and is estimated to have lower power consumption, this DPR implementation was inferior in terms of area footprint. This was as a result of over sizing the RP regions to enclose enough Block RAMs to store the training set for each core. This issue is clearly due to limited Block RAMs in the targeted device. Fig. 5.19(b) illustrates the routing of the DPR implementation of the ensemble classifier with K=7, emphasising that more of the area in Core 1 not being utilised as a result of being sized to accommodate K=13 whereas the shown routing is for an implementation configured with K=7.



(a) *K9K7K13* (b) *K3k3k3* (c) *K7k7k7*

Figure 5.18: *Floorplan of the ensemble K-NN classifier based on A1 architecture illustrating the difference in area footprint for three different configurations.*



(a) Non-DPR Implementation (b) DPR Implementation

Figure 5.19: *(a) Non-DPR implementation of the K-NN ensemble classifier based on three cores where K1=9, K2=7 and K3=13. (b) PAR image highlighting the routing of the DPR implementation of the K-NN ensemble classifier.*

5.5.8 DPR Implementation of Ensemble K-NN classifier based on Reconfigurable Memory Block

The possibility and benefits of making the memory containing the training set reconfigurable was additionally investigated in this chapter. A multi-core ensemble classifier was implemented to run ten cores simultaneously using the same training set; additionally the memory block was made reconfigurable by setting it as RP. This added feature allows the initialisation of the Block RAMs from the bitstream using DPR rather than using I/Os. This feature can not be used in small devices and works best when the FPGA is large and have abundant Block RAMs, this is due to the difficulty in enclosing large number of Block RAMs associated with typical sizes of Microarray training sets.

Ten cores were fitted in Xilinx*' XC4VLX60* FPGA which were configured with different K values and meant to access the same training set. Fig. 5.20 illustrates the floorplan of the implementation based on the resources shown in Table 5.9. The full bitstream of the implementation was 2163 KB leading to a full reconfiguration time of 262.2 ms based on JTAG as estimated from (5.7). On the other hand, the partial bitstream associated with the reconfigurable memory block was 209 KB leading to a partial reconfiguration time of 25 ms

as estimated from (5.7) based on JTAG. Consequently, it can be stated that for this particular ensemble classifier, partially reconfiguring the memory block of the implementation is ~10x quicker than reconfiguring the whole FPGA. This implementation is useful for cases when only portion of the device memory need to be modified dynamically.

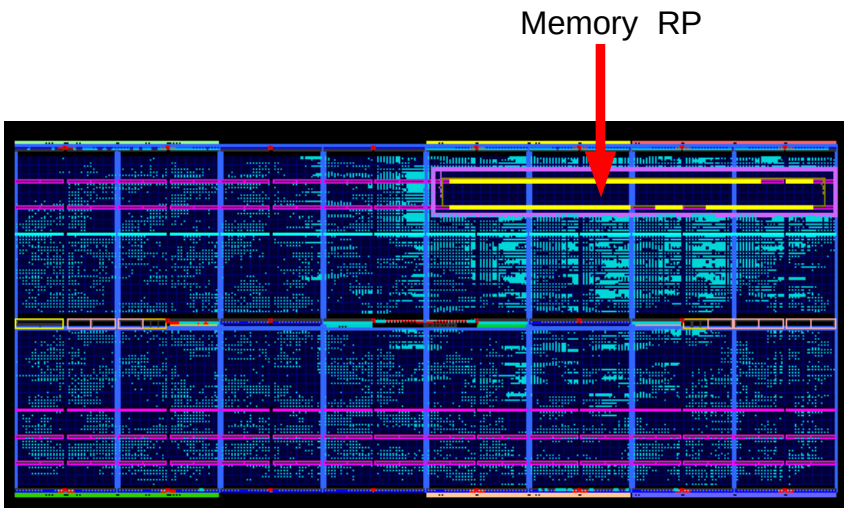


Figure 5.20: Floorplan of the ten-core ensemble implementation highlighting the reconfigurable memory block.

Table 5.9: Place and Route Synthesis results of ten-core implementation based on the following K values: (K1=2, K2=5, K3=8, K4=10, K5=12, K6=15, K7=17, K8=20, K9=25, K10=30)

Device	Xilinx' *XC4VLX60-10ff668*		
Parameters	(B=13, C=2, N=2048, M= 16)		
	Used	**Available**	**% Utilisation**
Slices	13,550	26,624	50
Slice Registers	16,566	53,248	31
LUT's	22,114	53,248	41
Block RAMs	25	160	15
IOB's	231	448	51
Clock Frequency	234 MHz		

5.6 Summary and Conclusions

In this chapter, a total of eight FPGA implementations of the K-NN classification were presented. At first, two different hardware architectures of the K-NN classifier were proposed; each was utilising different level of parallelism and was adaptive to N, M, K, N, and B. The two architectures, namely A1 and A2 offer trade-offs between performance and area, where A1 occupies smaller footprint than A2 while being slightly inferior in terms of performance. The K-NN implementation based on A1 architecture achieved a speed-up of ~76x over an equivalent GPP implementation whereas A2 achieved ~68x. The third implementation was based on multi-core K-NN classifier allowing multiple cores to run in parallel each configured with different value of K to facilitate the implementation of an ensemble K-NN classifier.

In addition, the role of DPR was investigated for the first time through the design and implementation of five different implementations of the K-NN classifier. The first DPR implementation was based on dynamically reconfiguring a specific kernel within the K-NN classifier, namely, the KNN core, while the remaining cores stayed static. This specific implementation investigated the benefits of DPR on the kernels sensitive to changes in the values of K. This was a consequence of K being a user defined variable that gets changed more frequently, as there is no fixed criterion given in the literature for choosing K when applying the K-NN classification, leading users to experiment with different K's. This requirement can be served well by DPR. This implementation achieved speed-up in partial reconfiguration by 4x-5x over full chip reconfiguration while leaving other tasks placed onto to the same FPGA running without interruption.

The second novel implementation was based on reconfiguring the whole K-NN classifier rather than reconfiguring portion of it as offering similar speed-up in partial reconfiguration time to the first implementation using different parameters. However, this DPR implementation can benefit from an additional capability to modify any parameter of the classifier not just the K value, given that the resources enclosed in the RP region are set to accommodate the desired variants.

Additionally, the third and fourth novel DPR implementations of the K-NN classifier were presented in this chapter; one was based on multi-core FPGA implementation while the other was a special case of the multi-core implementation known as the ensemble classifier. The two implementations were at least four times quicker in partial reconfiguration time over full chip reconfiguration. The multi-core offer wide range of applications such as speeding

up the processing of given training set by partitioning the data across the multi cores, another application is parallel processing of multiple queries coming from different users as part of a server solution similar to that explained in chapter four, where cores get activated or modified on demand to cater for different user's requests. Moreover, the ensemble classifier is a novel implementation based on performing voting among classification results coming from multiple cores, this application enhances the accuracy of the classification by combining results from classifiers configured with different values of K.

Over and above, a fifth DPR implementation was presented which sets the memory block as reconfigurable partition offering an advantage to change the contents of the memory by partially reconfiguring the RP region enclosing the Block RAMs serving the K-NN classifier, consequently eliminating I/O limitations and bottlenecks of external memory access. The implementation achieved ~10x speed-up in partial reconfiguration time over full chip reconfiguration. This result was based on using larger FPGA than the one used in the aforementioned DPR implementations emphasising the fact that large FPGAs are due to benefit more from reconfiguration speed-up when compared to small FPGAs.

Future work includes applying self-reconfigurable DPR implementation to the multi-core and ensemble K-NN classifier using hard or soft processors. In addition, the DPR implementation of the ensemble K-NN classifier will be implemented to target real Microarray datasets. Furthermore, future work will investigate the potential benefits of setting the distance kernel within the K-NN classifier as a reconfigurable partition to configure the classifier with different distance metrics such as Euclidean, Manhattan, or cosine.

Chapter-6

Hardware Implementation of SVM Classification on FPGA

6 Hardware Implementation of the Support Vector Machines Classification on FPGA (SVM)

6.1 Introduction

Support Vector machine (SVM) is one of the most recent class of supervised classifiers, and one that has been gaining wide acceptance in recent years [7]. Since its emergence in the early 1990s, SVM has been used extensively in a vast number of applications such as text categorisation, image recognition, and bioinformatics [92]. Bioinformatics applications use SVM in protein homology detection and gene expression. In the latter, SVM is used for the molecular classification such as assigning functions to genes, or in automating the diagnosis and prediction of cancer tissues [93]-[94]. In many Microarray studies, SVM has shown superior classification performance when compared with other supervised classifiers, mainly due to its amenability to high dimensionality, flexibility in choosing a similarity function and ability to identify outliers [95]-[96].

On the other hand, the large dimensionality of Microarray data impose high computational demands on training and classification tasks involved in SVM which impedes the full exploitation of Microarray data in formulating complex biological studies. In an effort to counteract the computational limitations of current GPPs applied to the analysis of Microarray data, a hardware implementation of the SVM classification decision function is proposed in this chapter using state-of-the-art FPGAs. Additionally, the role of DPR applied to the proposed SVM implementation is investigated. Accordingly, a collection of various SVM architectures is constructed using combinations of non-DPR and DPR-based SVM cores.

The remainder of this chapter will present a background on the SVM classification followed by an overview of prior work on the area of hardware implementation of the SVM classifier. A total of six FPGA implementations of the SVM classifier will be introduced in this chapter, three of which are based on harnessing DPR technology. First, two variable FPGA architectures of the SVM classifier will be presented, namely, A1 and A2 including the details of the various blocks constituting the two architectures. Second, a multi-core architecture based on combining multiple SVM classifiers will be presented. Third, a novel DPR implementation of the SVM classifier will be presented based on a reconfigurable

single-core. Fourth, a novel DPR implementation of a multi-core SVM classifier will be presented based on dynamically reconfigurable quad cores. Then, a novel DPR implementation based on dynamically reconfiguring a single-core classifier as either a K-NN or SVM classifier will be presented. This implementation is referred to in this thesis as the multi-classifier architecture. Following this, the results of the aforementioned implementations will be presented and discussed including the advantages gained from each of the proposed architectures. Finally, a summary and conclusion of this chapter will be laid out along with plans for future work.

6.2 Background on SVM Classification

Support vector machine (SVM) is a class of supervised machine learning methods that is based on projecting data from an input space to a large feature space and then separating features of two class labels using a hyperplane. The latter is constructed using a kernel function. Data are projected to a larger feature space because of the difficulty associated with separating data in the original input space using a hyperplane.

The SVM classification consists of two discreet phases; one is the training phase while the other is the evaluation of the decision function known as the classification phase. During the training phase, SVM estimates a function which classifies the data into two classes, by forming a hyperplane that maximises the separation of the two classes [95]. SVM deals mainly with problems of binary classes (class label= 1 or class label= -1), and when multi-class problems are used, the SVM is performed on two classes' at a time until all classes are covered. During the training phase, data are mapped to a large feature space as mentioned earlier where the classifier tries to estimate a multivariate function from a given training set that can separate the two classes by constructing a hyperplane that maximises this separation as shown in Fig 6.1 [93]. SVM is characterised as being able to operate in a large feature space that is not explicitly defined, and of being able to address the issue of non-separable data by constructing soft margins allowing some points to be misclassified for a given penalty.

Given a training set (x_i,y_i), where i=0 to N-1 (N is the number of training samples), $x_i \subseteq \Re^M$ are the training features, M is the number of features or dimensions, and $y_i \in \{-1,1\}$ being the known classifications of the i training samples. The classification function of linearly separable training data is given by equation (6.1) [92]:

$$f(x) = \langle w.x \rangle + b, \tag{6.1}$$

where b is the bias or distance between the hyperplane and the origin as shown in Fig. 6.1 and **w** is a normal vector of the separating hyperplane. The hyperplane seeks the maximisation of the distance between the two soft margins through the minimisation of the norm **w** for linearly separable training data. However, since minimising **w** involves

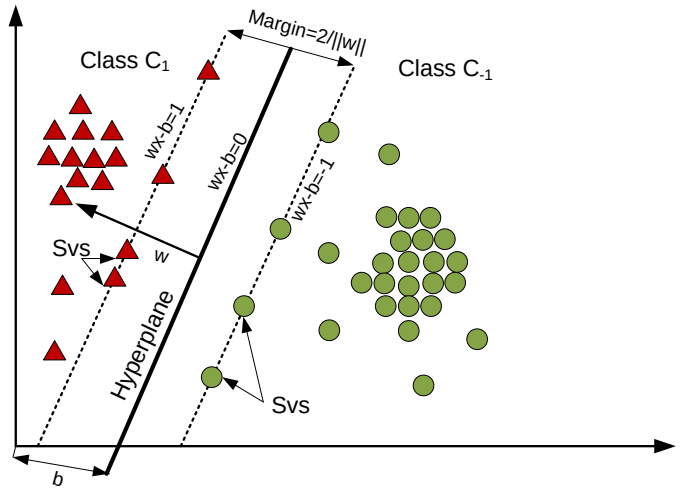


Figure 6.1: *Illustrative diagram of the support vector machines (SVM) describing the separating hyperplane for the two classes and the associated soft margins, the points laying on the margins are the most difficult to separate training points, and referred to as the support vectors.*

minimising ||w|| which involves dealing with a square root operation, the minimisation of **w** leads to the optimisation given by equation (6.2).

$$min \frac{\|w\|^2}{2} \text{ ,subject to } y_i(w.x_i - b) \geq 1 \text{ for } i = 0 \text{ to } N-1 \tag{6.2}$$

The solution to (6.2) is given by the saddle point of the Lagrangian function as shown in equation (6.3) [95]:

$$L(w,b,\alpha) = \frac{\|w\|^2}{2} - \sum_{i=0}^{N-1} \alpha_i \{[(x_i.w) - b] y_i - 1\}, \tag{6.3}$$

where α_i's are the Lagrange multipliers. L has to be minimised with respect to w and b, and maximised with respect to $\alpha_i > 0$. Based on the Kühn-Tucker theorem, which implies that for an optimum hyperplane, α_i must be ≥ 0, **w** can be expressed as in equation (6.4) for linear SVM.

$$\mathbf{w} = \sum_{i=0}^{N-1} y_i \alpha_i x_i \tag{6.4}$$

Substituting (6.4) into (6.3) and applying the associated constrains leads to the dual formulation given in equation (6.5).

$$L(\alpha) = \sum_{i=0}^{N-1} \alpha_i - \frac{1}{2}\left[\sum_{i,j=0}^{N-1} \alpha_i \alpha_j y_i y_j (x_i . x_j)\right], \; given \sum_{i=0}^{N-1} \alpha_i y_i = 0, and \; \alpha_i \geq 0 \tag{6.5}$$

The bias b is set to zero here assuming that the hyperplane is passing through the origin. For non-linearly separable data, the dot product $x_i.x_j$ in equation (6.5) can be replaced with an alternative function, leading to the kernel notation shown in equation (6.6):

$$L(\alpha) = \sum_{i=0}^{N-1} \alpha_i - \frac{1}{2}\left[\sum_{i,j=0}^{N-1} \alpha_i \alpha_j y_i y_j K(x_i, x_j)\right], \tag{6.6}$$

where K(.) is the kernel function which can be linear, Gaussian, or polynomial as stated in equation (6.7), respectively:

$$\begin{aligned} k(x_i, x_j) &= x_i . x_j \\ k(x_i, x_j) &= e^{\left(\|x_i - x_j\|^2 / 2\sigma^2\right)} \\ k(x_i, x_j) &= (1 + x_i . x_j)^p . \end{aligned} \tag{6.7}$$

The linear SVM classifier is considered for the hardware implementation proposed in this chapter; consequently the linear kernel function is used leading to the transformation of (6.1) to equation (6.8).

$$f(x)=\sum_{i=0}^{N-1} y_i \alpha_i K(x_i, x) + b, where\ k(x_i, x_j) = x_i . x_j \tag{6.8}$$

For simplifying the hardware implementation, the bias b can be set to zero assuming that the hyperplane is passing through the origin, and the query x_j will be alternatively labelled as Q_j during the classification phase to distinguish it from the support vectors x_i obtained during the training phase. Note that the x_i used in the training phase represents the complete training samples, whereas x_i used during the classification phase represents the support vectors (SVs). The latter forms a subset of the training samples having non-zero α_i's. During classification phase, when the SVM classifier is presented with a query vector Q, it performs the classification function defined in equation (6.9) based on the linear kernel (dot product) to try to determine in what side of the hyperplane the query lies as illustrated in equations (6.9) to (6.11) [95]:

$$Query\ Class(Q) = sgn\left(\sum_{i=0}^{N-1} y_i \alpha_i x_i^T Q\right), \tag{6.9}$$

$$Query\ Class(Q) \geq 0 \Rightarrow Q \in C_1, \tag{6.10}$$

$$Query\ Class(Q) < 0 \Rightarrow Q \in C_{-1}, \tag{6.11}$$

where C_1 and C_{-1} are the class labels associated with y_i as shown in Fig. 6.1. In this work, equations (6.9)-(6.11) form the basis of the hardware architecture presented in this chapter, where the FPGA implementation tries to solve equation (6.9) given that the training phase is done off-line, and that the hardware design is supplied readily with the trained subset of the training set corresponding to the support vectors (SVs) having non-zero coefficients.

6.3 Prior Work on FPGA Implementation of the SVM Classification

SVM has established itself as superior supervised classification method in a wide range of applications [92]-[96]. As a result of its popularity, many efforts have been expended toward the acceleration of SVM and the enhancement of its real time performance using FPGAs. FPGA implementations of SVM are directed toward three main areas, namely: accelerating the training phase, the classification phase, or accelerating both in a single architecture. The

following overview is a selection of relevant and most recent published efforts in the FPGA implementation of SVMs.

The earliest work reported in the literature on FPGA implementation of SVM training was in [97], targeting non-linear classification. The authors proposed and implemented a digital architecture of SVM in FPGA targeting the learning or training phase only. The architecture consists of two main parts, the first solves the constrained quadratic problem (CQP) given an initial constant bias (b), while the second iteratively updates this bias. At the end of the learning phase, the architecture returns the bias and the coefficients. The proposed architecture is kernel based, which maps the training data to an infinite feature space using the Gaussian kernel (non-linear), and then solves an optimisation problem to obtain the support vectors and their coefficients. Additionally, the authors have evaluated the quantisation effect on the parameters of the SVM, the error associated with the fixed-point representation and compared it with equivalent floating point concluding that the fixed implementation yields acceptable accuracy. Finally, the authors have tested the architecture on Xilinx Virtex-II for the case of 8 and 32 patterns achieving acceptable rate of classification for both [97]. The work did not include acceleration results with respect to GPP and it was mainly focused on proving the suitability of the application to hardware implementation.

The authors of [97] have various subsequent works in this area, one of their recent works was reported in [98]. They presented an FPGA core generator tool aims for automatically generating Gaussian kernel SVM architecture in VHDL based on several entries i.e., user requirements, relevant device characteristics and constrains. The user basically enters the desired parameters to the graphical user interface (GUI), the number of support vectors, number of features, data rate, and the gamma variable required for the computation of the Gaussian kernel. Additionally, the GUI requires the user to enter the precision of the input data, supply the support vectors and their coefficients'. The core then customises the hardware description accordingly to solve the classification function given in equation (6.12).

$$f(x)=\sum_{i=0}^{N-1} y_i \beta_i K(x_i, q), where\, K(x_i, q) = 2^{-\gamma \left\| x_i - q \right\|_1}, \tag{6.12}$$

where x_i is the support vectors, N is the number of support vectors, q is the query, β is a non-zero coefficient of the support vectors, and γ is a constant. The SVM architecture consisted of three main parts, the first is to compute the Manhattan distance, which is the most

computationally intensive part in the whole algorithm and has been extensively parallelised by the core. The second part receives the Manhattan distances and use them to compute the kernel $K(x_i,q)$. Finally the third part makes the classification decision according to the sign of f(x). The software tool was made available online by the authors, therefore, it was downloaded and tested to customise various SVM cores demonstrating a quick and easy to use tool. Apparently, the tool was designed to implement the SVM classification phase only assuming that training was done off-line, and realises the Gaussian kernel only based on Manhattan distances [98].

In [99], the authors reported an SVM architecture that performs the training phase based on sequential Minimal Optimization (SMO) for solving the CQP problem analytically rather than numerically. The main contribution of the work was to implement the SMO-SVM using DPR, whereby the modular blocks performing the tasks associated with SVM training were time multiplexed leading to an area saving of 22.38% of the design implemented in Xilinx' Virtex-4 *XC4VLX25* FPGA.

In [100], the authors reported a hardware implementation of the SVM classifier which performs both training and classification on FPGA based on three types of kernels: linear, Gaussian, and polynomial, using recursive-updating equation instead of solving the CQP. The architectures targeted disease diagnosis based on using Microarray data, which is the same scope of the work presented in this chapter. The main components of the architecture shown in Fig. 6.2 adopted from [100] are: a Learning Element (LE) for generating the final coefficients and the bias data to be supplied to the Support Vector Elements (SVEs) at the start of the classification phase; a control module; and a decision making module which determines the class label of the query. The architecture operates in four distinct modes: loading, kernel computation, training and classification which are overseen by the control module. The authors have tested their architecture on sonar and cancer data achieving superior classification performance in terms of classification, accuracy especially with the linear kernel. Additionally, the authors have reported equivalent performance to floating-point implementation when classifying the sonar data based on input and address WLs of 22 and 16 bits, respectively. As with the Leukemia cancer data that were characterised by 3571 (M) genes, 72 patterns (N) (38 used for training and 24 for testing), the authors reported one misclassification for the case when input WL was 14 bits and the kernel scaled down by 256. On the other hand, when 20 bits were used for the input WL, one misclassification was reached when the kernel was scaled down by 4. The main drawback of this architecture is its area footprint, where each SVE contained three multipliers, therefore testing the Leukemia

dataset on Xilinx' Virtex-II *XC2V8000* FPGA required 40 SVEs. This led to the consumption of 120 multipliers, which is an enormous amount. The authors have not reported the total CLB slices consumed by the architecture. On the other hand they have reported that the architecture attained a clock speed of 25 MHz for the Leukaemia implementation.

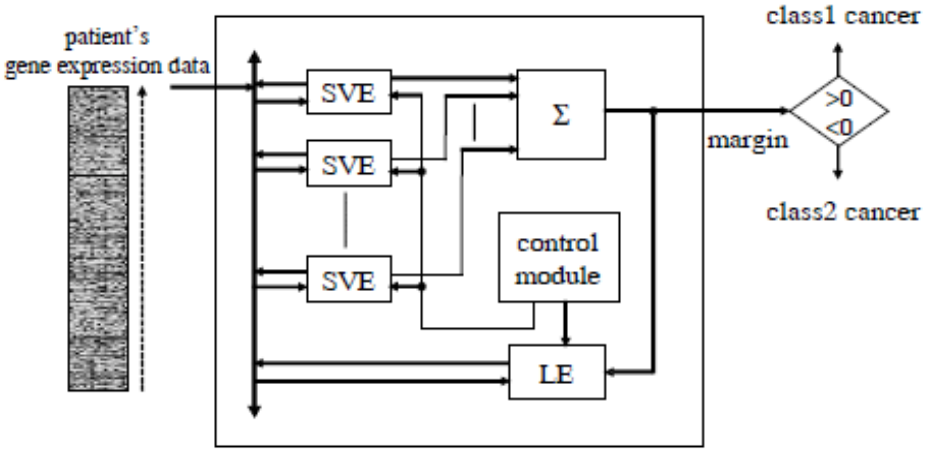


Figure 6.2: *The SVM architecture presented in [100].*

In [101], the authors presented an FPGA implementation of the SVM classifier targeting brain computer interface requiring real-time decision, and assumes that training is done off-line using linear kernel. The architecture realised the classification decision function based on parallelising the computation of the linear kernel. The architecture was based on processing six dimensions in parallel using the embedded 18 × 18 multipliers available on Xilinx' Virtex-II *XC2V1000-4* FPGA. The architecture was used for validating the design using two datasets. The first was based on 1000 SVs of 6-dimensions while the other based on classifying MRI data of 256 × 256 pixels of 3-dimensions (the resources of the remaining three dimensions were left un-used). Training was performed off-line using LibSVM Matlab extension. The architecture performed well in terms of classification when compared with a floating point implementation. However comparing the FPGA implementation of the MRI case which achieved 50 MHz in clock speed with an equivalent implementation (written in C) running on 550 MHz AMD Athlon GPP, showed that FPGA performed worse in terms of processing speed consuming twice more time than GPP. In addition, the FPGA architecture was non-scalable limiting the implementation to six dimensions only [101].

Another FPGA implementation performing the training phase for SVM based on Gilbert's algorithm was presented in [102]. The implementation was based on a scalable systolic array architecture performing linear SVMs, which achieved a speed-up of three orders of magnitude over an equivalent Matlab implementation running on 3 GHz Intel Core 2 Duo and 2 GB RAM GPP [102].

A coprocessor Implementation of SVM training and classification was reported in [103] based on Xilinx' Virtex-5 *LX330T* achieving speed-up of 20x over a GPP implementation running on 2.2 GHz GPP. The implementation was based on partitioning SVM-SMO classifier between a host and FPGA, whereby the host performs the training and supplies the results to the FPGA which computes the kernel dot product using low precision. Once finished, the FPGA sends the results back to the host.

As a consequence of the widespread use of SVM in various applications, more recent work has been reported in the literature on several high performance FPGA implementations, the reader is advised to consult [104]-[108] for details about such additional work on FPGA implementation of SVM. In addition to being popular algorithm for hardware implementation, SVM has been incorporated in many data mining software suites such as the LIBSVM tool, Matlab, R-statistical package, SVM Fu, SVM Torch, and many others.

Although in [99] and [108], the authors have presented FPGA implementations of the SVM classification which have employed DPR technology, where in [99] a coarse-grained time-multiplexed implementation demonstrated an area saving by ~22%; and in [108] a difference based partial reconfiguration achieved power reduction by 3-5%; in this chapter three novel DPR implementations of the SVM classifier are presented and detailed offering high performance particularly suited to server solutions targeting Microarray data.

6.4 Novel Hardware Implementation of the SVM classifier on FPGAs

6.4.1 Single-core SVM Classifier Architecture

Two proposed architectures of the SVM classifier based on systolic arrays architectures similar to A1 and A2 architectures of the K-NN classifier are presented, both solve equation (6.9). The two architectures constitute a modular design captured in Verilog HDL consisting

of the blocks shown in Fig. 6.3. The first block is the memory, which is responsible for storing the training data. The second block is responsible for the computation of the linear kernel shown in equation (6.9) using the training data, class labels and coefficients received from the local memory. The third block is responsible for accumulating the results coming from the kernel computation block as they get computed. Finally, the decision making block receives the result from the accumulation block when all data in the training set have already been processed to determine the class label of the query based on the sign of the accumulation result. The following subsections will provide more details on each block for the two proposed architectures.

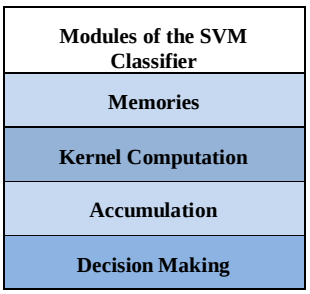


Figure 6.3: *The main blocks of the SVM classifier.*

6.4.1.1 Memory Block

A) A1 Architecture (M>>SVs)

This block stores four types of data describing the training set, thus comprising of four memory sub-blocks as shown in Fig. 6.4(a). The first memory sub-block stores the complete training set in the form of a matrix of a number of rows equal to the number of support vectors (SVs) and a number of columns equal to the number of features or dimensions (M). These are implemented as a set of FIFOs of a number equal to the number of SVs having a depth of M each, and a width of B (WL of each feature). The second memory sub-block is a FIFO used to store the class labels of the SVs. When these SVs are small, the associated labels are stored in registers instead of wasting a complete Block RAM due to the fact that each class label requires one bit only (to represent binary class labels). Since M is expected to be significantly large for this architecture, the third memory sub-block is used as a FIFO to store the features of the query. Note that multiple queries can be stores in this memory if enough block RAMs are available. As for the fourth memory sub-block, it is used to store

the training coefficients (α_i's) computed offline during the training phase, which has a depth equal in number to the SVs. Similar to the class label's memory, the synthesis tool is allowed to infer the type of storage automatically based on the number of SVs to avoid using additional Block RAM resources while small distributed RAM or registers can hold the class labels.

The complete block is parameterisable in terms of parameters B, M, and SVs. Each of the SV-FIFOs instantiated by the core design to store the training set is allocated to one of the Kernel Processing Elements (Kernel PEs), to supply each Kernel PE with one feature every clock cycle in a pipelined manner. Additionally, one feature of the query FIFO is read by the first Kernel PE every clock cycle and propagated through the pipeline allowing for parallel SV kernel computations. On the other hand, the class labels are read from the memory every clock cycle after a latency of M clock cycles needed to fetch the first kernel result. Similarly with the training coefficients, they are read after M clock cycles, as they are required at the same time the class labels are needed for the completion of the kernel computation (as will be illustrated in the subsequent subsections).

The architecture of the memory storing the training set is the most sophisticated part within the memory block as it involves adaptive techniques to instantiate multiple FIFOs each responsible for storing the whole dimensions of one SV (1 SV is a vector of M features). As such, the complete number of FIFOs is equal to the number of SVs. Since these FIFOs are designed to feed the kernel computation systolic array, the read address from those FIFOs is pipelined throughout the SV FIFOs to ensure that data are read by the corresponding kernel PEs on a timely manner. This functionality has been achieved by placing a shift register in front of the read address port of each FIFO starting from the second FIFO onward to achieve the synchronised pipelining required by the Kernel computation as illustrated in Fig 6.4(b). The interconnectivity between the FIFOs to serve this purpose has been automated using appropriate HDL coding style based on the initial parameters entered by the user comprising B, M, and SVs.

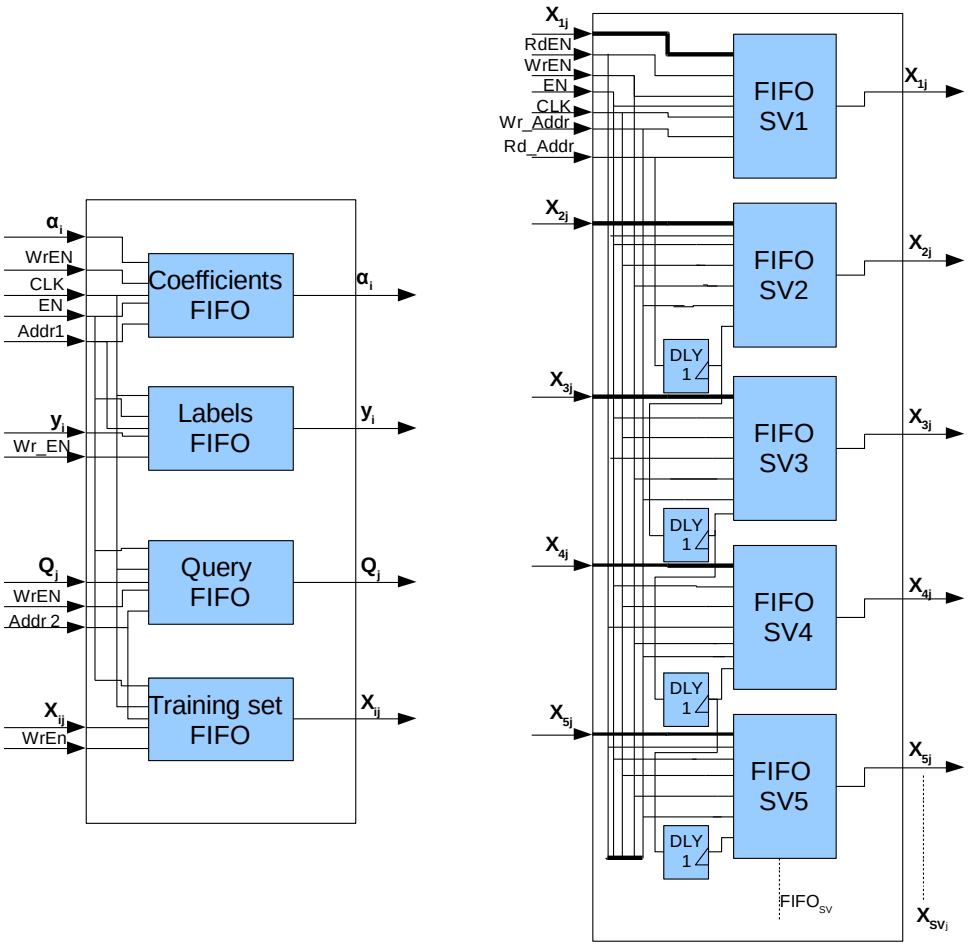


Figure 6.4: *The datapath of the memory block of the SVM classifier:(a) illustrates the components constituting the memory block highlighting that there are four different storage sub-blocks to service the SVM classifier, and (b) illustrates the components of the Memory responsible for storing the training set in the SVM architecture A1 highlighting that the block comprises multiple FIFO's of number equal to the number of SVs each having a depth of M, and that the read address is delayed by one clock cycle throughout the pipeline to achieve the required synchronisation.*

B) A2 Architecture (SVs>>M)

The components and functions of the memory block are similar to that of A1 architecture. However, the specifications, arrangement, and number of FIFOs are different since they serve the case when the number of SVs is much higher than M. In A2 architecture, the training coefficients are stored in FIFO having a depth equal to the number of SVs, and width equal to the WL of each coefficient. As for the class labels and query memory, the tool is left to decide whether to store the data inside registers or utilise Block RAMs based on the preset M and SVs. Memory handling the training data consists of M-FIFOs each having a depth of SV, where each FIFO is associated with a specific kernel PE (there are M kernel

PEs), and are pipelined similar to A1 architecture to serve the requirement of the kernel computation block.

In summary, the size of the memory used to store the training set in both architectures is based on the wordlength of each feature, the total number of support vectors (SVs), and the number of features (M). The depth of the training set memory in each of the two architectures is different: in A1, the depth is M, while in A2, it is SV. On the other hand, the number of FIFOs to store the training set for A1 and A2 is SV and M, respectively.

6.4.1.2 Kernel Computation Block

A) A1 Architecture (M>>SVs)

The kernel computation block is partitioned into three sub-blocks operating in two stages as shown in Fig 6.5, whereby each sub-block is pipelined to perform portion of the computation.

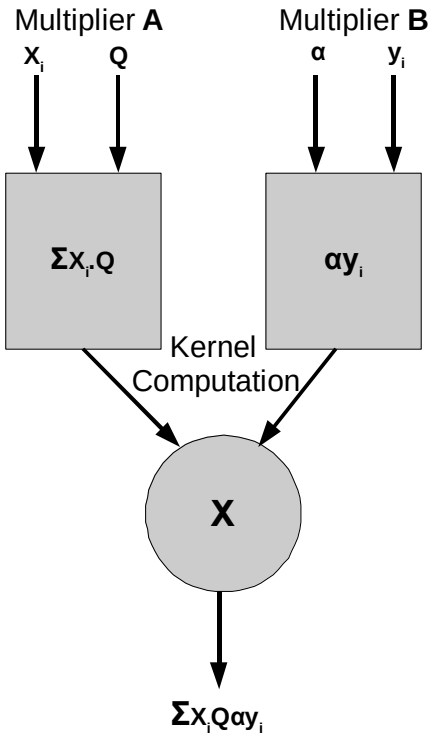


Figure 6.5: *Datapath of the kernel computation block illustrating the operations of the two stages involved in computing the kernel product.*

In the first stage, the largest and most time consuming work is carried out by the sub-block referred to as Multiplier A, computing the dot product (linear kernel) shown in equation (6.13):

$$Multiplier\ A = \sum_{j=0}^{M-1} x_j Q_j , \tag{6.13}$$

where x_j is a support vector (SV) feature, Q_j is the corresponding query feature. This sub-block consists of a systolic array of a number of SV kernel PEs where each PE has the role of receiving one SV feature every clock cycle (x_{ij}) along with the corresponding query feature (Q_j). While each PE has a local FIFO associated with it whose sole responsibility is to provide that particular PE with one SV feature every clock cycle, only one Query FIFO is commonly used by all PEs as explained in subsection 6.4.1.1 (A). The architecture of the systolic array of this sub-block is shown in Fig. 6.6(a). The first PE in the array reads a query feature every clock cycle from the Query FIFO and propagates that feature throughout the pipeline. Fig. 6.6(b) illustrates the functionality of each PE, where each PE processes all the features of one SV independent from each other, except for the pipelining of the query features.

The systolic array is fully parallelised such that the SV computations of equation (6.13) are carried out simultaneously. This operation is facilitated by the capability to obtain the needed feedstock for each PE continuously from the local memory attached to each PE. The latency of the pipeline is M clock cycles, while the throughput is one result per clock cycle corresponding to the multiplication shown in equation (6.13) for one SV. Consequently, for processing one query vector, M+SV clock cycles are required by the pipeline to finish the computation.

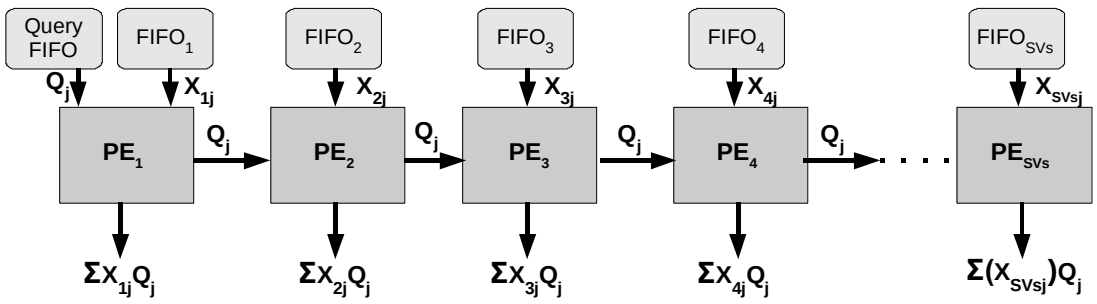
Additionally, while the above computations are being carried out by the first sub-block, the second sub-block, referred to as Multiplier B, which is associated with the first stage of the kernel computation block starts operating just after a period of M-1 clock cycles. This delay is required to ensure appropriate synchronisation with the previous and subsequent classifier operations, and ensure efficient propagation of the data throughout the pipeline. The function carried out by this sub-block reads the training coefficients and the class labels associated with each SV simultaneously from the Memory block to compute the scalar product shown in equation (6.14):

$$Multiplier\,B = \alpha_i y_i\,, \tag{6.14}$$

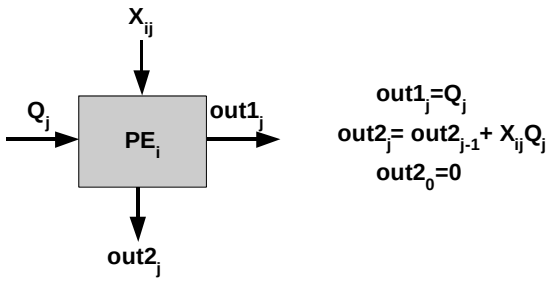
where i is from 0 to N-1. To complete the computation of the kernel for a single SV, the sub-block in stage two multiplies the two results of stage one (results of Multipliers A and B) to obtain the final kernel results given by equation (6.15).

$$Kernel\,Computation = \sum_{j=0}^{M-1} X_{ij} Q_j \alpha_i y_i \tag{6.15}$$

It can be stated that the last multiplier is pipelined with the previous two such that it starts working after M clock cycles (the latency of the first sub-block) thereafter it outputs one result per clock cycle.



(a)



(b)

Figure 6.6: *(a)The systolic array of "Multiplier A1" of the kernel computation block illustrating the vector multiplication array $\Sigma x_{ij}Q_j$, which consists of the partial kernel multiplication processing elements (PEs) of a number equal to the number of support vectors, and(b) illustrates the functionality of a single PE.*

In this work, the multiplication operation carried out in the three sub-blocks is performed using DSP48 blocks available on the Xilinx' Virtex-4 FPGAs and subsequent series. The use of this dedicated hardware ensures fast kernel computation and enhances the overall performance of the SVM classifier [32] and [110].

B) A2 Architecture (SVs>>M)

The components of the kernel computation block of the A2 architecture of the SVM classifier are similar to SVM A1 architecture. However, the difference lies in the arrangement and number of PEs constructing the systolic array of Multiplier A. The systolic array now scales with M instead of SVs in A1 architecture, where M PEs comprise the systolic array responsible for computing equation (6.13), each having a local memory attached to it having a depth of SV as was explained in subsection 6.4.1.1 (B).

This architecture serves the case when SVs >> M, parallelising the computation of the kernel partial product by allowing M computations to be carried out simultaneously. On the other hand, the query M features are stored in registers within the sub-block as they are needed by the PEs every clock cycle. Each of the M PEs has the role of receiving a feature from one SV every clock cycle, and propagating the partial product result to the next PE in the pipeline to complete the computation of equation (6.13) for the same SV. Unlike A1 architecture, the computation of equation (6.13) for one SV is partially carried out by each PE, and the result of the last PE of the systolic array will correspond to the final result of equation (6.13). The latency of this sub-block is M clock cycles, and the throughput is one result per clock cycle. Consequently, the time needed for this sub-block to complete the computation of the kernel's partial product is the same as A1 architecture, being M+SVs. Fig. 6.7 outlines the arrangement and functionality of A2 architecture.

Lastly, the computation of the remaining parts of the kernel is activated in a similar way to A1 architecture, whereby Multiplier B starts working after a period of M-1, such that after M clock cycles the results from both sub-blocks of stage one are ready and synchronised, allowing the sub-block of stage two to start receiving results. DSP48 blocks are also used in each PE of the systolic array of the first sub-block, where M DSP48 blocks are needed, whereas for the remaining two sub-blocks one DSP48 is needed for each.

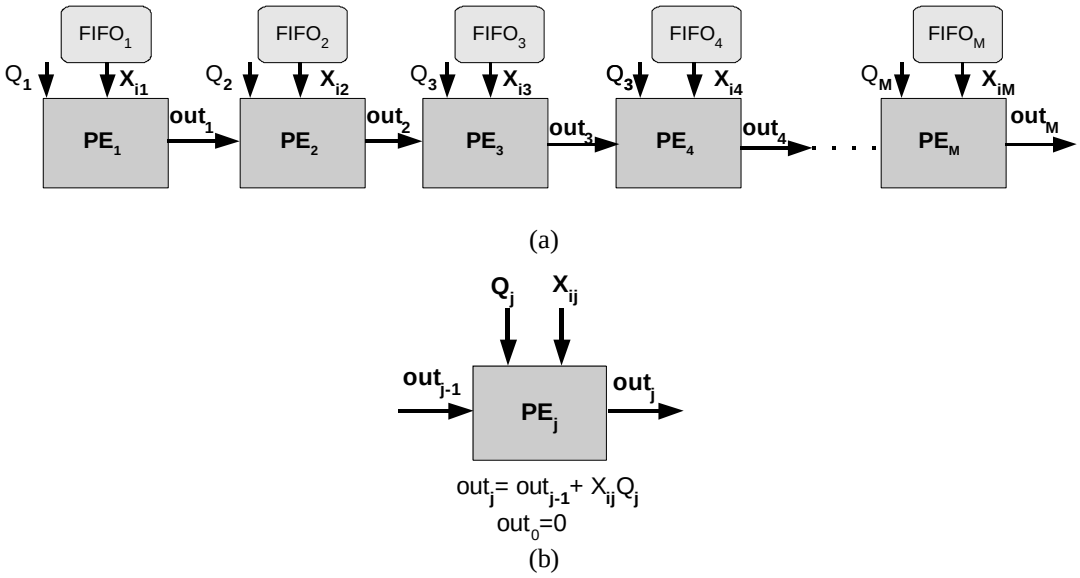


Figure 6.7: *(a) Systolic array architecture of the first sub-block in the Kernel computation block for SVM A2 architecture, and (b) the functionality of each PE, note that the Q_j value is one feature of the query stored in a register within the PE as it is used in every clock cycle until all SVs have been processed.*

6.4.1.3 Accumulation Block

This block is a simple add-and-accumulate circuitry required to accumulate the results as they come in from the kernel computation block (see Fig. 6.8). The size of the accumulator is dependent on the wordlengths (WLs) of the inputs received from the kernel computation and the number of SVs. Consequently, the size of the accumulator is given by equation (6.16):

$$Accumulator\ WL = Kernel\ WL + log_2(no.of\ SVs), \tag{6.16}$$

where the kernel WL is given by equation (6.17):

$$Kernel\ WL = 2B + B + log_2[M], \tag{6.17}$$

where B is the wordlength of each feature, and M is the number of features. Lastly, based on the number of SVs the final accumulation result is given by equation (6.18).

$$Accumulation\ result = \sum_{i=0}^{SVs-1}\left(\sum_{j=0}^{M-1} X_{ij} Q_j (\alpha_i y_i)\right) \tag{6.18}$$

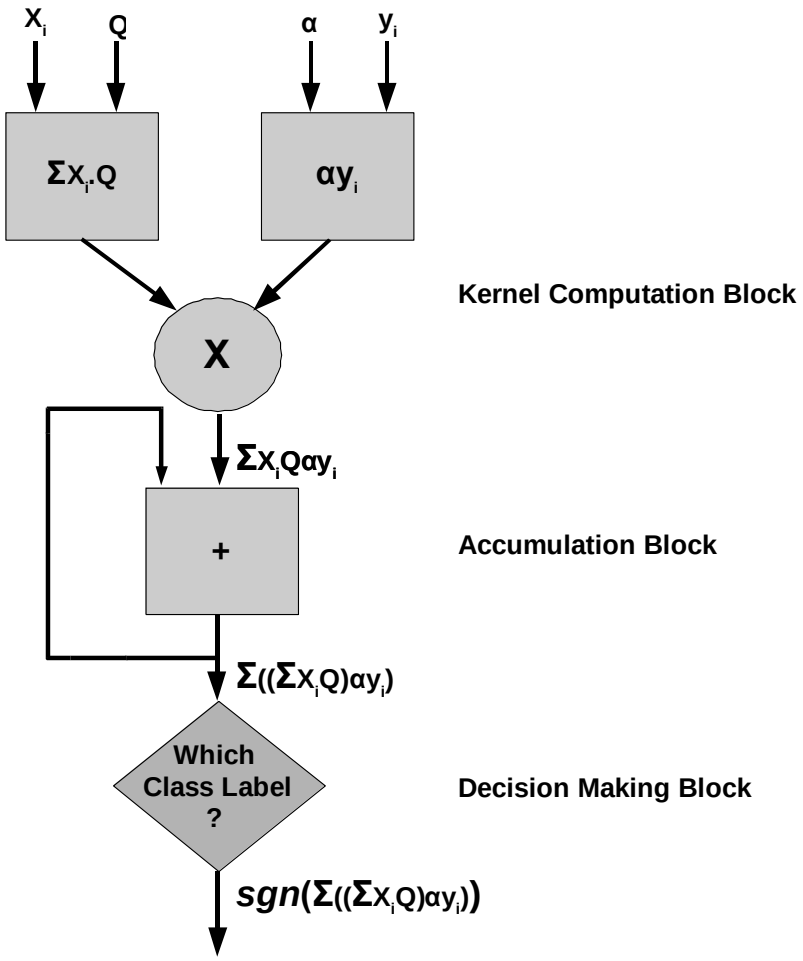


Figure 6.8: *The functionality of the accumulator and decision making illustrated with respect to the kernel computation block. The diagram illustrates the datapath of the complete SVM classifier applicable to A1 and A2 architectures.*

6.4.1.4 Decision Making Block

This block is the simplest circuit in the whole SVM classifier common to both architectures A1 and A2, whose role is to determine the class label of the query based on the sign of the accumulation result given by (6.19). Given that the SVM classifier proposed in this work targets a binary class label e.g., class zero distinguishes a diseased tissue and class one is healthy tissue. The block basically checks the MSB of the final accumulated result whereby the query is assigned a class label of 1 when the MSB is 1, and to zero otherwise.

$$Query\ class\ label = sign\left(\sum_{i=0}^{SVs-1}\left(\sum_{j=0}^{M-1} X_{ij}Q_j(\alpha_i y_i)\right)\right). \tag{6.19}$$

6.4.2 Multi-core Architecture of the SVM classifier based on A1 Architecture

The performance of the FPGA implementation of the two SVM classifiers illustrated above can be enhanced through the implementation of multi-core SVM classifiers similar to the architectures previously presented for the K-means clustering and K-NN classification in chapters four and five, respectively. The applications of the multi-core SVM classifier are the same as those explained in the previous two chapters, which are in short: the processing of multiple queries simultaneously, partitioning the training set among the multi-cores to further accelerate the classification of queries, performing ensemble SVM classifier and implementing server solutions for multiple users. Since the aim of this project is to investigate the benefits of applying the classifier to typical Microarray data, which are characterised by having large number of features, the proposed multi-core implementation is chosen to be based on A1 architecture of the SVM classifier described in subsection 6.4.1. However, this is not always the case, as some Microarray classification problems may require the use of A2 architecture.

The implemented multi-core architecture is based on quad-core implementation. It is based on fitting four complete single-core A1 SVM classifiers onto the same chip, with each having its own memory and processing different queries. All of the four SVM cores are based on the following parameters: (B=8, M=1024, and SVs=20). This implementation forms the basis for a DPR quad-core implementation which will be presented in subsection 6.4.4.

6.4.3 Novel DPR Implementation of Single-core SVM Classifier

An attempt is made here to investigate the candidacy of the SVM classifier and the benefits of applying DPR to the single-core SVM classifier. As such, two reconfigurable single-core implementation of the SVM classifier are presented here. The first reconfigurable single-core SVM classifier is based on an A1 SVM classifier, whereby the single-core is used to construct a PR design. PR is used to set the complete SVM core as reconfigurable RP following Xilinx' PR design flow and hierarchical methodology discussed in chapters 3 to 5 [38]-[39]. The second reconfigurable single-core is based on setting one core within a multi-core architecture as reconfigurable core. The quad-core SVM classifier described in subsection 6.4.2 is used in this implementation whereby one of the quad cores is made

reconfigurable only while the others are left static. The implementation is based on the following parameters for each core: (B=9, SV=20, and M=1024). Results of the two implementations will be presented and discussed in the subsequent sections to highlight benefits and advantages of such implementation in terms of reconfiguration time, placement and flexibility. Fig. 6.9 illustrates a block diagram of the reconfigurable single-core SVM implementation with respect to other non-reconfigurable tasks places on the FPGA.

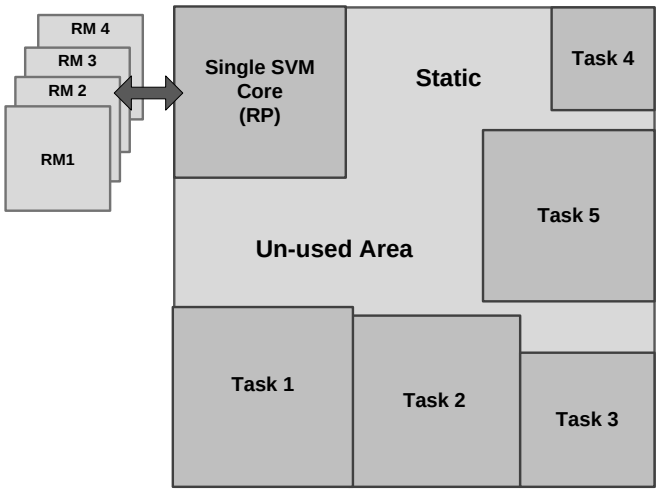


Figure 6.9: *Block diagram illustrating the placement and functionality of the reconfigurable single-core SVM classifier.*

6.4.4 Novel DPR Implementation of Multi-core SVM Classifier

To enhance the multi-core architecture proposed in subsection 6.4.2, and add more flexibility to it, a DPR implementation based on setting the entire SVM classifiers within the quad-core as reconfigurable partitions (RPs) has been constructed. The flexibility added here is in altering the contents of the memory in addition to the re-locatability of the cores. The latter is crucial in server solutions where users may request cores to be activated on demand consequently re-arranging existing cores within the same chip is necessary.

The DPR implementation requires the formation of a new hierarchical modular architecture (new design Wrapper) which instantiates the quad-core SVM classifiers first. Then each of the quad cores is set as RP. Second, multiple variants of each core corresponding to different memory contents, SVs, M, and B are created. These variants RMs will be used in creating multiple configurations. Third, the quad RPs are constrained within specific regions on the FPGA containing all the logic and resources required to account for

the maximum parameters chosen in any of the required RMs. Then, the bitstreams of several configurations are created corresponding to variable copies of the SVM classifiers (variable RMs). This process is associated with the generation of two bitstreams per configuration; one is full bitstream that can be used to configure/reconfigure the whole FPGA device, while the other is a partial bitstream that is used to partially reconfigure any of the quad cores during run-time as required.

The DPR implementations stated above permit the modification of any of the SVM classifier parameters such as the number of dimensions, support vectors, or wordlength of the features. However, an increase in any of these parameters must be accounted for beforehand, otherwise the implementation will fail. On the other hand, using reconfigurable modules (RMs) which reflect SVM cores having parameters smaller than those used to set the RP region will not impose any additional requirement. Fig. 6.10 illustrates this DPR implementation.

To sum up, it can be stated that DPR implementation of single core or multi-core SVM classifier allows for tailoring an existing core to suit user's requirements in terms of memory content, location, and classifier parameters given that hardware resources are accounted for. This is done dynamically without interrupting the operation of cores which are running on the same chip. Additionally, the multi-core architecture described above can be extended to work as an ensemble SVM classifier, where results from several cores can be combined through voting to enhance the accuracy of the classifier as discussed with the K-means clustering and the K-NN classifier. However, due to similarity in concept and application, this implementation will not be considered in this chapter. Instead, a multi-classifier approach will be implemented which makes use of both the K-NN and SVM classifiers presented in this thesis to dynamically reconfigure specific region within the FPGA as one classifier or the other.

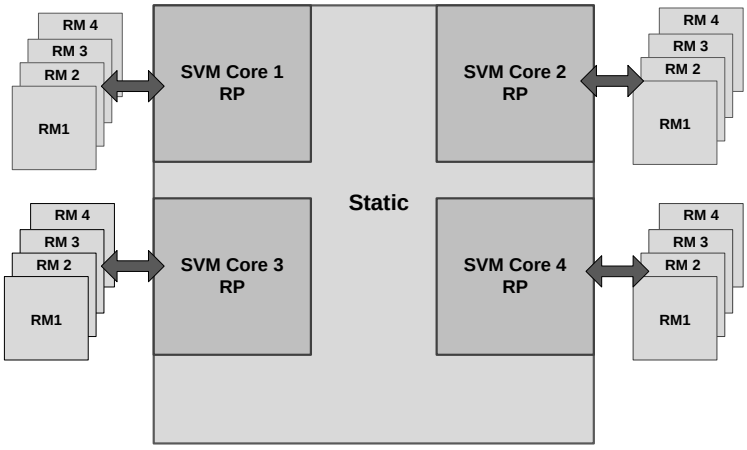


Figure 6.10: *Block diagram illustrating the placement and functionality of the reconfigurable quad-core SVM classifier.*

6.4.5 Novel DPR Implementation of K-NN/ SVM Classifier

Based on the fact that in many classification problems, users tend to experiment with different classifiers, a DPR design has been constructed based on a black-box classifier which can be configured as either SVM or K-NN classifier dynamically. The implementation is designed such that it allows for swapping any of the two cores (SVM of K-NN) in and out of the FPGA at run-time.

The size of the RP is set to accommodate the requirement by the largest RMs of the two classifiers, where the new RMs represent variants of each core implemented with different parameters. To make the two classifiers compatible and candidate for this application, the K-NN core was modified slightly to include an additional input to act as a dummy input such that the two cores have the same number of I/O ports, as the SVM classifier has a coefficient input whereas the K-NN lacks that input. This action was necessary as one of DPR design requirements is to have equivalent number of I/O ports for all the possible RMs, otherwise the implementation will fail due to interface mismatch.

The procedure to construct the SVM/K-NN classifier starts by creating a new design Wrapper which instantiates a black-box. The black-box is basically an empty core having the same I/O signals as the SVM classifier. The reason the SVM I/Os are selected is because the SVM include all the K-NN I/Os with the SVM has an additional input as mentioned above. Following the instantiation of the black-box, the PR design is created following the usual Xilinx PR design flow methodology to set one black-box as RP region on the FPGA, add two RMs corresponding to the K-NN and SVM cores, and define the locations and size of

the RP based on the largest resource requirement. Several configurations are then created along with the associated bitstreams. The implementation results will be presented in the following subsections.

6.5 Implementation Results

This section presents the results of the aforementioned implementations, which include results from both hardware and software implementations of the SVM classifiers. In hardware, synthetic data of different sizes were used that can be stored within the Block RAMs of the selected FPGAs. The hardware implementations targeted two different Xilinx' FPGAs, namely: *XC4VFX12* and *XC4VSX35*. On the other hand, the software implementations on GPP are based on using Matlab (R2009b) bioinformatics toolbox running on a 2.60 GHz Pentium Dual-Core E5300, with 3 GB RAM workstation. The toolbox includes an optimised SVM classification function that can be easily utilised, in addition to using the Fixed-point toolbox to quantise the features according to the selected precision and required integer WL. The following subsections summarise the implementation results.

6.5.1 Single-core Implementation of the SVM Classifier based on A1 Architecture

The proposed single-core A1 architecture of the SVM classifier was simulated first using synthetic data which mimic Microarray dataset, then synthesised, mapped, placed and routed using Xilinx ISE 12.2 to target the *XC4VFX12* FPGA available on Xilinx ML 403 platform board [41]. The implemented design was based on the parameters (B=8, M=1024, and SVs=20). The place and route results are shown in Table 6.1.

The hardware implementation was tested using Xilinx' ChipScope^{TM} Pro Analyser 12.2 and checked against simulation results. The number of clock cycles to classify one query was found to be 1048, achieving an execution time of 10.62 µs based on the attained frequency shown in Table 6.1. On the other hand, the execution time of the GPP implementation was 675 µs based on taking the average of ten thousands runs. Consequently, the FPGA implementation of the A1 architecture outperformed the GPP implementation by approximately 61 times, as summarised in Table 6.2.

Table 6.1: Place and Route Synthesis results of Single-core SVM classifier based on A1 Architecture

Device	Xilinx' *XC4VFX12-12ff688*	
Parameters	(B=8, M=1024, SVs=20)	
	Used/Available	**Utilisation ratio (%)**
Slices	1,703/5,472	31
Slice FF	2,137/10,944	19
4 input LUTs	1,799/10,944	16
Block RAMs	23/36	63
DSP48	22/332	68
Clock Frequency	98.7 MHz	

The same design was implemented using a higher end FPGA, namely Xilinx' *XC4VSX35*, achieving a frequency of 137.7 MHz, which lead to hardware execution time of 7.64 µs, consequently achieving a speed-up of ~85 times over an equivalent GPP implementation, this finding was based on simulation results only; actual implementation on board was not feasible due to the unavailability of the large hardware device.

Table 6.2: Summary of Timing performance of the A1 architecture

GPP Software (µs)	**FPGA (µs) Based on 98.7 MHz clock speed**	**Speed- up**
646	10.62	~61

6.5.2 Single-core Implementation of the SVM classifier based on A2 Architecture

Following the same procedure above with A2 architecture of the SVM classifier, the design was implemented with the parameters (B=8, SVs=1024, and M=20). Table 6.3 states the place and route results of the FPGA implementation of A2 architecture.

Similar to the previous implementation, the hardware implementation was tested using Xilinx ChipScope™ Pro Analyser 12.2 and checked against simulation results. The number of clock cycles to classify one query was found to be 1048, which was the same as the previous block as was expected since the two architectures require the same number of clock cycles to classify one query. However, the attained clock frequency of the A2 architecture was higher than A1 as reported in Table 6.6 leading to an execution time of 7.34 µs.

Table 6.3: Place and Route Synthesis results of Single-core SVM classifier based on A2 Architecture

Device	Xilinx' *XC4VFX12-12ff688*	
Parameters	(B=8, M=20, SVs=1024)	
	Used/Available	**Utilisation ratio (%)**
Slices	1,206/5,472	27
Slice FF	1810/10,944	17
4 input LUTs	1,705/10,944	15
Block RAMs	21/36	58
DSP48	21/32	65
Clock Frequency	142.9 MHz	

On the other hand, the execution time of the GPP implementation was 359 µs based on taking the average of ten thousands runs. Consequently, the FPGA implementation of the A2 architecture outperformed the GPP implementation by approximately 49 times, as summarised in Table 6.4.

Table 6.4: Summary of Timing performance of the A2 architecture

GPP Software (µs)	**FPGA (µs) Based on 142.9 MHz clock speed**	**Speed- up**
359	7.34	~49x

6.5.3 Effect of Data Dimensionality: GPP vs. FPGA

The FPGA implementation of the SVM classifier was compared with GPP implementation running on Matlab (R2009b) bioinformatics toolbox to particularly investigate the effect of changing the number of dimensions M on the timing performance of the two implementations. The number of SVs was fixed at 1024, B at 16, and M was varied from 1 to 16. Fig. 6.11 shows that when M was increased, both implementations took longer times, however FPGA suffered less in classification time as compared with the GPP implementation. For instance, changing M from 1 to 16 features increased the classification time by 1.4% only for the case of the FPGA implementation as compared to 5.7% for the GPP implementation. Fig. 6.11(b) implies that increasing the dimensions of the SVs yields significant increase in the classification time of the GPP implementation as compared to smaller increase for the FPGA case as shown in Fig. 6.11(a). Thus, the GPP implementation

seems to scale linearly with increasing dimensions while the FPGA implementation seems to be nearly constant.

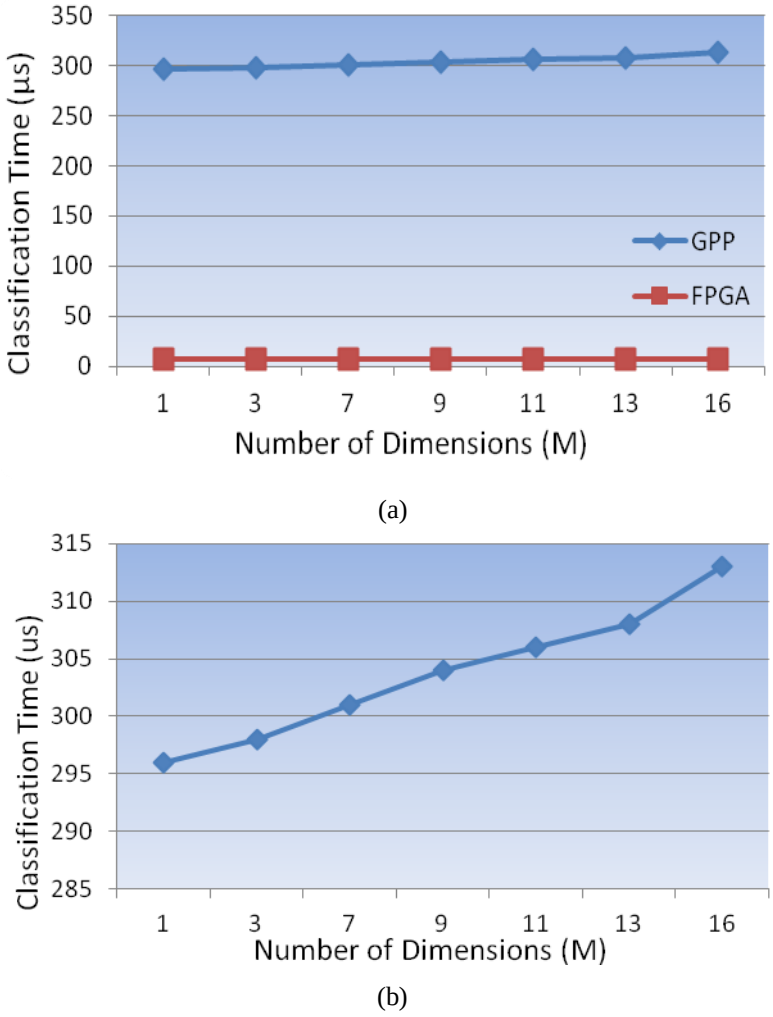


Figure 6.11: *The effect of increasing the dimensions (M) on classification time for the FPGA and GPP implementations of the SVM classifier based on (B=16, M=1024, SVs=16): (a) GPP and FPGA, and (b) Enlarged GPP to emphasise the timing effect.*

Increasing the dimensionality of the SVs beyond those shown in Fig. 6.11 causes the GPP implementation to suffer more due to the fact that the multiply-accumulate operations involved in the kernel computation phase will have to access the main memory to store the intermediate results since the cache will not be able to hold these values. On the other hand, the FPGA implementation still maintains its performance due to the abundant on-chip storage resources and the use of fixed precisions set carefully to avoid wasting resources.

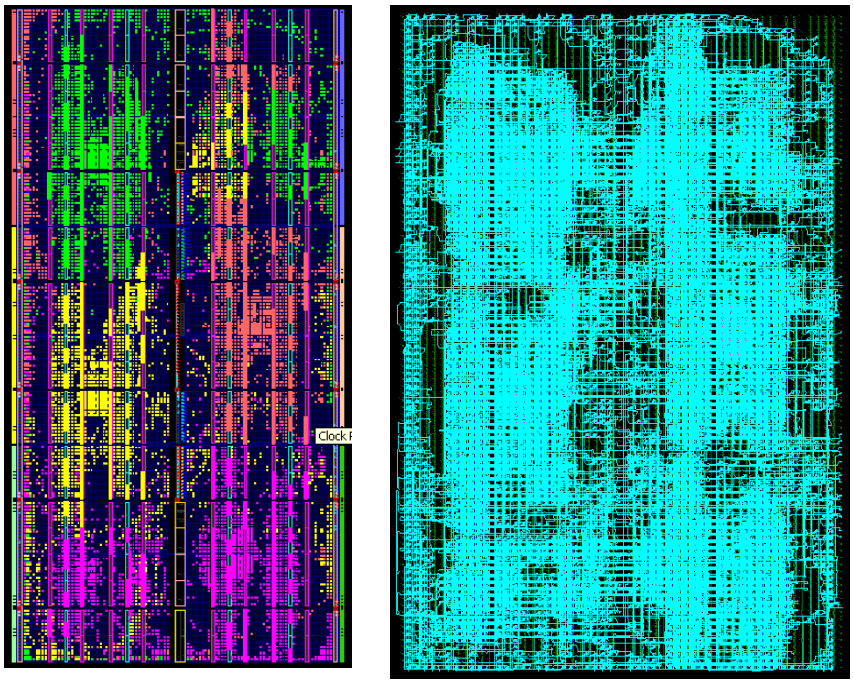
In summary, it can be stated that FPGA generally scales better than GPP when the dimensions of the SVs are large, mainly due to the extensive pipelining and parallelism employed in the FPGA implementation of the SVM classifier as compared to pure sequential behaviour in GPP, and due to the abundant distributed memory in medium and high end FPGAs.

6.5.4 Multi-core implementation of the SVM Classifier based on A1 Architecture

The proposed quad-core architecture was simulated, synthesised, mapped, placed and routed using Xilinx ISE 12.2 to target the *XC4VSX35* FPGA. Each of the four SVM cores were configured with the following parameters: (B=8, M=1024, SVs=20). Therefore, the number of clock cycles required to classify a query by any of the four cores was found to be 1048 clock cycles, which was same as that of a single-core SVM classifier i.e., the cores perform totally independently of each other. As a consequence, the complete execution time based on the attained clock frequency was 7.64 µs for each core. The resources of the quad-core implementation compared to the single core architecture are shown in Table 6.5 indicating that the quad-core occupies four times the resources occupied by a single-core. Fig. 6.12 illustrates the area footprint and the device routing based on the targeted device.

Table 6.5*:*Place and Route Synthesis results for the Single and Multi-core FPGA implementations of the A1 SVM classifier

Device	Xilinx' *XC4VSX35-10ff688*			
Parameters	(B=8, M=1024, SV=20 for single-core for four-core)			
	Used/Available Quad cores	**Utilisation ratio (%)**	**Used/available Single core**	**Utilisation ratio (%)**
Slices	7,928/15,360	51	1,941/15,360	12
Slice FF	8,708/30,720	28	2,156/30,720	7
4 input LUTs	5,024/30,720	16	1,296/30,720	4
Block RAMs	92/192	47	23/192	11
DSP48	88/192	45	22/192	11
Clock Speed	137.7 MHz		137.7 MHz	



(a) (b)

Figure 6.12: *(a) The implementation of the quad-core SVM classifier showing the area footprint occupied by the FPGA, where each core is highlighted in a different colour, and (b) is the routed implementation.*

6.5.5 DPR Implementation of Single-core SVM classifier based on A1 Architecture

Based on the single-core implementation described in subsection 6.3.3, a DPR implementation was constructed using Xilinx' PlanAhead 12.2 tool. Fig. 6.13(a) illustrates the floorplan image of the implementation of a configuration based on an RM having the following parameters: (SVs=20, M=1024, and B=8). Other RMs were successfully created reflecting variable SVs and M. This is to ensure that the RP can cater for the resources required by the RMs since the RP is sized to accommodate maximum SVs and M of 20 and 1024, respectively. The configuration was run, verified, the full and partial bitstreams were generated. The targeted Xilinx device was the same as that reported in Table 6.5, namely, the *XC4VSX35* FPGA. The full and partial bitstreams were 1,673 KB and 199 KB in size, respectively. Similar to the previous two chapters, the full and partial reconfiguration times were computed using equation (6.20).

$$Configuration\,Time = \frac{Size\,of\,the\,bitstream\,file}{Bandwidth\,of\,the\,Configuration\,Mode} \tag{6.20}$$

The results reported here are based on the JTAG as configuration mode having a bandwidth of 66 Mbps. Accordingly, the full reconfiguration time was found to be 202.78 ms while the partial reconfiguration time was 24.12 ms, leading to a speed-up in partial reconfiguration time of ~8x compared to a full reconfiguration of FPGA according to equation (6.21).

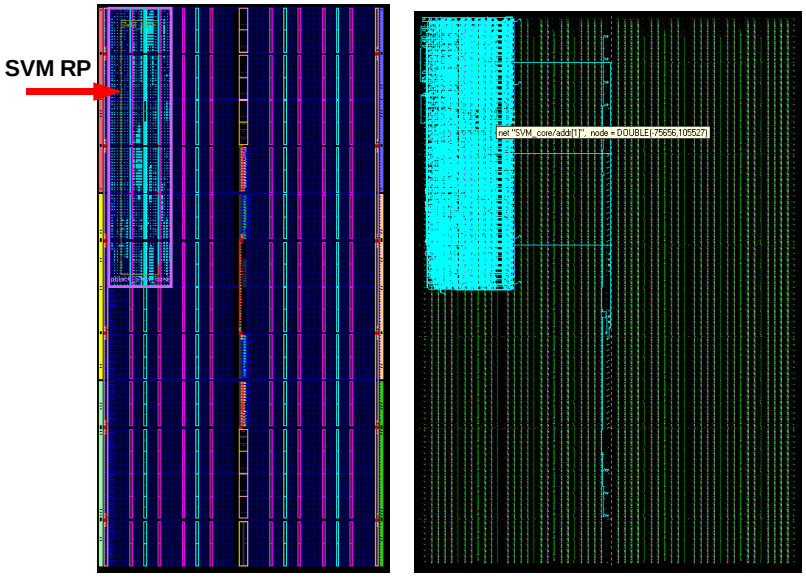
$$Speed\text{-}up\ in\ Reconfiguration\ Time = \frac{Full\,Reconfig.\,Time}{Partial\ Reconfig.\,Time} \tag{6.21}$$

This means that partially reconfiguring the FPGA not only leaves other tasks running on the FPGA un-interrupted, but also provides quick reconfiguration. Fig. 6.13(a) highlights the area footprint occupied by the reconfigurable SVM core and the compactness of the placement, which is further emphasised in the routing diagram shown in Fig. 6.13(b). On the other hand, unlike the case with the K-NN DPR implementation, the SVM DPR implementation was inferior to the equivalent non-DPR implementation in terms of clock speed. The DPR implementation achieved 94.8 MHz while the non-DPR implementation achieved 137 MHz, leading to drop in clock frequency by 31% when using DPR. This drop in clock speed could be attributed to the routing algorithm employed by the tool particularly related to routing the DSP48 blocks with respect to other logical resources.

As for the effect of using DPR on the area footprint for the aforementioned SVM classifier, it can be stated that DPR had negligible effect, where the DPR utilised 1908 CLB slices compared to 1941 for the non-DPR implementation, leading to 12% utilisation in CLB slices in both implementations.

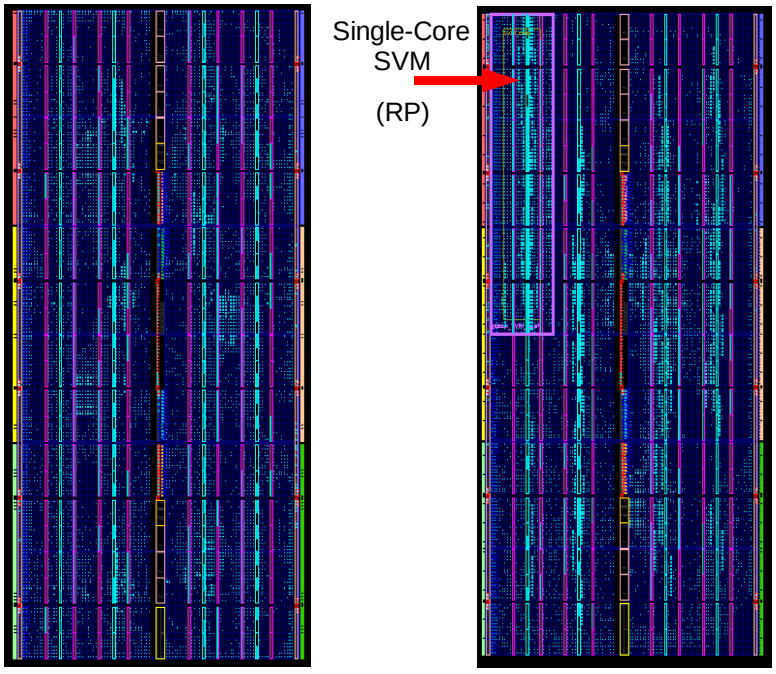
In addition, the second implementation based on reconfigurable single-core in the quad-core implementation of the SVM classifier was configured with the same parameters used for the aforementioned implementation. As such, the size of the RP was identical to the aforementioned single-core DPR implementation leading to partial reconfiguration time of 24.12 ms and speed-up of ~8x in reconfiguration time. Fig. 6.14 shows the implementations of this DPR implementation compared with the non-DPR quad core implementation. The two implementations are identical in partial reconfiguration speed-up since they are both based on the same FPGA as well as identical RP having the same parameters (SVs=20,

M=1024, and B=8). However, the second implementation illustrates an additional advantage of being able to dynamically reconfigure a single SVM core while leaving the other three SVM cores un-interrupted as well as any other tasks running on the same FPGA.



(a) (b)

Figure 6.13: *(a) Floorplan image of the DPR implementation of the single-core SVM classifier highlighting a small compact RP region, and (b) is the routed image.*



(a) (b)

Figure 6.14: *The Floorplan image of: (a) the non-DPR implementation of the quad-core SVM classifier, and (b) a DPR implementation based on a reconfigurable single-core.*

6.5.6 DPR Implementation of Multi-core SVM Classifier based on A1 Architecture

The results presented here are for the quad-core SVM classifier presented in subsection 6.4.4. Fig which was created using Xilinx' PlanAhead 12.2. 6.15 illustrates the floorplan of the implementation highlighting the four RPs and the area footprint occupied by each core targeting Xilinx' *XC4VSX35* FPGA. The full bitstream was 1,673 KB in size, while the partial bitstream was 199 KB for each of the four cores. Consequently, the full and partial reconfiguration times were 202.78 ms and 24.12 ms, respectively, resulting in a speed-up in partial reconfiguration time of ~8x over full chip reconfiguration. Partially reconfiguring individual cores not only ensures that other cores remain operable, but also ensure fast re-configurability of the core when only one core is to be altered. Additionally, power saving is gained from the avoidance of having to reconfigure the whole chip every time a small modification is required in one of the cores.

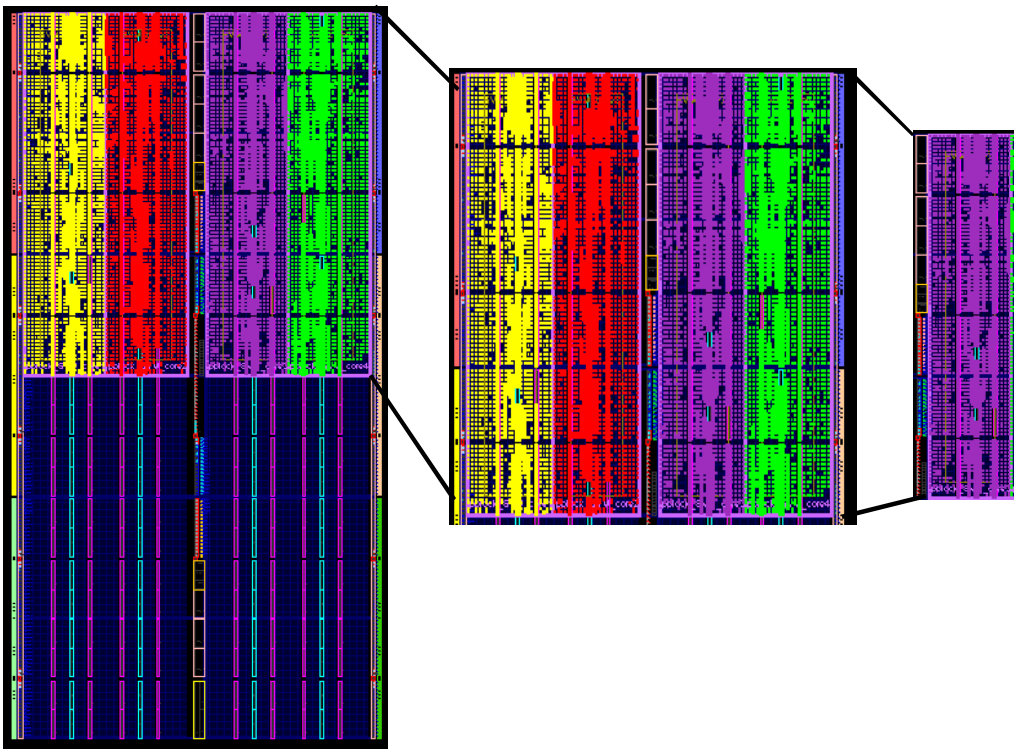
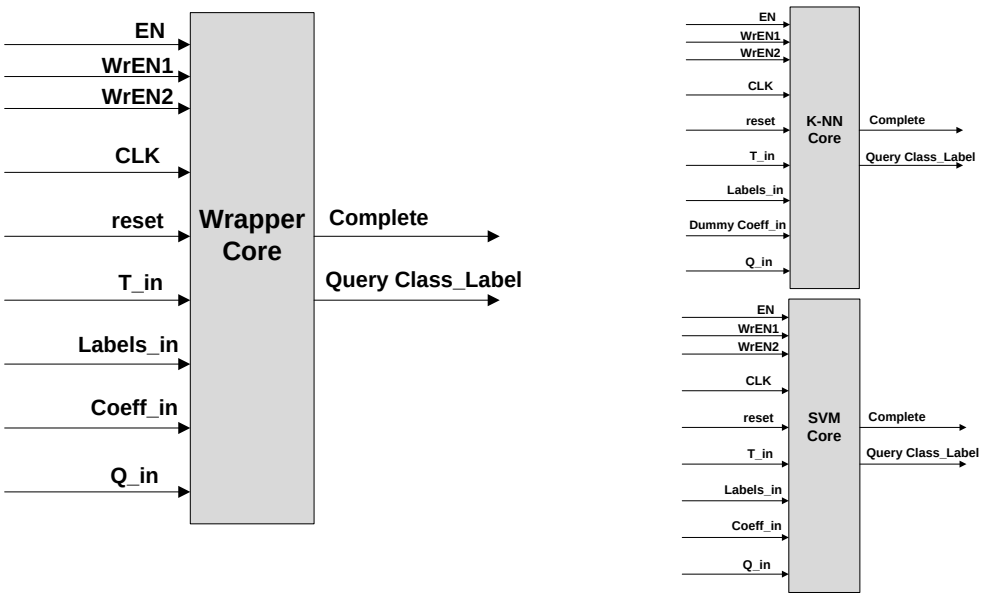


Figure 6.15: *The Floorplan image of the DPR implementation of the quad-core SVM classifier illustrating the area footprint of the quad core as well as of the single-core.*

Similar to the previous two applications, this multi-core implementation of the SVM classifier targets a server solution, whereby cores get added, re-located, modified, switched-on and off, all according to users' requests.

6.5.7 DPR Implementation of the K-NN/SVM Classifier

The Single-core K-NN and SVM classifiers were used in this implementation allowing for a particular RP region in the FPGA to be configured with either classifier offering the user the choice to select whichever to use, or to run both sequentially and compare the classification results of the two classifiers. The design Wrapper instantiates a black-box as was mentioned in subsection 6.4.5 having the same number of I/O ports required by the two classifiers is shown in Fig. 6.16. The parameters used in the implementation are the same as those required by the two classifiers being: (B=8, N=1024, SVs=1024, M=20, k=13) and the size of the RP region was set according to those preset parameters, leading to full and partial reconfiguration bitstreams of 1,673 and 199 KB, respectively based on *XC4VSX35* FPGA. Consequently the partial reconfiguration times for this multi-classifier implementation are the same as those obtained for the single DPR implementation of the SVM core, leading to ~8x speed-up in reconfiguration time. The area requirement for the SVM classifier was 1442 CLB slices compared with 1475 CLB slices for the K-NN classifier; and clock frequencies were 131.2 and 142.7 MHz for SVM and KNN, respectively.



(a) (b)

Figure 6.16: *The I/O ports of the SVM/K-NN classifier showing: (a) the ports of the wrapper module which instantiate a black box that can be configured as either an SVM or K-NN classifier, and (b) illustrates the I/O ports of both classifiers, emphasysing that the interface of both classifiers must match those of the wrapper.*

Several variants of the SVM and K-NN classifiers have been used to construct a collection of RMs that can be used to reconfigure the RP region as illustrated in Fig. 6.17. Note that the parameters chosen to create the RMs were of values that can be accommodated by the logic resources within the RP region. Following the implementation of several configurations, the associated bitstreams were created for each configuration. The resulted full and partial bitstreams were the same for all configurations as expected since the same device and same RP are used in all the configurations being 1,673 and 199 KB, respectively.

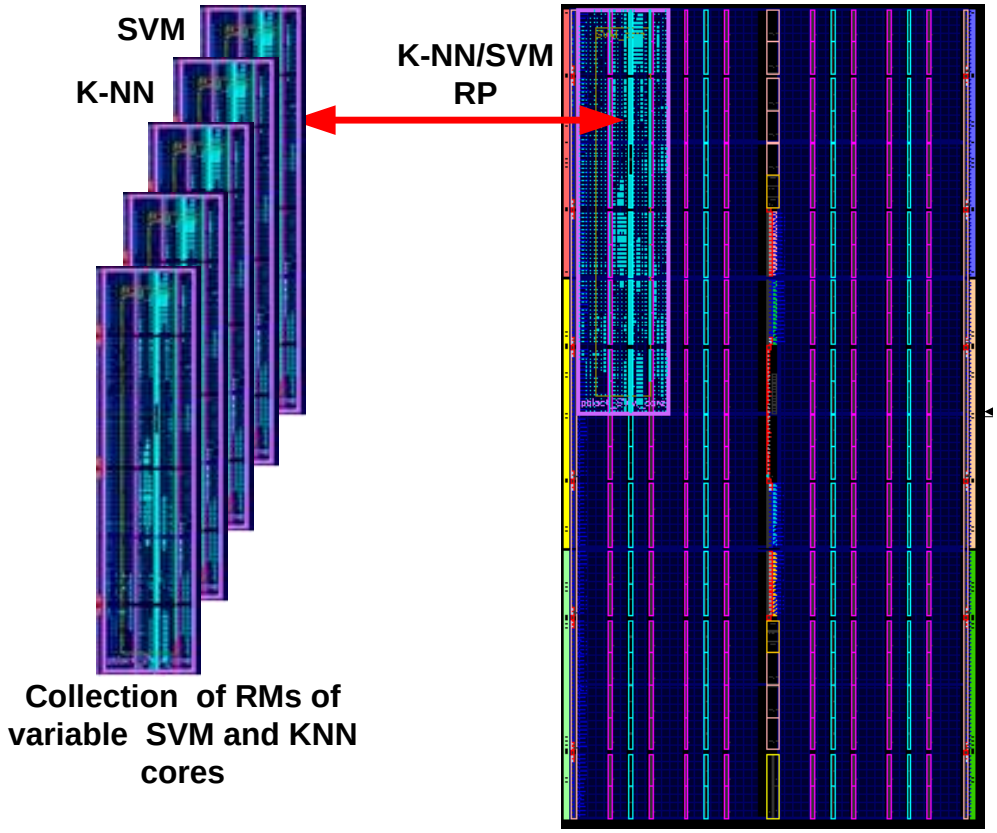


Figure 6.17: *Illustrative diagram showing the process of swapping the RP core with variants of the SVM and K-NN classifiers(RMs).*

6.6 Summary and Conclusions

In this chapter, the detailed architecture of two FPGA implementations of the SVM classifier have been presented, which form the basis of three novel DPR implementations. Similar to the K-NN classifier, the two architectures were called A1 and A2, respectively. The hardware architecture of the original two SVM classifiers based on a linear systolic array of processing elements (PEs) to partially compute the kernel of the classification decision function. The latter is the most computationally demanding part, and one that is candidate for hardware acceleration due to its inherent parallelism. The number of PEs in each architecture are different, whereby the number of PEs in the first architecture is equivalent to the number of support vectors (SVs), and equivalent to the number of dimensions (M) in the second architecture. The single-core designs for both architectures were captured in Verilog HDL based on modular blocks each performing specific parts of the classification decision function employing high level of pipelining and parallelism. The designs were parameterised in terms of the feature's WL, the number of support vectors SVs, and number of features M. When comparing the performance of the two FPGA implementations with equivalent implementations running on GPP, A1 achieved speed-up of ~61x (based on B=8, SVs=20, M=1024), while A2 achieved ~49x (based on B=8, SVs=1024, M=20).

Based on A1 architecture, a quad-core implementation of the SVM classifier was presented which replicates the aforementioned single-core four times allowing for parallel classification of four queries, or partitioning a large dataset among the quad-cores leading to improved speed-up in classification compared to GPP.

Moreover, a novel DPR implementation of the SVM classifier based on reconfigurable single-core SVM classifier was presented. The proposed architecture attained a speed-up of ~8x in reconfiguration time over full device reconfiguration. A multi-core DPR implementation based on four reconfigurable SVM cores was also implemented whereby the quad cores were set as reconfigurable partitions (RPs) having identical sizes. Partially reconfiguring each core has resulted in speed-up in reconfiguration time of ~8x. In addition to the added flexibility in altering the contents of each core during run time without affecting the operation of other tasks and the speed-up in reconfiguration time, the DPR implementation benefits from the capability of relocating any of the cores. This implementation caters for the requirements of server solutions where cores can be added, modified, moved around the FPGA, or removed according to user's requests.

Lastly, a novel DPR implementation of multi-classifier, namely SVM/K-NN was presented in this chapter based on reconfiguring specific RP as either SVM or K-NN classifier. This solution is an alternative to having to configure the same FPGA with two classifiers resulting in reduction in area footprint. This implementation is particularly useful for cases when users conduct studies which look at classifications made using different classifiers such as ensembles, mainly to increase the accuracy of the implementation. The size of the RP was set for the case of the SVM classifier having the parameters: (B=8, SVs=20, M=1024), and the K-NN classifier having: (B=8, T=20, M=1024, C=2, K=13), and the resulted speed-up in reconfiguration time was ~8x.

In conclusion, the hardware implementation of the SVM classifier on FPGA realises high performance customised solutions applied to Microarray research, which outperforms GPPs in terms of execution. Additionally, the SVM classifier lends itself for non-DPR and DPR FPGA implementations. Due to lack of time SVM training has not been implemented in this work and only the SVM classification function was considered for its simplicity. Given that the classification function demonstrated promising results in terms of speed-up over GPP in addition to the success of the DPR implementation, future work will include implementing partially reconfigurable SVM training on FPGA.

Furthermore, future goals include testing the SVM cores with benchmark datasets instead of synthetic data, and running the DPR implementations on boards housing state-of-the-art FPGAs that can accommodate benchmark dataset. Additionally, the possibility to incorporate a time-multiplexed reconfigurable on-chip training core will be investigated, which can be swapped with the SVM classifier once the training phase is finished. Moreover, the possibility and advantages of incorporating a reconfigurable kernel block will be investigated, whereby RMs corresponding to different kernel functions e.g., linear or Gaussian, are used to reconfigure the kernel core within the SVM classifier instead of reconfiguring the whole classifier, consequently shortening the reconfiguration time for such application. As a short term goal, more rigorous tests will be performed on the DPR implementation of the multi-classifier (SVM/KNN) applied to real benchmark Microarray data including the option of more classifiers.

Chapter-7

Evaluation of FPGAs as High Performance Solution for BCB Applications

7 Evaluation of FPGAs as High Performance Solution for BCB Applications

7.1 Introduction

The FPGA implementations of BCB applications applied to Microarrays which have been presented in this thesis have shown superior performance in terms of speed-up, power and energy consumptions compared to other platforms, namely, GPPs and GPUs. In this chapter, an attempt is made to look back at the results presented in chapters 4, 5 and 6 and try to evaluate the FPGA technology within the context of BCB applications. The FPGA implementations presented in this thesis were designed with high performance strategies in mind, including full exploitation of parallelisms while minimising the area footprint. The performance of FPGA implementations is affected by the way the design is captured in HDL. As such, proper design and choices of resources are determinant factors in the overall performance and cost of the FPGA implementation. The following subsections intend to evaluate FPGAs as economic, high performance computing platforms for BCB applications compared to GPPs and GPUs. Comparison criteria are design entry, execution time, power and energy consumption, area footprint, development time, cost of purchase and development. The evaluations presented in the following subsections are based on the results obtained from the implementations proposed in this thesis as well as on results of other related implementations published elsewhere in the literature.

7.2 Design entry- resource allocation and HDL coding for parallel FPGA designs

FPGAs are characterised as having fine-grained granularity components i.e., register and LUTs that can be harnessed effectively for accelerating algorithms containing fine granularity instructions. In addition, the heterogeneous architecture of modern FPGAs facilitates high performance computing through the possibility to directly map some functions to dedicated hardware blocks such as DSPs that can be used efficiently to implement functions such as multiply and accumulate. Exploiting dedicated blocks ensures efficient design flow in terms of area footprint and clock speed. Furthermore, the abundant slice resources and their arrangement within the FPGA allow for replicating and distributing

some functions many times given that the logic resources available in the device are sufficient, facilitating the concurrent execution of those functions. This parallel execution is assisted by the capability of the FPGA to receive continuous data through its abundant I/Os or through localised Block RAMs that can be used to store data on-chip. The availability of such resources is based on the selected FPGA platform and device family.

Large FPGA vendors such as Altera and Xilinx provide so many resources for inferring the desired logic efficiently including HDL templates, direct instantiation of the resources into the design, and selection and customisation of functional blocks from library. Using these resources results in high performance implementation in terms of speed and area footprint. Furthermore, synthesis tools which are responsible for transforming the design entry to a netlist allow the user to specify design goals i.e., area or speed and set the desired efforts for the tool to reach the preset goal i.e., normal or high; as such, the tool will work toward achieving the desired design goals. All these factors contribute toward the performance of the design.

The level of pipelining and parallelism in any algorithm is mapped to the FPGA during the design entry phase. Consequently, this step is very crucial in determining the overall performance. Additionally, following stringent coding style when capturing the design in HDL reduces the amount of resources inferred leading to considerable reduction in the area footprint. As a result of the combined effect of high levels of parallelism and small area footprint, the cost of running the design will be lower. Moreover, using small amount of resources is coupled with lower power and energy consumption leading to lower operating cost. In the BCB case studies presented in chapters 4, 5 and 6, the designs were captured in Verilog HDL and were area optimised. However, the parallelism level for each case study varied from the others as will be discussed shortly.

In the FPGA implementation of the K-means clustering, the parallelism was achieved through the concurrent execution of the distance computation kernel whereby the number of distance processing units (DPs) is equivalent to the number of clusters (K) times the number of dimensions. As such, many DPs are mapped onto the FPGA running concurrently through the complete dataset to be clustered. The results of the DPs associated with individual dimensions for each cluster are then combined, resulting in K distances which are compared via a comparator tree to determine the closest cluster to the pattern being processed. The operation of the comparator is pipelined such that one result becomes ready every clock cycle after a latency period necessary to receive all K distances from the DPs and the number of results of the comparator tree. The data is stored in Block RAMs which distribute the

features of each pattern to the corresponding DP. The high bandwidth of the Block RAMs ensures continuous streaming of the data to the DPs facilitating their concurrent execution. On the other hand, when running the K-means clustering on GPP, the distance computation kernel runs sequentially whereby one distance is computed at a time. As a result of the high clock frequency of current GPPs, these could outperform some FPGA implementations when the number of features (dimensions) is small, enabling the storage of the data within the cache memory of the GPP. Nevertheless, when the number of features is high, the K-means algorithm running on GPP will be forced to access the main memory leading to slower execution. Consequently, in such circumstances, the performance of the FPGA will excel the GPP's. The high dimensionality is one of the prominent characteristics of biological data which is why BCB can benefit greatly from FPGAs. It is difficult to comment on mapping K-means to GPU as this task was not actually implemented in this project. Nonetheless, a comparison between the K-means implementation on FPGA and an equivalent GPU implementation published in [77] was attempted in chapter 4 using Xilinx' Virtex-4 device and Nvidia GeForce 8600 GT which are both 90 nm CMOS process technology. The comparison revealed that the GPU implementation was ~8x faster than the GPP and the FPGA was ~26x faster than GPP (see Table 4.7). On the other end, recent developments in GPUs have created ultimate high end devices containing many cores running at high clock speed and having high memory bandwidth. Such devices are eligible to compete with medium and high end FPGAs in terms of execution time. However, actual performance is subjective to the way the design is captured and mapped to both technologies and the size of the dataset.

As for the FPGA implementation of the K-NN classification presented in chapter 5, the design was based on systolic array architecture parallelising the distance computation kernel and the KNN finder (systolic array of comparators). Although two architectures were presented having different PEs within the systolic arrays, both architectures operate on the same principle whereby each distance calculation PE was allocated a local memory to ensure continuous streaming of the data to the pipeline. Additionally, the systolic array benefited greatly from pipelining whereby each distance PE was able to read one feature from its localised memory one clock cycle later than the previous distance PE, and the result of each individual PE propagated to the next PE. Similar to K-means, the K-NN kernels execute sequentially in GPPs suffering from lower execution and memory bottlenecks. No GPU implementation was attempted in this project. However the author of [84] performed GPU-FPGA comparison and found that FPGAs were 1.5x-3x faster than GPUs when using Xilins'

Virtex-II *XC2VP30-6* FPGA and GeForce 8800 GTX GPU. However, this comparison does not seem to be fair since the FPGA device used was based on 130 nm CMOS process technology compared to 90 nm for the GPU device. In addition, the comparison was based on a small number of features masking the effect of memory bottlenecks of the GPU device.

In addition, the FPGA implementation of the SVM classification presented on chapter 6 was based on a systolic array. Parallelism and pipelining were deployed where possible, in addition to the utilisation of the DSP blocks, leading to high performance of the FPGA implementation compared to the equivalent GPP implementation. No comparison with GPU was carried out with respect to execution time.

In all three case studies, a hierarchical design methodology was used during design entry to partition the kernels associated with each method enhancing the verification process and facilitating the DPR technology. The three case studies were re-implemented using DPR, a feature which allows the partial reconfiguration of the FPGA during run-time without interrupting the operation of other tasks. This feature particularly enhances the application of FPGAs in server solutions in which owners of tasks placed onto the same FPGA have limited access to re-programming the whole chip. In addition, DPR contributes to significant power saving. DPR capability helps in bridging the gap between GPPs and FPGAs in terms of flexibility of resource utilisation.

7.3 Comparative study: FPGAs vs. GPPs vs. GPUs

This subsection presents a comparison between FPGAs and other computing platforms, namely, GPPs and GPUs in terms of execution time, power, energy consumption and cost (purchasing, operation and development). In subsections 4.7 and 4.8 of chapter 4, comparisons between the three computing platforms were presented for the K-means clustering in terms of execution time, power and energy consumption. These results are revisited here in addition to newly added criteria to assist in evaluating FPGAs in high performance computing for BCB applications. Although great effort was exerted to match the K-means implementation in all three platforms for the purpose of conducting a fair comparison, the unavailability of large FPGAs has obstructed the actual on-chip measurements of some of the K-means implementations compared with the GPU implementations reported in [77] and [74]-[76]. Therefore, some of the execution times were based on synthesis results and simulations. Actual measurements of execution times and

power consumptions were based on using Xilinx' Virtex-4 *XC4VFX12* FPGA available on the ML 403 board.

7.3.1 Execution Time

The execution times of the three case studies, namely, K-means clustering, K-NN and SVM classification in GPPs and FPGAs were reported in chapters 4, 5, and 6, respectively. The results revealed that the single-core FPGA implementations applied to the three case studies using *XC4VFX12* FPGA outperformed equivalent GPP implementations by up to 32x, 76x, and 61x for the K-means clustering, K-NN and SVM classification, respectively; for details on the parameters chosen, the reader is referred to the corresponding chapters.

On the other hand, the results of the GPU implementations compared with both GPP and FPGA were reported for the K-means clustering only and were based on the results reported in [77] and [74]-[76] as mentioned earlier. The results were shown in Table 4.6 and normalised with respect to the GPP execution times as shown in Table 7.1. The results were based on full execution times of the algorithms, and on using the following platforms:

- GPP- Intel 2.2 GHz Core 2 Due, with 4 GB memory.
- GPU- Nvidia GeForce 9600M GT.
- FPGA- Xilinx' Virtex-4 *XC4VSX35*.

Table 7.1: The normalised speed-up of the K-means implementation on FPGA and GPU

Clusters	**FPGA/GPP Speed-up**	**GPU/GPP Speed-up**
0.4 MPx		
16	110x	10x
32	181x	18x
64	284x	25x
6 MPx		
16	92x	13x
32	173	25x
64	287	40x

The results reveal that the FPGA solution outperforms the GPP by approximately two orders of magnitude while the GPU by 10x to 40x although the GPU technology was more

advanced than the FPGA used in the comparison (the GPU is a 65 nm CMOS technology compared to 90 nm for the FPGA).

Additionally, another comparison was presented in Table 4.7 between the FPGA implementation of the K-means cluttering presented in this thesis with the equivalent GPP and GPU implementations presented in [77] which was based on compatible GPU and FPGA devices fabricated in 90 nm CMOS technology. The computing platforms used in this comparative study were as follows:

- GPP- Intel Pentium IV (no further details about the GPP were provided in [77]).
- GPU- Nvidia GeForce 8600 GT.
- FPGA- Xilinx' Virtex-4 *XC4VSX35.*

The comparison results reveal that the FPGA implementation outperform the equivalent GPP implementation by up to 50x times while the GPU implementation outperform the GPP implementation by up to 30x. The results also show that the performance of both FPGA and GPU improve as the dimensions of the data were increased with the FPGA showing higher efficiency than GPU as data sizes and dimensions grow. To sum up, both FPGA and GPU are eligible to become accelerators for such BCB applications. This fact is supported by the results revealed in the literature, one of which is the work presented in [19] comparing the three technologies for a popular BCB application, namely, the Smith-Waterman biological sequence alignment using compatible FPGA and GPU devices, both were based on 90 nm CMOS technology. The authors reported that the FPGA solution was two orders of magnitude quicker than the GPP, while the GPU was one order of magnitude quicker than the GPP implementation. The devices used in [19] were:

- GPP- Intel 3.4 GHz Pentium IV, with 1 GB RAM.
- GPU- Nvidia GeForce 8800 GTX.
- FPGA- Xilinx' Virtex-4 *LX160-11*.

The performance of the aforementioned FPGA implementations surpassed the equivalent GPP and GPU implementations in terms of executions times. Nevertheless, it can be stated that GPU's new cutting edge devices benefiting from large number of cores (and hence

threads), larger memory capacity and bandwidth could compete more aggressively with FPGAs in terms of speed-up.

7.3.2 Power and Energy Consumption

In this thesis, the power consumption of the K-means clustering implementation mapped onto the ML403 board was measured and compared with measurement made of a GPP implementation running the equivalent sequential version of the K-means algorithm as reported in chapter 4, subsection 4.8. However, the GPU power figures were extracted from the Nvidia data sheets [78]. The power analysis is based on the following devices:

- GPP- Intel 3.0 GHz Core 2 Due E8400 processor, with 3 GB memory.
- GPU- Nvidia GeForce 9600M GT.
- FPGA- Xilinx' Virtex-4 *XC4FX12* available on the ML403 board.

The measured power consumption of the FPGA and GPP implementations of the K-means clustering were 15 W and 90 W, respectively, leading the FPGA implementation to be 6x more power efficient than the equivalent GPP implementation. Moreover, based on the execution times of 0.024 s and 742 µs for of the K-means implementations on GPP and FPGA, respectively, the energy consumption of the former was found to be ~2 J as compared to 11141 µJ for the latter. Hence, the FPGA implementation was ~192x (two orders of magnitude) more energy efficient than the GPP implementation.

In chapter 4, Table 4.8 presented another power and energy consumption comparison based on the execution times reported in [75] including GPU's. Normalising the results of Table 4.8 with respect to GPP yields the figures illustrated in Table 7.2.

Table 7.2: Power and energy consumption of the K-means clustering implementations on GPP, FPGA, and GPU

Platform	Power (Watt)	Energy (Joule)	Normalised energy efficiency
GPP	120	517	1
FPGA	15	0.84	615
GPU	59	26	20

The results shown in Table 7.2 reveal that the FPGA implementation consumed the least power and energy followed by the GPU and GPP. Consequently, it can be stated that the FPGA solution of the K-means clustering is approximately three orders of magnitude more energy efficient than GPP followed by the GPU.

In [19], the authors reported similar findings regarding the power and energy consumptions of the Smith-Waterman biological sequence alignment implementation whereby the FPGA solution also achieved energy efficiency of three orders of magnitude higher than an equivalent GPP implementation, and one order of magnitude compared to the equivalent GPU implementation.

With respect to the power and energy consumptions of the K-NN and SVM implementation, they were not actually measured. Nonetheless, the speed-up of the FPGA implementations with respect to the equivalent GPP implementations indicate that the FPGA solutions are estimated to be more energy efficient that the GPP solutions by at least two orders of magnitude. Similarly, GPU implementation of the K-NN and SVM implementations are estimated to surpass equivalent GPP implementations in terms of energy efficiency by at least one order of magnitude.

In summary, it can be stated that the FPGA solution outperform both GPP and GPU solutions in terms of power and energy efficiency. This privilege of FPGA is expected to last for long in front of the power hungry GPP and GPUs. The aforementioned 128-core Nvidia GeForce 8600 GPU was rated for 155 W compared to 300 W for the GeForce GTX 690 GPU to accommodate the 3072 cores available in the latter device. As for the Nvidia GeForce 9600M GT used in the comparison shown in Table 7.2, the maximum power rating was 59 W for the 32 cores housed within this GPU. As such, the increase in the number of CUDA cores and memory will result in GPUs consuming considerably higher power.

FPGAs' power consumption increases however are starting from a lower base, meaning that they will last longer than GPUs before hitting a power wall.

7.3.3 Cost

The cost of any computing platform depends on several factors i.e., purchase cost, development cost, operating cost and maintenance/upgrading cost. At first, purchasing any technology is associated with its technical specifications and available resources e.g., logic resources, peripherals, hardened IP blocks and external memory. Computing platforms are

offered within a wide range of device families allowing a user to select the device having the right combination of resources for the application in hand. Table 7.3, reports the cost of the computing platforms used in the K-means clustering implementations which were reported in chapter 4, Table 4.3 based on using the following computing platforms:

- GPP- Intel 3.0 GHz Core 2 Due E8400 processor, with 3 GB memory.
- GPU- Nvidia GeForce 9600M GT.
- FPGA- Xilinx' ML403 board.

Table 7.3 reveals that the GPP solution is most cost effective in terms of purchase cost when compared with FPGA and GPU followed by the GPU solution and last by the FPGA solution.

Table 7.3: Purchase cost of the three computing platforms with/without host

Platform	Purchase Cost W/O Host (£)	Purchase Cost With Host (£)	Normalised Cost With Host (£)
GPP	N/A	747.55	1
FPGA	511	1260	1.69
GPU	127	874	1.17

Moreover, with respect to development time, GPP consumes the least time, followed second by GPU and third by FPGA. Design entry in GPP and GPU is associated with high level language (HLL) while FPGA is associated with low level language, namely, HDL. The latter is more complex process involving difficulties in verifying, debugging and learning how to use the API tools. To counteract these problems, high level synthesis tools (HLS) have been proposed since the late 1980s trying to raise the abstraction level of FPGA design by enabling design entry using HLL. However early HLS tools failed to solve this problem, but continued to evolve and overcome the weakness' of their predecessors. HLS tools convert the HLL design to hardware HDL which then get converted to netlist using the FPGAs vendor's synthesis tool e.g., Xilinix' ISE or Altera's Quartus II. Among the commercial C-based HLS tools are Xilinx' AutoESL, Celoxia's Handel-C, AutoPilot, Cadence's C-to-Silicon Compiles, NEC's Cyber Workbench, Synopsys' Synphony C and

Mentor's Catapult C. Although HLS has progressed significantly from being used in research and academia toward actual deployment, producing more efficient implementations and improved productivity, they still suffer from some limitations related to use of memory, on-chip debugging, and requirement for skills in FPGA design [111]-[112]. Furthermore, the level of complexity involved in designing with current HLS tools imposes the requirement for having a skilled FPGA engineer working in the same development team [9]. Nevertheless, additional evolutions in HLS tools will have the potential to override the long development time associated with FPGAs and reduce the associated cost. The latter contributes to faster time-to-market FPGA solutions as well as quick on-field programmability/upgradability.

Although Table 7.3 implied that GPP is the cheapest platform, this does not necessarily mean that GPP is the most economic solution since performance per dollar and performance per watt are more realistic assessments for the economic viability of the technology. For instance, although the authors of [19] reported that the GPP implementation of the Smith-Waterman algorithm was the most economic solution in terms of the overall cost including purchase and development costs, FPGAs actually scored the highest when it came to performance per watt and performance per dollar as shown in Table 7.4 adopted from [19]. Consequently, the authors stated that the FPGA was more economically viable than GPP and GPU despite the relatively high purchase and development costs of the FPGA. The economic superiority of the FPGA is nonetheless subject to it achieving at least two orders of magnitude speed-up compared to GPP and one order of magnitude speed-up compared to GPU in order to justify the overall cost of the FPGA as stated in [19].

Table 7.4: Performance per $ and per watt for the three platforms, adopted from [19]

Platform	Performance (MCUPS*) per $	Performance (MCUPS) per watt
GPP	1.18	13.7
FPGA	0.34	508
GPU	1.27	196

*MCUPS- Mega Cell Updates Per Second, a common performance measure used in computational biology.

In this research it was not feasible to assess the development time for the three case studies, or the performance per watt, performance per dollar as the GPU solution was not

actually implemented here. In addition, the skills of the author of this thesis in FPGA design were built gradually, thus development times are incomparable with skilled FPGA designers who might have developed the architectures in a shorter time. Consequently, conducting a thorough comparison such as that presented in [19] was not possible in this project, but could be done as part of future work. Nevertheless, the time consumed to develop the FPGA implementation of the K-means clustering was significantly longer than the time it took to develop the GPP implementation in Matlab (both developed by the thesis author). Moreover, the GPP implementations used for the FPGA-GPP comparisons reported in chapters 4, 5 and 6 were based on the using Matlab Statistical and bioinformatics toolboxes.

In addition to the FPGA implementations presented in this thesis, and the comparative study presented in [19] related to the Smith-Waterman algorithm, the authors of [10] have also reported similar results in favour of the FPGA implementation of DNA and protein sequencing. The reported results support the superiority of three high performance reconfigurable computing (HPRCs) systems, namely, the Cray XD1, SRC-6 and SGI Altix/RASC containing FPGA boards as co-processors over reconfigurable Beowulf clusters. The authors reported that the Cray XD1 achieved 2,794 speed-up over FASTA program running on Beowulf clusters; the Cray XD1 implementation was based on using six FPGAs running at 200 MHz with eight cores per chip. Additionally the Cray XD1 solution was 28x more cost effective than the Beowulf clusters and 148x more power efficient while being 29x smaller in area footprint than the Beowulf clusters. More astonishing results were reported for the SGI Altix/RASC solution compared to the Beowulf clusters with speed-up of four orders of magnitude, and less power by three orders of magnitude. More and above, the SGI Altix/RASC was 22x more cost effective while occupying 253x less area footprint than the Beowulf clusters [31].

From the aforementioned analysis of the performance results of the FPGA implementations presented in this thesis along with the results of the FPGA implementation of other BCB applications presented by other work reported in the literature, it can be stated that FPGAs are viable economic and high performance solutions for BCB applications.

7.4 Summary and Conclusions

In this chapter, an evaluation of FPGAs as economic high performance computing platforms in BCB applications was presented with respect to other computing platforms, namely, GPPs and GPUs. Evaluation criteria included the effect of the HDL design entry on the exploitation of resources, execution time, power consumption, energy efficiency and cost. The evaluation was based on the FPGA implementations of the BCB applications presented in this thesis, namely, K-means clustering, K-NN and SVM classifications in addition to other BCB implementations reported in the literature.

In the context of the implementation of the K-means clustering in the three technologies, namely, GPP, FPGA and GPU, the FPGA implementation outperformed the GPP by approximately two orders of magnitude while the GPU outperformed the GPP by one order of magnitude. As for the power efficiency, the FPGA solution was 8x more power efficient than GPP while the GPU was 4x more power efficient than GPP. Moreover, the FPGA implementation was approximately three orders of magnitude more energy efficient than GPP while the GPU was 20x more energy efficient that GPP as shown in Table 7.2.

In addition, when considering the development time, the FPGA solution was estimated to consume longer time to develop compared with GPP and GPU solutions. The long development time of the FPGA solution is associated with using low abstraction level for design entry, long verifications associated with creating testbenches, learning and setting-up API tools. In addition to the long development time associated with low abstraction level and immature high abstraction level tools, FPGAs are most expensive to purchase and require high skilled engineers as compared to GPPs and GPUs. Nonetheless, when execution times are large leading to significant speed-ups over GPPs and GPUs, FPGA's performance per $ and per watt become large, and as a consequence, the overall cost and development times are justified. The BCB applications demonstrated in this thesis and others reported in the literature have shown significant speed-ups over equivalent implementations in GPP and other computing platforms. As such, it can be stated that FPGA is a viable economic high performance solution for BCB applications.

Consequently, new research trends in the area should be directed toward optimising the FPGA solutions and advancing their usability to the BCB industry. More and above, new research should look at customising FPGAs for actual laboratory and clinical use.

Chapter-8

Summary, Conclusion, and Future Work

8 Summary, Conclusion and Future Work

8.1 Introduction

In this thesis, an attempt was made at assessing the viability of FPGA technology as a high performance economic platform for BCB applications. Towards this, the FPGA implementation of three data mining methods commonly used in the analysis of Microarray data were performed, namely, K-means clustering, K-NN and SVM classifiers. A particular emphasis was made on the use of these popular algorithms on Microarray data. This is because Microarray is one of the high-throughput biotechnologies used to analyse the genome of human and other organisms requiring intensive computations. However, the potential of this promising biotechnology is not fully exploited as a result of insufficiency of current computing platforms. The proposed implementations make use of state-of-the-art features of modern FPGAs including highly dense logic cells (LCs), heterogeneous resources (e.g., memory and DSPs), and dynamic partial reconfiguration (DPR) to overcome current limitation and enable the analysis of complex Microarray data.

The remainder of the chapter will present summary of the implementations and contributions to knowledge presented in the previous chapters followed by the conclusion of the thesis. Lastly, an outline of future work will be given including short term and long term goals.

8.2 Thesis Summary

After an introduction to the research question, research objectives and major achievements of this thesis in chapter 1, chapter 2 introduced Microarray as a high-throughput biotechnology widely used in genomics. The types of Microarrays were briefly outlined, the instrumentation and the experimental procedure were then explained. The chapter also discussed the pre-processing requirements of Microarray images and the data analysis methods used to convert the numerical Microarray matrices to biological or clinical findings. The data analysis methods introduced in this chapter included supervised and un-supervised methods. Furthermore, the current applications of Microarrays were outlined with focus on clinical applications related to cancer, a leading cause of death in most countries around the world thought to have heterogeneous causes influenced by the genome. Additionally, cancer

studies have confirmed that genomic factors affect the prognosis and treatment response of this disease. As such, few Microarray based diagnostic tools have already been developed and currently on use for confirming the diagnosis of the disease and assess its prognosis. However, limitations in terms of execution time, cost, practicality, and power consumption have been thought to deter the exploitation of the full power of Microarray data and slow its progression toward more clinical use. The higher potential of Microarray comes from the fact that combining multiple datasets leads to asking complex biological or health questions that will definitely give rise to new scientific findings. Finally, the chapter concluded with remarks about the rationale behind selecting to implement K-means clustering, K-NN and SVM classifiers on FPGAs, which included overcoming limitations of current methods, the fitness of such methods to hardware implementation, and their popularity in BCB applications.

Chapter 3 provided background on FPGAs including an introduction to the technology, description of a generic Xilinx' FPGA ASMBL^{TM} architecture, and the main applications of FPGAs. Additionally, the chapter outlined the methodology used to map algorithms onto FPGAs including design flow and EDA tools (with respect to Xilinx design suite). The design flow included design entry whereby an algorithm is first captured in HDL, simulated, synthesised, implemented, and finally the bitstream file required to program the FPGA is generated. The chapter explored briefly some of the features available in current EDA tools to assist in capturing a highly optimised design entry. Furthermore, the DPR feature of current state-of-the-art FPGAs was introduced covering both design flow and significant design considerations. Finally the advantages of DPR were briefly outlined including modifying a task already running on a chip without re-configuring the whole FPGA, speed-up in partial reconfiguration time over full chip reconfiguration, area footprint reduction and power saving. The ML 403 platform board was also briefly introduced since it constitutes the testing platform for the case studies presented in this thesis.

Chapter 4 presented the hardware implementation of the K-means clustering algorithm targeting Microarrays. The initial work was centred on capturing the complete K-means kernels in Verilog, consisting of the distance computation, cluster assignment, accumulation-counting, division, cluster centroid updates and repeating the K-means clustering using the newly updated centroids. This process keeps iterating until convergence occurs, that is when the division results corresponding to the new centroids are similar to the current centroids or within an acceptable error. The design entry of the K-means kernels was based on modular hierarchical architecture whereby each K-means kernel was a module or block of its own;

the design's Wrapper instantiated each block and was named the single K-means core. This approach facilitated reuse of the individual blocks in subsequent implementations, simplified design verification, and facilitated the use of DPR. Additionally, the design was parameterised in terms of the number of clusters, wordlength of the data, size of the dataset (including number of patterns and dimensions). The hardware resources of the single K-means core are easily adaptive to the entered parameters enhancing the scalability of the core to the specified problem in hand. The single K-means core was tested on a Virtex-4 FPGA achieving an execution time of 32x more than an equivalent implementation running on a GPP, while consuming 6x less power resulting in the FPGA and being 192x more energy efficient than GPP. Based on the single-core architecture, a five-core architecture of the K-means was implemented targeting a Microarray server solution. The implementation achieved speed-up in execution time of 205x over an equivalent GPP implementation targeting Xilinx' *XC4VLX25* FPGAs. As a consequence, the five-core implementation was 615 times more energy efficient than the equivalent GPP implementation.

Furthermore, three novel implementations of the K-means clustering were presented in chapter 4. The first implementation was based on using DPR to partially reconfigure the distance computation kernel already placed onto the FPGA with variable distance kernel. Currently, the implementation supports two distance metrics: Euclidean and Manhattan. This implementation allowed the reconfiguration of the distance block while the FPGA is running without the requirement to reconfigure the whole device. In addition to the capability of clustering with variable distance metrics without disrupting the operation of other tasks on the same FPGA, the partial reconfiguration demonstrated time saving with respect to full chip reconfiguration of ~10x when targeting *XC4VFX12*. This speed-up in reconfiguration time is amplified when targeting larger FPGAs.

The second DPR implementation was based on setting the complete K-means core as reconfigurable partition, thus DPR is used to partially reconfigure the complete core rather than reconfiguring a specific portion of it. This implementation allowed for modifying the complete core, such as exchanging the K-means core running on FPGA with a variable core. Variations are based on user desires including changing the number of clusters, changing the wordlength or size of the Microarray data given that any increase in logic resources is accounted for when setting the size of the reconfigurable region on the FPGA. The implementation achieved ~5x speed-up in partial reconfiguration time with respect to full chip reconfiguration targeting *XC4VFX12*, with the speed-up improving significantly when larger chips are used. From examining the speed-up result of the two aforementioned DPR

implementations, it can be stated that altering specific kernel within the K-means clustering such as the distance kernel is twice faster than reconfiguring the complete core. Consequently, the first DPR implementation is favoured for cases requiring the alteration of the distance metric only, while the second DPR implementation is favoured when more modifications to the core are required.

Moreover, the third DPR implementation was based on multi-core architecture whereby eight-core implementation was constructed achieving speed-up in partial reconfiguration time of 17x over full chip reconfiguration targeting Xilinx' *XC4VLX60* FPGA. The main advantage of the multi-core DPR implementation beside the time saving and continuous operation of the FPGA, is that changing specific core is carried out as needed only while other cores remain running without disruption. This feature is highly desirable in server solutions whereby cores placed onto a large FPGA belong to different users. Consequently, reconfiguring the whole device will impose interruption to many users. Additionally, in server solutions, full device reconfiguration is usually granted to a server administrator to prevent individual users from interrupting the operation of crucial tasks. Therefore, DPR grants individual users access to reconfigure specific regions on the FPGA allocated for their own tasks.

Over and above, the FPGA implementation of the K-means clustering was compared with an equivalent and recent GPU implementation reported in the literature in an attempt to evaluate the two technologies for this particular algorithm with respect to GPP. The comparison looked at aspects of execution time with respect to dimensionality, size of the data and the number of clusters. Another aspect of this comparison was the power consumption of three platforms GPP, GPU, and FPGA. The single-core FPGA implementation of the K-means clustering and the GPU implementation both outperformed the GPP implementation as the number of clusters increased from 16 to 64, with FPGA achieving twice the speed-up of the GPU implementation. Furthermore, the effect of changing the dimensions of the data on the performance of FPGA and GPU with respect to equivalent GPP solution was studied resulting in the five-core FPGA implementation being 16x and 33.5x faster than GPU for four and nine dimensions, respectively. The FPGA solution was found to behave robustly as the dimensions and number of clusters were increased compared to GPP and GPU implementations. This was mainly attributed to two facts. The first was the higher level of parallelism in the FPGA solution compared to the sequential behaviour in GPP. The second was the efficient memory access in FPGA compared to GPP and GPU. Although GPUs exploit parallelism, they suffer from memory

bottlenecks associated with access to global memory. It is worth noting that higher end GPUs and FPGAs could demonstrate variable results depending on the amount of available resources and on the level of parallelism exploited through proper design entry using HDL for FPGAs; and CUDA or other API programming for GPUs. Nonetheless, the variability of FPGA devices within the same device family offers the choice of selecting the FPGA chip that is best suitable for a particular application in terms of resource variability, density, and clock speed, whereas GPU vendors offer small number of devices within the same family range restricting the selection to fewer options.

Comparing power consumption of the three technologies while running the K-means algorithm revealed that FPGA was ~8x more power efficient than GPP and ~4x more power efficient than GPU. Additionally, the FPGA solution was the fastest in execution time followed by GPU and then GPP. Consequently FPGA was found to be ~615x (three orders of magnitude) more energy efficient than GPP and ~31x more energy efficient than GPU.

In chapter 5, the FPGA implementation of the K-NN classifier was presented. The K-NN classifier is a commonly used supervised learning method in BCB applications, which is based on using a training set (Microarray data with known class labels) to construct a classification model to be used to classify unknown samples. The K-NN classification consists of three main kernels: distance computation, the K minimum distance finder and their labels (known as KNNs), and a voter which finds the most commonly encountered label among the KNNs. Chapter 5 presented two different architectures of the K-NN classifier based on a systolic array architecture. The first architecture was referred to as A1 while the other as A2 architecture. The difference between the two architectures lays in the way the hardware resources are inferred and interconnected within the systolic arrays giving the option to select the architecture best suitable for the application in hand and the available hardware resources; whereby A1 is best suitable for applications having a large number of patterns and small number of dimensions while A2 for the opposite case. The FPGA implementation of A1 architecture was found to be ~76x faster than an equivalent GPP implementation, while A2 architecture was ~68x faster than GPP. A quad-core implementation of the K-NN classifier was also presented. This was capable of classifying four different queries simultaneously, with each core was based on different training sets.

Five novel DPR implementations of the K-NN classifier were additionally presented in chapter 5. The first implementation was based on reconfiguring a specific kernel within the classifier known to be sensitive to the parameter K. The choice of K has been known to affect the accuracy of the classification; as such classifying a sample using different K values

may yield different results leading to the requirement to experiment with different values of K. The main advantage here is the continuation of operation of other tasks on the FPGA as well as the reduction in reconfiguration time compared with complete core reconfiguration and complete device reconfiguration. The DPR implementation achieved between 4x to 5x speed-up in partial reconfiguration time compared to full device reconfiguration.

The second DPR implementation of the K-NN classifier was based on reconfiguring the complete K-NN core instead of reconfiguring a portion of the core as stated in the previous paragraph. The implementation achieved ~5x speed-up in reconfiguration time compared to full chip reconfiguration based on a maximum K of 15.

The third and fourth DPR implementations were based on multi-core and ensemble K-NN classifiers, respectively with both achieving speed-up in partial reconfiguration time of ~4x over full chip reconfiguration. As such, in addition of maintaining the operation of some tasks placed onto the FPGA, the two DPR implementation offer quicker reconfiguration. Ensemble here is the act of combining the classification of a query using three K-NN cores each configured with different K. Voting is then used for combing the results of the multi cores. More cores can be added to the multi-core or ensemble architecture depending on the available resources and design requirement. Additionally, DPR can be used to reconfigure the three cores with new Ks to include more classification results to the voter.

The fifth DPR implementation based on a ten-core ensemble classifier was also presented which only sets the memory block as reconfigurable partition. The implementation is specifically useful in cases where training data need to be updated while the device is running. The implementation was based on a large FPGA, namely the *XC4VLX60* achieving ~10x speed-up in reconfiguration time compared with full chip reconfiguration. The aforementioned DPR implementations of the K-NN classifier form a collection of novel and flexible architectures adaptive to particular applications, mainly for a server solution.

Chapter 6 presented the FPGA implementation of the SVM classifier targeting the classification phase only. SVM is one of the relatively new classifiers which gained popularity in classification problems of Microarray data. It uses a given training set to construct a model for classifying unknown samples. Unlike K-NN which involves classification only, SVM requires both training phase and classification phase leading to more intensive computations especially when the size of the data is large. As such, SVM was found to be a candidate for hardware implementation. The FPGA implementation of only the classification phase was considered in this chapter assuming that training was done off-line

on a host which then supplies the classifier with the required support vectors and training coefficients. The main steps of the SVM classification are. First, the distance or kernel computation, which consists in measuring the distance between a query and all of the support vectors using a specific kernel- (the adopted kernel was the linear kernel). The second step accumulates the kernel results for all the support vectors involved. The last step assigns a class label to the query equivalent to the sign of the accumulated result.

According to the aforementioned steps, the hardware implementation of the SVM classifier was based on three main blocks performing the SVM classification. Similar to the K-NN classifier, two architectures of the SVM classifier were constructed using linear systolic arrays, namely, A1 and A2, scaling with the number of support vectors and dimensions, respectively. The design of the two architectures was captured in Verilog HDL adopting a hierarchical design methodology. The single-core A1 architecture of the SVM classifier achieved ~61x speed-up over an equivalent GPP implementation, while A2 achieved ~49x speed-up. Building on the concept of multi-core architecture proposed for the K-means clustering and the K-NN classification, a quad-core implementation based on A1 architecture was implemented for the SVM classifier. The implementation is capable of processing four queries simultaneously.

Chapter 6 included three novel DPR implementations of the SVM classifier. The first was based on a reconfigurable single-core for both A1 and A2 architectures; the second was based on multi-core; and the third was based on a multi-classifier, namely, K-NN/SVM. The single-core DPR implementation achieved ~8x speed-up in partial reconfiguration time compared to full chip reconfiguration on a Xilinx' *XC4VFX12*. In addition to the time saving, the implementation permitted the modification of the SVM core dynamically without interrupting other tasks on the FPGA. As for the quad-core DPR implementation that is meant to be used for server solution, each of the four SVM cores were made reconfigurable offering the advantage of modifying any core without affecting the operation of the others, achieving speed-up of ~8x in partial reconfiguration time over full chip reconfiguration. Moreover, the quad SVM cores were also re-locatable leading to the capability of changing the locations of the cores whenever needed. In addition, the K-NN/SVM classifier permitted the reconfiguration of a particular region on FPGA to run as a K-NN or as SVM classifier dynamically. The implementation achieved speed-up in partial reconfiguration time of ~8x over full chip reconfiguration. In summary, chapter 6 provided a collection of SVM architectures adaptive to user parameters, offering high level of flexibility.

8.3 Conclusion

The overall aim of this thesis was to investigate the viability of FPGAs as high performance, economic solutions for BCB applications which are characterised as computationally intensive. This thesis approached this general research question in the context of BCB case studies applied to Microarray, a biotechnology characterised as being high throughput producing large amount of genomic or proteomic data that are known to be highly dimensional. The case studies consisted of one un-supervised and two supervised data analyses methods commonly used in Microarray, namely, K-means clustering, K-NN and SVM classification. FPGA implementations were assessed in terms of execution time, power consumption and cost. This has led to the following conclusions:

- The adaptive FPGA implementation of the K-means clustering outperformed an equivalent GPP and GPU implementation by two orders and one order of magnitude, respectively; while being 615x (three orders of magnitude) and 31x (one order of magnitude) more energy efficient, respectively. Consequently, it can be stated that FPGAs are high performance computing platforms for this particular BCB application in terms of execution time and energy consumption. Lower energy consumption is associated with lower operating cost.

- The novel DPR implementations of the K-means clustering place FPGAs few steps forward ahead of where they used to be as a result of the added flexibility to alter the device configuration dynamically without interrupting other tasks on the FPGA and at speed-up between 5x to 17x over full chip reconfiguration. In addition, DPR has shown capability of swapping out particular parts of the algorithms, namely, the distance computation kernel and swapping in another variant corresponding to an alternative distance metric. The implementation based on reconfiguring the distance metric of the K-means core with Euclidean or Manhattan distance resulted in 10x speed-up in reconfiguration time.

- The adaptive FPGA implementations of two variable K-NN classifiers, namely, A1 and A2 architectures outperformed equivalent GPP implementations by 68x and 76x, respectively. Consequently, it can be stated that FPGAs are high performance computing platform in terms of execution time for this particular BCB application. Although power and energy results were not actually measured for this

implementation as with the K-means clustering, it can be implied from the above that this application is more energy efficient when run on FPGA as compared with GPP, leading to lower energy cost.

- The novel DPR implementations of the K-NN classifier on FPGAs have the following advantages: it offered between 4x to 10x speed-up in partial reconfiguration time, allowed for the dynamic partial reconfigurations of the cores, added flexibility to relocate K-NN core within the same FPGA, and saved area footprint.

- The adaptive FPGA implementations of two SVM classifiers, namely, A1 and A2 architectures outperformed equivalent GPP implementations by 49x and 61x, respectively. Consequently, it can be stated that FPGA is a high performance computing platform in terms of execution time for this particular BCB application. Similar to both K-means and K-NN cores, the FPGA implementation of the SVM classifier is associated with lower energy and operating cost.

- The novel DPR implementations of the SVM classifiers of the FPGA offered 8x speed-up in partial reconfiguration time in addition to having the same advantages as K-means and K-NN cores.

- A novel DPR implementation of the K-NN/SVM multi-classifier illustrated the capability to dynamically reconfigure portions of the FPGA with the desirable BCB application. The implementation achieved 8x speed-up in partial reconfiguration time.

From the conclusions drawn above, it can be stated that current state-of-the-art FPGAs offer high performance computing in terms of execution time, power efficiency, energy efficiency, and re-configurability at run-time when applied to BCB applications. Consequently, FPGAs can become reliable, high performance and economic solution for the computationally intensive BCB applications in the near future. FPGAs have the potential of complementing the performance of GPPs and transforming BCB research to a higher level as a result of the higher processing capabilities offered by modern FPGAs. In addition, the capability of using FPGAs as add-ons to larger computer systems permits the customisation of such boards for particular BCB applications.

8.4 Future Work

Although the objectives of this thesis have been met regarding the assessment of the viability of FPGAs as economic high performance solutions in BCB applications, there is still great potential for additional work to be done to improve the implementations presented in this thesis and enhance their performance, as well as tackle other BCB applications. The following outlines short and long term goals:

- **K-means clustering**: Improving the K-means core by adopting a divider consuming smaller area footprint and having shorter latency. One consideration is to use soft processors such as MicroBlaze or use hard core processors such as Power PC, and exploring the potential of using time-multiplexed divider/K-means core whereby the K-means clustering can be divided into two main cores, one is the K-means core performing the distance and accumulation kernels, while the other is a divider core performing the division operation only to obtain the new cluster centroids. As such, the divider can be time-multiplexed between different K-means cores to save area footprint making use of the fact that the divider is needed only once per iteration.

- Developing and testing self-reconfigurable DPR implementation of the K-means clustering, K-NN and SVM classifiers; whereby embedded processors are used as custom Internal Configuration Access Port (ICAP controller) to control the configuration/reconfiguration of the FPGA internally. Collaborative efforts have already been made by colleagues in the SLIg group at the University of Edinburgh to develop such systems with the K-means clustering and K-NN having already been implemented and tested preliminarily [91] and [113]. Additional work is required to validate the systems and assess their impacts.

- Implementing all the aforementioned DPR implementations on high-capacity state-of-the-art FPGAs as server solutions, and evaluate the overheads involved in actual reconfiguration compared with estimated results. One case study was presented in the K-means clustering chapter which indicated negligible overheads in terms of reconfiguration time when using ICAP to reconfigure the FPGA internally. However, additional work is required to confirm and generalise this finding with respect to variants of all three applications presented in this thesis and others.

- **K-NN classification**: Beside the common goals stated above, the K-NN classifier can be re-designed to adopt reconfigurable distance metrics using DPR. Comparisons with the implementations based on partially reconfiguring the K-NN core and implementations based on reconfiguring the complete K-NN core can be carried out to assess the performance of all approaches.

- **SVM classification:** Beside the common aims mentioned above, partially reconfigurable SVM training will be implemented on FPGA as this task take long time in GPPs and can benefit greatly from hardware acceleration. In addition, the SVM classifier will be implemented on FPGA using reconfigurable kernels, whereby the user can download the configuration file associated with a particular desired kernel e.g., linear, Gaussian, or others, and be able to partially modify the kernel configuration dynamically making use of DPR.

- Apply additional classifiers to the SVM/K-NN multi-classifier implementation, or add other supervised cores subject to compatibility with the current system in terms of area footprint, interface, and memory requirement.

- A long term goal is to implement the aforementioned un-supervised/ supervised methods in addition to other methods using a number of high end FPGAs and GPUs to be able to fairly asses the performance of the two technologies with respect to execution time, power, flexibility and cost.

- Another long term goal is to explore possible ways to exploit multiple technologies including GPPs, FPGAs and GPUs in a heterogeneous platform that combines the advantages of all of these technologies.

- The ultimate aim would be to build FPGA-based machines for server solutions and portable ones to be used in BCB labs.

References

[1] I. Rojas, H. Pomares, O. Valenzuela, and J. L. Bernier, "Applications in Bio-informatics and Biomedical Engineering," in *Bio-Inspired Systems: Computational and Ambient Intelligence* (Lecture Notes in Computaer Science Series), J. Cabestany et al. Ed., 1st ed., vol. 5517, Germany: Springer-Verlag, 2009, pp. 820-828. ISBN 978-3-642-02477-1.

[2] A. M. Lesk, *Introduction to Bioinformatics*, 1st ed., US: Oxford Univ. Press, 2002. ISBN 0-19-925196-7.

[3] A. H. C. van Kampen and A. J. G. Horrevoets, "The Role of Bioinformatics in Genomic Medicine" in *Cardiovascular Research: New Technologies, Methods, and Applications*, G. Pasterkamp and D. P. de Kleijn, Eds., US: Sringer, 2005, ch. 6, pp. 103-119. ISBN 0-38-7233288.

[4] J. Cohen, " Bioinformatics-An Introduction for Computer Scientists," *ACM Computing Surveys*, vol. 36, no. 2, pp. 122-158 , Jun. 2004.

[5] D. Feng, *Biomedical Information Techbology*, 1st ed., US: Elsevier, 2008. ISBN 978-0-12-373583-6.

[6] P. Baldi, and G. W. Hatfield, *DNA Microarrays and Gene Expression from Experiements to Data Analysis and Modeling*, 1st ed., New York, US: Cambridge Univ. Press, 2003. ISBN 978-0-521-17635-4.

[7] D. Stekel, *Microarray Bioinformatics*, 1st ed. Cambridge, U.K: Cambridge Univ. Press, 2003. ISBN 0-521-52587-X.

[8] K. Le Cao and G. McLachlan, "Statistical Analysis on Microarray Data: Selection of Gene Prognosis Signatures," in *Computational Biology: Issues and Application in Oncology* (Applies Bioinformatics and Biostatistics in Cancer Research Series), T. Pham, Ed., 1st ed., New York, US: Springer, 2009, ch. 3, pp. 55–75. ISBN 978-1-4419-0811-7.

[9] M. Akay, Ed., Genomics and Proteomics Engineering in Medicine and Biology (IEEE Press Series in Biomedical Engineering), 1st ed., US: John Wiley & Sons, 2007. ISBN:10 0-471-63181-7.

[10] A. K. Jain, M. N. Murty, and P.J. Flynn, "Data Clustering: A Review", *ACM Computing Surveys*, vol. 31, no. 3, pp.264-323, Sep.1999.

[11] A. K. Jain, R. P. W. Duin, and J. Mao, "Statistical Pattern Recognition: A Review," *IEEE Trans. Pattern Anal. Mach. Intell*, vol. 22, no.1, pp. 4-37, Jan. 2000.

[12] H. Causton, J. Quackenbush, and A. Brazma, *Microarrays Gene Expression: A Beginner's Guide*, 1st ed., U.K. : Wiley-Blackwell, 2003. ISBN 978-1405-106825.

[13] M. Madan Babu, "Introduction to Microarray Data Analysis," *in Computational Genomics: Theory and Application*, R. Grant, Ed., U.K. : Horizon Press, 2004, ch.11, pp. 225-249. ISBN 978-1-904933-01-4.

[14] http://en.wikipedia.org/wiki/File:Microarray2.gif. [online accessed June, 2012].

[15] M. F. Ochs and A. K. Godwin, "Microarrays in Cancer: Research and Applications," *J. BioTechniques*, vol. 34, Suppl, pp. 4-15, Mar. 2003. [online at: http://www.biotechniques.com/multimedia/archive/00072/Mar03Ochs_72032a.pdf. accessed june, 2012].

[16] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein, "Cluster Analysis and display of genome-wide expresison patterns," *Proc. Natl. acad. Sci*, vol. 95, pp. 14863-14868, Dec. 1998.

[17] J. Quackenbush, "Computational Analysis of Microarray Data" *J. Nature*, vol. 2, pp. 418-427, Jun. 2001.

[18] P. F. Macgrefor and J. A. Squire, "Application of Microarray to the Analysis of Gene Expression in Cancer," *J. Clinical Chemistry*, vol. 48, no. 8, pp. 1170–1177, Aug. 2002.

[19] S. K. Gruvberger-Saal, H. E. Cunliffe, K. M. Carr, I. A. Hedenfalk, "Microarray in Beast Cancer Research and Clincal Practice- The Future Lies Ahead," *J. Endocrine-Related Cancer*, vol. 13, no.4 , pp. 1017-1031, Dec. 2006.

[20] L. D. Miller, and E. T. Lie, "Expression Genomics in Breast Cancer Research: Microarrays at the Crossroads of Biology and Medecine," J. *Breast Cancer Res*., vol. 9, no. 2, pp. 209-2019, Mar. 2007.

[21] M. Diamandis, N. M. A. White, and G. M. Tousef, "Personalised Medecine: Making a New Epoch in Cancer Patient Management," *J. Mol. Cancer Res*., vol. 8, no. 9, pp. 1175-1187, Sep. 2010.

[22] D. Brennan et al., "Application of DNA Microarray Technology in Determining Breast Cancer Prognosis and Therapeutic Response," *J. Expret. Opin. Biol. Ther*., vol. 5, no. 8, pp. 1069–1083, Aug. 2005.

[23] http://www.agendia.com/pages/coloprint/173.php. [online accessed June, 2012].

[24] T. R. Golub et al., "Molecular Classification of Cancer: Class Discovery and Class Prediction by Gene Expression Monitoring," *J. Science*, vol. 286, no. 5439, pp. 531–537, Oct. 1999.

[25] K. Benkrid, A. Akoglu, C. Ling, Y. Song, X. Tian, and Y. Lue, "High Perfomance Biological Pairwise Sequence Alignment: FPGA vs. GPU vs. CellBE vs. GPP," *Int. J. of Reconfig. Comput*., Apr. 2012.

[26] S. Kasap and K. Benkrid, "High Performance Phylogenetic Analysis with Maximum Parsimony on Reconfigurable Hardware", *IEEE Trans. on Very Large Scale Integr. (VLSI) Syst.*, vol. 99, pp. 1-13, Feb. 2010.

[27] S. Kasap, K. Benkrid, and Y. Liu, "Design and Implementation of an FPGA-based Core for Gapped BLAST Sequence Alignment with the Two-Hit Method," *IAENG J. Eng. Letters, Special Issue on High Performance Reconfigurable Systems*, vol. 16, no. 3, pp. 443-452, Aug. 2008.

[28] J. Nurmi, *Processor Design System-On-Chip Computing for ASICs and FPGAs*, 1st ed. The Netherlands: Springer, 2010. ISBN 978-90-481-7385-3.

[29] P. J. Ashenden, *Digital Design An Embedded Systems Approach Using Veilog*, 1st ed. USA: Elsevier, 2008. ISBN 978-0-12-369527-7.

[30] D. Buell, T. El-Ghazawi, K. Gaj, and V. Kindratenko, "High Performance Reconfigurable Computing," *IEEE Computer*, vol. 40, no. 3, pp. 23-27, Mar. 2007.

[31] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaji, V. Kindratenko, and D. Buell, "The Promise of High-performance Reconfigurable Computing," *IEEE Computer*, vol. 41, no. 2, pp. 69-76, Feb. 2008.

[32] Xilinx Inc., "Virtex-4 User Guide ug070", v. 2.6, 2008. [online at http://www.xilinx.com. aceesed April 20, 2012].

[33] C. Maxfield, *FPGAs: World Class Designs*, 1st ed. U.K: Newnes, 2009. ISBN 978-1856176217.

[34] Xilinx Inc., "Virtex-5 Platform FPGA Family Technical Backgrounder," May 2006. [online at http://www.xilinx.com/company/press/kits/v5/v5backgrounder.pdf. accessed June 13, 2012].

[35] Xilinx Inc., "Xilinx-7 Series FPGAs Press Kit Backgrounder," Jun. 21, 2010 [online at http://www.xilinx.com/company/press/kits/v7v7ackgrounder.pdf. accessed June 13, 2012].

[36] R. Dubey, *Introduction to Embedded System Design Using Field Programmable Gate Arrays*, 1st ed. London: Springer-Verlag, 2009. ISBN 978-1-84882-015-9.

[37] Xilinx Inc., "Partial Reconfiguration Guide ug73," v. 12.3, 2010. [online at http://www.xilinx.com. aceesed April 20, 2012].

[38] Xilinx Inc., "Xilinx Partial Reconfiguration Guide ug702," v. 12.3, p. 103, 2010. [online at http://www.xilinx.com. aceesed April 20, 2012].

[39] Xilinx Inc., "Hierarchical Design Methodology Guide ug748," v. 13.3, 2011. [online at http://www.xilinx.com. aceesed April 20, 2012].

[40] D. Dye, "Partial Reconfiguration of Virtex FPGA in ISE 12," Xilinx White Paper WP 374, July 23, 2010. [online at http://www.xilinx.com. aceesed April 20, 2012].

[41] Xilinx Inc., "Xilinx ML401/ML402/ML403 Evaluation Platform User Guide ug080," v. 2.5, 2006. [online at http://www.xilinx.com. aceesed April 20, 2012].

[42] B. Andreopoulos, A. An, X. Wang, and M. Schroeder, "A Roadmap of Clustering Algorithms: Finding a Match for a Biomedical Application," *J. Briefings in Bioinformatics*, vol. 10, no. 3, pp. 297-31, Feb. 2009.

[43] G. Thijs et al., "Gene Regulation Bioinformatics of Microarray Data," in *Genomics and Proteomics Engineering in Medicine and Biology* (IEEE Press Series in Biomedical Engineering), M. Akay, Ed., US: A John Wiley & Sons, 2007, ch. 3, pp. 55-67. ISBN:10 0-471-63181-7.

[44] K. Benkrid, Y Liu, and A. Benkrid, "A Highly Parameterised and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment," *IEEE Trans. on Very Large Scale Integr. Syst. (VLSI Systems)*, vol. 17, no. 4, pp. 561-570, Apr. 2009.

[45] M. C. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. V. Court, "Single Pass streaming BLAST on FPGAs," *J. Parallel Computing*, vol. 33, no. 10-11, pp. 741-756, 2007.

[46] N. Alachiotis, E. Sotiriades, A. Dollas, and A. Stamatakis, "A Reconfigurable Architecture for Phylogenetic Likelihood Function," in *Proc. of the 2009 IEEE Conf. on Field Programmable Logic and Applications(FPL)*, Prague, Czech Republic, Aug. 31- Sep. 2, 2009, pp. 674-678.

[47] M. C. P. de Souto, S. C. M. Silva, V. G. Bittencourt, and D. S. A. de Araujo, "Cluster Ensemble for Gene Expression Microarray Data," in *Proc. of the 2005 IEEE Int. Joint Conf. Neural Networks (IJCNN)*, vol. 1, Montreal, Canada, Jul.31-Aug. 4, 2005, pp. 487- 492.

[48] C. Chatfield and A. J. Collins, *Intruduction to Multivariate Analysis*, 1st ed., London, UK: Chapman and Hall, 1980. ISBN 978-04121-6-0400.

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

[50] M. Estlick, M. Leeser, J. Theiler, and J. Szymanski, "Algorithmic Transformations in the Implementation of K-means Clustering on Reconfigurable Hardware," in *Proc. of the 2001 ACM/SIGDA ninth Int. Symp. on Field Programmable Gate Arrays*, Monterey, CA, US, Feb. 11-13, 2001, pp. 103-110.

[51] J. Theiler, M. Leeser, M. Estlick, and J. Szymanski, "Design Issues for Hardware Implementation of an Algorithm for Segmenting Hyperspectral Imagery," in *Proc. of the VI. Imaging Spectrometry (SPIE)*, vol. 4132, pp. 99-106, 2000.

[52] A. K. Jain, "Data Clustering: 50 Years Beyon K-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651-666, Jun. 2010.

[53] R. Farivar, D. Rebolledo, E. Chan, and R. H. Cambell, "A Parallel Implementation of K-means Clustering on GPUs," in *Proc. of the 2008 Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 08)*, Las Vegas, NV, Jul. 14-17, 2008, pp. 340-345.

[54] M. Gokhale, J. Frigo, K. McCabe, J. Theiler, C. Wolinski, and D. Lavenier, "Experience with a Hybrid Processor: K-means Clustering," *J. Supercomputing*, vol. 26, pp. 131-148, Aug. 2003.

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

[56] M. Leeser, P. Belanovic, M. Estlick, M. Gokhale, J. J. Szymanski, and J. Theiler, "Applying Reconfigurable Hardware to the Analysis of Multispectral and Hyperspectral Imagery," in *Proc. of SPIE*, 2002, pp.4480.

[57] M. Leeser, P. Belanovic, M. Estlick, M. Gokhale, J. J. Szymanski, and J. Theiler, "Parameterised K-means Clustering for Rapid Hardware Development to Accelerate Analysis of Satellite Sata," in *Proc. of the 5th Annu. Workshop on High Performance Embedded Computing (HPEC) Workshop*, MIT Lincoln Laboratory, Sep. 25-27, 2001.

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

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

[60] A. Gda.S. Filho et al., "Hyperspectral Images Clustering on Reconfigurable Hardware using the K-means Algorithm," in *Proc. of the 16th Symp. on Integrated Circuits and Syst. Design (SBCCI'03)*, Sao Paulo, Brazil, Sep. 8-11, 2003, pp. 99-104.

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

[62] X. Wang and M. Leeser, "K-means Clustering for Multispectral Images using Floating-Point Divide," *in Proc. of the 15th Annu. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM*), Napa, CA, Apr. 23-25, 2007, pp. 151-159.

[63] G. A. Covington, L. G. Comstock, A. A. Levine, J. W. Lockwood, and Y. H. Cho, "High Speed Document Clustering In Reconfigurable Hardware," in *Proc. of the 16th Annu. Conf. on Field Programmable Logic and Applications (FPL)*, Madrid, Spain, Aug. 28-30, 2006, pp. 411-417.

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

[65] M. Leeser, M. Estlick, N. Kitaryeva, J. Theiler, and J.J. Szymanski, "Applying Reconfigurable Hardware for Multispectral Imagery," in *Proc. of the Annu. Workshop. on High Performance Embedded Computing (HPEC) Workshop*, Boston, MA, Sep, 2000.

[66] 14 hr Microarray Yeast datasets. [Online, at http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE28. accessed April 20, 2012].

[67] Matlab Bioinformatics Toolbox. [Online, at http://www.mathworks.com/products/bioinfo. accessed April 20,2012].

[68] E. L. Oberstar, "Fixed-Point Representation & Fractional Math," Oberstar Consulting, Rev. 1.2, 2007. [Online at http://www.superkits.net/whitepapers.htm. accessed April 19, 2012].

[69] R. Narayanan, D. Honbo, J. Zambreno, G. Memik, and A. Choudhary, "An FPGA Implementation of Decision Tree Classification," in *Proc. of IEEE Int. Conf. on Design, Automation and Test in Europe (DATE),* Nice, France, Apr. 16-20, 2007, pp. 1-6.

[70] A. Choudhary, R. Narayanan, B. Ozisikyilmaz, G. Memik, J. Zambreno, and J. Pisharath, "Optimizing Data Mining Workloads using Hardware Accelerators," in *Proc. of the 10th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Phoenix, AZ, Feb. 11, 2007.

[71] A. Nayak, M. Haldar, A. Choudhar, and P. Banerjee, "Precision and Error Analysis of MATLAB Applications During Automated Hardware Synthesis for FPGAS," *in Proc. of the Conf. on Design, Automation, and Test in Europe*, Grenoble, France, Mar. 13-16, 2001, pp. 722-728.

[72] H. Hussain, K. Benkrid, H. Seker, and A. Erdogan, "FPGA Implementation of K-means Algorithm for Bioinformatics Application: An Accelerated Approach to Clustering Microarray Data," in *Proc. of the 2011 NASA/ESA Conf. on Adaptive Hardware and Syst. (AHS)*, San Diego, CA, Jun. 6-9, 2011, pp. 248-255.

[73] H. Hussain, K. Benkrid, H. Seker, and A. Erdogan, "Highly Parametrized K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs," in *Proc. of the Int. Conf. on Reconfigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, Nov. 30- Dec. 2, 2011, pp.475–480.

[74] S. A. Shalom, M. Dash, and M. Tue, "Efficient K-means Clustering Using Accelerated Graphics Processors," in *Proc. of the 10th Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK '08)*, Turin, Italy, Sep. 1-5, 2008, pp. 166–175.

[75] G. Karch, "GPU Based Acceleration of Selected Clustering Techniques," M.S. thesis, Dept. Elect. and Comp. Science, Silesian University of Technology in Gliwice, Silesia, Poland, 2010.

[76] A. Choudhary, D. Honbo, P. Kumar, B. Ozisikyilmaz, S. Misra, and G. Memik, "Accelerating Data Mining Workloads: Current Approaches and Future Challenges in System Architecture Design," *Wiley Interdisciplinary Reviews: WIREs Data Mining and Knowledge Discovery*, vol. 1, pp. 41-54, Jan. 2011.

[77] S. A. Shalom, M. Dash, and M. Tue, "GPU-based Fast K-means Clustering of Gene Expression Profiles," *Presented at the 12th Annu. Int. Conf. on Research in Computational Molecular Biology (RECOMB)*, Singaphore, 2008, P66 (poster).

[78] Nvidia GEForce 9600 GT datasheet. [Online at http://www.nvidia.com/object/product_geforce_9600gt_us.html. accessed Dec.2011].

[79] S. Kentaro, T. Nishikawa, T. Aoki, and S. Yamamoto, "Evaluating Power and Energy Consumption of FPGA-Based Custom Computing Machines for Scientific Floating-Point Computation," in *Proc. of the Int. Conf. on Field-Programmable Technology (ICFPT*), Taipei, Taiwan, Dec. 7-10, 2008, pp. 301-304.

[80] M. Martin-Merino and J. De Las Rivas, *Advances in Intelligent Data Analysis VIII* (Lecture Notes in Computer Science), A. Niall et al., Eds., Berlin Heidelberg, Germany: Springer, 2009, vol. 5772, pp. 107-118.

[81] A. Statnikov et al., "A Comprehensive Evaluation of Multicategory Classification Methods for Microarray Gene Expression Cancer Diagnosis*," Bioinformatics*, vol. 21, no. 5, pp. 631–643, Sep. 2004.

[82] R. M Parry et al., "K- Nearest Neighbor Models for Microarray Gene Expression Analysis and Clinical Outcome Prediction," *Pharmacogenomics J.*, vol. 10, no. 4, pp. 292-309, Aug 2010.

[83] E. Manolakos and I. Stamoulias, "IP-cores Design for the KNN Classifier, " in *Proc. of the IEEE Int. Symps. on Circuits and Systems (ISCAS)*, Paris, France, May 30- Jun. 2, 2010, pp. 4133-4136.

[84] E. Manolakos and I. Stamoulias, "Flexible IP Cores for the K-NN Classification Problem and their FPGA Implementation," in *Proc. of IEEE Int. Symp. on Parallel and Distributed Processing Workshops and phd Forum*, Atlanta, GA, Apr. 13-23, 2010, PP. 1-4.

[85] Garcia, V., Debreuve, E. and Barlaud, M, "Fast k Nearest Neighbor Search Using GPU," *in Proc. of IEEE Computer Society Conf. on Computer Vision and Pattern Recognition Workshops (CVPR)*, Anchorage, Alaska, Jun. 24-26, 2008, pp. 1-6.

[86] M. A. Tahir and A. Bouridane, "An FPGA Based Coprocessor for Cancer Classification Using Nearest Neighbor Classifier," in *Proc. of IEEE Int. Conf. on Acoustic, Speech and Signal Processing (ICASSP)*, Toulouse, France, May.14-19, 2006, pp. 1012-1015.

[87] M. Murugappan, "Human Emotion Classification using Wavelet Transform and KNN," in *Proc. of the 2011 Int. Conf. on Pattern Analysis and Intelligent Robotics (ICPAIR)*, vol. 1, Putrajaya, Malasia, Jun.28-29, 20122, pp. 148-153.

[88] M. A. Tahir and J. Smith, "Improving Nearest Neighbor Classifier Using Tabu Search and Ensemble Distance Metrics," in *Proc. of IEEE 2006 Int. Conf. Data Mining*, Hong Kong, China, Dec. 18-22, 2006, pp. 1086–1090.

[89] T. G. Dietterich, "Ensemble Methods in Machine Learning," in *Multiple Classifier Systems, ser. Lecture Note in Computer Science*, vol. 1857, 2000, pp. 1-15.

[90] M. C. de Souto, S.C.M Silva, V.G. Bittencourt, and D.S.A. de Araujo, "Cluster Ensemble for Gene Expression Microarray Data," in *Proc. of 2005 IEEE Int. Joint Conf. Neural Networks (IJCNN)*, vol. 1, Monteral, Canada, Jul.31- Aug.4, 2005, pp. 487- 492.

[91] C. Hong, K. Benkrid, X. Iturbe, and H. Hussain, "Efficient Run-time System Support for High Performance Reliable Reconfigurable Systems," *Int. Conf. on Reconfigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, Dec. 5-7, 2012. (in Press).

[92] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and other Kernel-Based Learning Methods*, 1st ed., USA: Campridge Univeristy Press, 2000. ISBN 0-521-78019-5.

[93] S. Mukherjee, "Classifying Microarray Data Using Support Vector Machines," in *A Practical Approach to Microrray Data Analysis*, D. Berrar et al., Eds., 1st ed., USA: Springer, 2009, ch. 9, pp. 1-19. ISBN 1441912266.

[94] S. Cho and H. Won, "Machine Learning in DNA Microarray Analysis for Cancer Classification," in *Proc. of the First-Asia-Pacific Conf. on Bioinformatics*, Seoul, Korea, vol. 19, Feb. 4-7, 2003, pp. 189-198.

[95] V. Vapnik, *The Nature of Statistical Learning Theory*, 2nd ed., New York, US: Springer-Verlag, 2000. ISBN 0-387-98780-0.

[96] M. P. S Brown et al., "Knowledge-Based Analysis of Microarray Gene Expression Data Using Support Vector Machines," in *Proc. of Natl. Acad. Sci*, USA, vol. 97, no. 1, pp. 262-267, Jan. 4, 2000.

[97] D. Anguita, "A Digital Architecture for Support Vector Machines: Theory, Algorithm, and FPGA Implementation," *IEEE Trans. Neural Netw.*, vol. 12, no. 5, pp. 993-1009, Sep. 2003.

[98] D. Anguita, L. Carlino, A. Ghio, and S. Ridella, "A FPGA Core Generator for Embedded Classfication Systems," *J. Circuits, Syst., and Comput*, vol. 20 , no. 2 , pp. 263-282, Apr. 2011.

[99] J. Gomes Filho, M. Raffo, M. Strum, and W. Jiangg Chau, "A General-purpose Dynamically Reconfigurable SVM," in *Proc. of the VI Southern Programmable Logic Conference (SPL)*, Ipojuca, Brazil, Mar. 24-26, 2010, pp.107-112.

[100] J. Woo Wee and C. Ho Lee, "Concurrent Support Vector Machine Processor for Disease Diagnosis," in *Neural Information Processing* (Lecture Notes in Computer Science) Pal et al., Eds., Berlin Heidelberg: Springer-Verlag, 2004, vol. 3316, pp.1129-1134. ISBN 978-3-540-23931-4.

[101] O. Pina-Ramirez, R. Valdes-Cristerna, and O. Yanez-Suarez, "An FPGA Implementation of Linear Kernel Support Vector Machines," *in Proc. of the 2006 Int. Conf. on Reconfigurable Computing and FPGAs* (ReConFig), San Luis Potosi, Mexico, Sep. 27-29, 2006, pp. 1-6.

[102] M. Papadonikolakis and C.-S. Bouganis, "Effeceicnt Mapping of Gilbert's Algorithms for SVM Training on Large-Scale Classification Problems," *in Proc. of the 2008 Int. Conf. on Field Programmable Logic and Application (FPL)*, Heidelberg, Germany, Sep. 8-10, 2008, pp. 385-390.

[103] S. Cadambi et al., "A Massively Parallel FPGA-Based Coprocessor for Support Vector Machines," in *Proc. of the 17th IEEE Symps. on Field Programmable Custom Computing Machine*s (FCCM), Napa, CA, Apr. 5-9, 2009, pp. 115-122.

[104] C. Kyrkou and T. Theocharides, "SCoPE: Towards a Systolic Array for SVM Object Detection," *IEEE Embedded Systems Letters*, vol. 1, no. 2, pp. 46-49, Aug. 2009.

[105] M. Papadonikolakis and C. Bouganis, "A Novel FPGA-Based SVM Classifier," in *Proc. of the 2010 Int. Conf. on Field-Programmable Technology (FPT)*, Beijin, China, Dec. 8-10, 2010, pp. 283-286.

[106] S. Kim, S. Lee, K. Min, and K. Cho, "Design of Support Vector Macine Circuit for Real-time Classification," in *Proc. of the 13th Int. Symps. On Integrated Circuits (ISIC)*, Singapore, Dec. 12-14, 2011, pp. 384-387.

[107] M. Papadonikolakis and C. S. Bouganis, "Novel Cascade FPGA Accelerator for Support Vector Machines Classification," IEEE Trans. Neural Netw. Learn. Syst., vol. 23, no. 7, pp.1040-1052, Jul. 2012.

[108] R. A. Patil, G. Gupta, V. Sahula, and A. S. Mandal, "Power Aware Hardware Prototyping of Multiclass SVM Classifier Through Reconfiguration," in *Proc. of the 25th Int. Conf. on VLSI Design (VLSID)*, Hederbad, India, Jan. 7-11, 2012, pp. 62-67.

[109] C. Chang and C. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol. (TIST)*, vol. 2, no. 3, Article 27, pp. 1-27, May 2011.

[110] Xilinx Inc., "XtremeDSP for Virtex-4 FPGAs User Guide ug73," v. 2.7, pp. 12-24, 2008. [Online at http://www.xilinx.com. aceesed May 29, 2012].

[111] J. Cong et al., "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473-491, Apr. 2011.

[112] Berkely Design Technology Inc., "The Independent Evaluation of: High-Level Synthesis Tools for Xilinx FPGAs," 2010. [Online at http://www.bdti.com/MyBDTI/pubs/Xilinx_hlstcp.pdf. accessed July 5, 2012].

[113] H. Hussain, K. Benkrid, C. Hong, and H. Seker, "Novel Dynamic Partial Reconfiguration Implementation of K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs," *Int. J. of Reconfig. Comput.*(in Press).