# Optimized Hardware Accelerators for Data Mining Applications

by

Awos Kanan
B.Sc., Jordan University of Science and Technology, 2003
M.Sc., Jordan University of Science and Technology, 2006

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

© Awos Kanan, 2018
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by
photocopying or other means, without the permission of the author.

Optimized Hardware Accelerators for Data Mining Applications

by

Awos Kanan
B.Sc., Jordan University of Science and Technology, 2003
M.Sc., Jordan University of Science and Technology, 2006

Supervisory Committee

---

Dr. Fayez Gebali, Supervisor
(Department of Electrical and Computer Engineering)

---

Dr. Atef Ibrahim, Co-Supervisor
(Department of Electrical and Computer Engineering)

---

Dr. Brad Buckham, Outside Member
(Deptartment of Mechanical Engineering)

**Supervisory Committee**

Dr. Fayez Gebali, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Atef Ibrahim, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. Brad Buckham, Outside Member
(Deptartment of Mechanical Engineering)

## ABSTRACT

Data mining plays an important role in a variety of fields including bioinformatics, multimedia, business intelligence, marketing, and medical diagnosis. Analysis of today's huge and complex data involves several data mining algorithms including clustering and classification. The computational complexity of machine learning and data mining algorithms, that are frequently used in today's applications such as embedded systems, makes the design of efficient hardware architectures for these algorithms a challenging issue for the development of such systems. The aim of this work is to optimize the performance of hardware acceleration for data mining applications in terms of speed and area. Most of the previous accelerator architectures proposed in the literature have been obtained using ad hoc techniques that do not allow for design space exploration, some did not consider the size (number of samples) and dimensionality (number of features in each sample) of the datasets. To obtain practical architectures that are amenable for hardware implementation, size and dimensionality of input datasets are taken into consideration in this work. For one-dimensional data, algorithm-level optimizations are investigated to design a fast and area-efficient hardware accelerator for clustering one-dimensional datasets using the well-known K-Means clustering algorithm. Experimental results show that the optimizations adopted in the proposed architecture result in faster convergence of the algorithm using

less hardware resources while maintaining the quality of clustering results. The computation of similarity distance matrices is one of the computational kernels that are generally required by several machine learning and data mining algorithms to measure the degree of similarity between data samples. For these algorithms, distance calculation is considered a computationally intensive task that accounts for a significant portion of the processing time. A systematic methodology is presented to explore the design space of 2-D and 1-D processor array architectures for similarity distance computation involved in processing datasets of different sizes and dimensions. Six 2-D and six 1-D processor array architectures are developed systematically using linear scheduling and projection operations. The obtained architectures are classified based on the size and dimensionality of input datasets, analyzed in terms of speed and area, and compared with previous architectures in the literature. Motivated by the necessity to accommodate large-scale and high-dimensional data, nonlinear scheduling and projection operations are finally introduced to design a scalable processor array architecture for the computation of similarity distance matrices. Implementation results of the proposed architecture show improved compromise between area and speed. Moreover, it scales better for large and high-dimensional datasets since the architecture is fully parameterized and only has to deal with one data dimension in each time step.

# Contents

**Supervisory Committee** ii

**Abstract** iii

**Table of Contents** v

**List of Tables** viii

**List of Figures** ix

**Acknowledgements** xi

**Dedication** xii

**List of Abbreviations** xiii

**1 Introduction** **1**

1.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.1.1 Machine Learning and Data Mining . . . . . . . . . . . . . . . . 2

1.1.2 Parallel Computing . . . . . . . . . . . . . . . . . . . . . . . 2

1.2 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.3 Research Objectives . . . . . . . . . . . . . . . . . . . . . . . . 6

1.4 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.4.1 Efficient Hardware Implementation of K-Means Clustering for 1-D Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.4.2 2-D Processor Array Architectures for Similarity Distance Computation . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.4.3 Linear Processor Array Architectures for Similarity Distance Computation . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.4.4 Scalable and Parameterizable Processor Array Architecture for Similarity Distance Computation. . . . . . . . . . . . . . . . . . . . . . 8
1.5 Dissertation Organization . . . . . . . . . . . . . . . . . . . . . . . . . . 8

**2 Efficient Hardware Implementation of K-Means Clustering for 1-D Data 10**
2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 K-Means Clustering Algorithm . . . . . . . . . . . . . . . . . . . . . . 11
2.3 Conventional Implementation of the K-Means Algorithm . . . . . . . . . . 14
2.4 The Proposed Approach . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.4.1 Source Cluster . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.4.2 Destination Cluster . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.5 Hardware Design and Implementation . . . . . . . . . . . . . . . . . . . 18
2.5.1 Distance Calculation Unit . . . . . . . . . . . . . . . . . . . . . . 20
2.5.2 Minimum Distance unit . . . . . . . . . . . . . . . . . . . . . . . 20
2.5.3 Count Unit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.5.4 Centroids Update Unit . . . . . . . . . . . . . . . . . . . . . . . . 22
2.6 Results and Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . 24

**3 2-D Processor Array Architectures for Similarity Distance Computation 30**
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.2 Problem Formulation . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.3 A Systematic Methodology for Processor Array Design . . . . . . . . . . . 32
3.3.1 The 3-D Computation Domain and Domain Boundaries . . . . . . . 33
3.3.2 The Dependence Matrices . . . . . . . . . . . . . . . . . . . . . . 34
3.3.3 Data Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.3.4 Projection Operation . . . . . . . . . . . . . . . . . . . . . . . . 39
3.4 Design Space Exploration . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.4.1 Case 1: When $N \gg K, M$ . . . . . . . . . . . . . . . . . . . . 41
3.4.2 Case 2: When $M \gg K, N$ . . . . . . . . . . . . . . . . . . . . 46
3.4.3 Case 3: When $K \gg M, N$ . . . . . . . . . . . . . . . . . . . . 53
3.5 Design Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
3.6 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

**4 Linear Processor Array Architectures for Similarity Distance Computation 58**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.2 Data Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

4.2.1 Calculation of the first scheduling vector $\mathbf{s_1}$ . . . . . . . . . . . . . 59
4.2.2 Calculation of the remaining scheduling vectors . . . . . . . . . . . 61
4.3 Projection Operation . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.4 Design Space Exploration . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.4.1 Design #1: using $\mathbf{s_1}$ = [0 1 $M$] and $\mathbf{P_1}$ = [1 0 0] . . . . . . . . . . 66
4.4.2 Design #2: using $\mathbf{s_2}$ = [0 $N$ 1] and $\mathbf{P_2}$ = [1 0 0] . . . . . . . . . . 70
4.4.3 Design #3: using $\mathbf{s_3}$ =[1 0 $K$] and $\mathbf{P_3}$ =[0 1 0] . . . . . . . . . . 70
4.4.4 Design #4: using $\mathbf{s_4}$ =[$N$ 0 1] and $\mathbf{P_4}$ =[0 1 0] . . . . . . . . . . 70
4.4.5 Design #5: using $\mathbf{s_5}$ = [$M$ 1 0] and $\mathbf{P_5}$ =[0 0 1] . . . . . . . . . . 72
4.4.6 Design #6: using $\mathbf{s_6}$ = [1 $K$ 0] and $\mathbf{P_6}$ = [0 0 1] . . . . . . . . . . 72
4.5 Comparison and Results . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.5.1 Design Comparison . . . . . . . . . . . . . . . . . . . . . . . . 73
4.5.2 Implementation Results . . . . . . . . . . . . . . . . . . . . . . 75

**5 Scalable and Parameterizable Processor Array Architecture for Similarity Distance Computation 79**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
5.2 Methodology of processor array design . . . . . . . . . . . . . . . . . . 80
5.2.1 Data Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.2.2 Projection Operation . . . . . . . . . . . . . . . . . . . . . . . . 82
5.3 The proposed processor array architecture . . . . . . . . . . . . . . . . . 82
5.4 Design Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.5 Implementation Results . . . . . . . . . . . . . . . . . . . . . . . . . . 84

**6 Conclusions and Future Work 89**
6.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
6.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

**A Publications 92**

**Bibliography 93**

# List of Tables

Table 2.1 Implementation results for clustering one-dimensional dataset into 8 clusters on Xilinx XC4VLX25. . . . . . . . . . . . . . . . . . . . . . 24
Table 2.2 Execution times of GPU, conventional FPGA, and proposed FPGA implementations of the K-Means algorithm. . . . . . . . . . . . . . . 27
Table 2.3 Area occupied by the Divider and Centroids Update units in the conventional and proposed designs, Respectively. . . . . . . . . . . . . . 28

Table 3.1 Design Space Exploration of 2-D Processor Array Architectures for Similarity Distance Computation. . . . . . . . . . . . . . . . . . . . . . 40
Table 3.2 Circuit and time complexities of the obtained processor array architectures. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

Table 4.1 Possible projection directions and associated projection matrices . . . 67
Table 4.2 Design comparison. . . . . . . . . . . . . . . . . . . . . . . . . 76
Table 4.3 Implementation results. . . . . . . . . . . . . . . . . . . . . . . 78

Table 5.1 Design Comparison. . . . . . . . . . . . . . . . . . . . . . . . . 85
Table 5.2 Implementation results of the proposed architecture with $K = 26$, $w_k$= 13 ($K/2$), and different values of $w_n$. . . . . . . . . . . . . . . . 86
Table 5.3 Implementation results of previous architectures and the proposed architecture ($w_n = 2$, $w_k = K/2$) for different values of $K$. . . . . . . . 88

# List of Figures


Figure 1.1 Shared-Memory Multiprocessor Architecture [1]. . . . . . . . . . . . . 4
Figure 1.2 Distributed-Memory Multiprocessor Architecture [1]. . . . . . . . . 5

Figure 2.1 Functional blocks of the conventional K-Means algorithm hardware architecture. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
Figure 2.2 Functional blocks of the proposed K-Means algorithm hardware architecture. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
Figure 2.3 Distance Calculation unit. . . . . . . . . . . . . . . . . . . . . . . . 20
Figure 2.4 Minimum Distance unit. . . . . . . . . . . . . . . . . . . . . . . . . 21
Figure 2.5 Count unit. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
Figure 2.6 Centroids Update unit. . . . . . . . . . . . . . . . . . . . . . . . . 23
Figure 2.7 Speedup of the proposed design over the conventional design calculated as the ratio of number of iterations required to converge. . . . . 25
Figure 2.8 Speedup of conventional [2] and proposed FPGA implementations of the K-Means algorithm over GPU implementation [3]. . . . . . . . 28
Figure 2.9 Total Within-Cluster Variance for the approximated, shift-based, and original, division-based, Centroids Update units. . . . . . . . . . . . 29

Figure 3.1 The 3-D computation domain $\mathcal{D}$ for the distance calculation algorithm. 34
Figure 3.2 The broadcast subdomains for algorithm variables. . . . . . . . . . . 36
Figure 3.3 Equitemporal zones using linear scheduling and scheduling vector is [0 1 0]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
Figure 3.4 Processor array architecture for Design #1 when $K = 3$ and $M = 4$. . 44
Figure 3.5 Processing element for Design #1 in Figure 3.4. . . . . . . . . . . . . 45
Figure 3.6 Processor array architecture for Design #2 when $K = 3$ and $M = 4$. . 47
Figure 3.7 Processing element for Design #3 in Figure 3.8. . . . . . . . . . . . . 49
Figure 3.8 Processor array architecture for Design #3 when $K = 3$ and $N = 4$. . 49
Figure 3.9 Processor array architecture for Design #4 when $K = 3$ and $N = 4$. . 50
Figure 3.10 Processor array architecture for Design #5 when $K = 3$ and $N = 4$. . 51

Figure 3.11 Processor array architecture for Design #6 when $K = 3$ and $N = 4$. . . 52
Figure 3.12 Analytical speedups of Case 1 architectures for $K = 64$ and $M = 50$. 55
Figure 3.13 Analytical speedups of Case 2 architectures for $K = 64$ and $N = 50$. 56

Figure 4.1 Equitemporal zones for scheduling vector $\mathbf{s_1}$ . . . . . . . . . . . . . 59
Figure 4.2 Equitemporal zones for scheduling vector $\mathbf{s_2}$ . . . . . . . . . . . . . 61
Figure 4.3 Equitemporal zones for scheduling vector $\mathbf{s_3}$ . . . . . . . . . . . . . 62
Figure 4.4 Equitemporal zones for scheduling vector $\mathbf{s_4}$ . . . . . . . . . . . . . 63
Figure 4.5 Equitemporal zones for scheduling vector $\mathbf{s_5}$ . . . . . . . . . . . . . 64
Figure 4.6 Equitemporal zones for scheduling vector $\mathbf{s_6}$ . . . . . . . . . . . . . 65
Figure 4.7 Processor array architecture for Design #1 . . . . . . . . . . . . . . . 69
Figure 4.8 Processing element for Design #1 in Figure 4.7 . . . . . . . . . . . . 69
Figure 4.9 Processing element for Design #3 . . . . . . . . . . . . . . . . . . . . 71
Figure 4.10 Processor array architecture for Design #3 with $M$=4. . . . . . . . . . 71
Figure 4.11 Processor array architecture for Design #5 . . . . . . . . . . . . . . . 72
Figure 4.12 Inputs timing for Design of [4] with $K$=4 . . . . . . . . . . . . . . . 74
Figure 4.13 Area-delay product for different values of $\mathbf{K}$ . . . . . . . . . . . . . 77

Figure 5.1 2-D equitemporal zones using nonlinear scheduling function (5.1), K=6, M=3, $w_k$= 3, and $w_n$= 2. . . . . . . . . . . . . . . . . . . . . . 81
Figure 5.2 Proposed processor array architecture with $w_k$= 3, and $w_n$= 2. . . . . 83
Figure 5.3 Processing element structure for the proposed architecture. . . . . . . 83
Figure 5.4 Area-Delay Product. . . . . . . . . . . . . . . . . . . . . . . . . . . 87

# ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me conduct my research and write this dissertation. First and foremost, I would like to thank my supervisor Dr. Fayez Gebali for his guidance and support throughout my PhD. The research style and methodology I have learned from him will benefit me for the rest of my life.

I would like to thank my co-supervisor Dr. Atef Ibrahim for his scholarly advice, inspiration, and support to complete my PhD dissertation work.

I am also great thankful to Dr. Kin Fun Li for his support, efforts, and contributions to this work. I would also like to acknowledge the advice and support I received from my supervisory committee member Dr. Brad Buckham to make my dissertation complete and resourceful.

I also appreciate my beloved parents, sister, brothers, especially my wife and my lovely kids, and all other members of my family and friends who gave me so much help for the years in Canada.

# DEDICATION

To my parents **Orsan Kanan** and **Nawal Ismail**
for their love and support.

To my lovely wife **Soha Khatatbeh**
for her support and sacrifice.

To my beautiful children
**Zainah**, **Mohammad**, and **Molham**.

# List of Abbreviations

| | |
|---|---|
| **AI** | Artificial Intelligence |
| **DG** | Dependence Graph |
| **DNA** | Deoxyribonucleic Acid |
| **FPGA** | Field Programmable Gate Array |
| **GPU** | Graphical Processing Unit |
| **HDLSS** | High Dimensional Low Sample Size |
| **HPC** | High Performance Cluster |
| **I/O** | Input/Output |
| **KNN** | K-Nearest Neighbours |
| **LUT** | Lookup Table |
| **MIMD** | Multiple Instruction Multiple Data |
| **MISD** | Multiple Instruction Single Data |
| **MP** | Message Passing |
| **n-D** | n-Dimensional (n=1, 2, 3, ...) |
| **NUMA** | Non-Uniform Memory Access |
| **PE** | Processing Element |

**RAM** Random Access Memory

**RGB** Red Green Blue

**SIMD** Single Instruction Multiple Data

**SISD** Single Instruction Single Data

**SP** Stream Processor

**SVM** Support Vector Machine

**UMA** Uniform Memory Access

**VLSI** Very Large Scale Integration

# Chapter 1

# Introduction

## 1.1 Background

With the advances achieved in the field of computing, huge volumes of data are collected by governments, businesses, universities, and other organizations. As the data collection rate increases, the gap between our understanding of such data and the knowledge hidden in it becomes too large. To bridge this gap, data mining techniques and algorithms emerged to discover meaningful patterns and knowledge from such large volumes of data. Data mining plays an important role in a variety of fields including bioinformatics, multimedia, business intelligence, marketing, and medical diagnosis [5].

Analysis of today's huge and complex data involves several data mining tasks including clustering [6] and classification [7]. A clustering algorithm partitions data into subgroups called clusters such that data elements within a cluster share some degree of similarity while being dissimilar to data in other clusters. A classifier, on the other hand, is used to assign data elements to a set of predefined groups or classes based on prior assumptions and directions from a human user [8].

In order to design efficient hardware accelerators for data mining applications, it is important to analyze the computations involved in these applications at the algorithm level. The performance of these applications can be significantly improved by adopting certain algorithm-level approaches, that have been adopted in software implementations, to be implemented in hardware.

Systematic approaches to design parallel hardware architectures allow for design space exploration to optimize the performance according to certain specifications while satisfying design constraints. In this work, a systematic approach for exploring the design space

of processor array architectures for data mining applications is presented. The approaches and methodologies presented in this dissertation are used to optimize hardware acceleration for the computations involved in the K-Means clustering algorithm. However, these approaches and methodologies can be adapted to optimize the targeted computational kernels that are involved in a variety of other algorithms.

### 1.1.1 Machine Learning and Data Mining

Machine learning is a research area in artificial intelligence (AI) that is concerned with developing methods and algorithms that can learn from empirical data. These algorithms can be generally divided into two main categories; supervised learning [9] and unsupervised learning [10]. In supervised learning, a set of known and labeled data is used as training examples so that the algorithm will be able to predict the classes or labels of unknown data. On the other hand, unsupervised learning aims to extract hidden structures in a given unlabeled dataset. Classification and clustering are examples of supervised and unsupervised learning, respectively. Other categories of machine learning algorithms, including reinforcement learning and semi-supervised learning, also exist [11].

Data mining aims to discover patterns and useful knowledge from a large set of data. This is achieved by utilizing methods from machine learning and other fields such as statistics, database systems, and optimization. Data has to be represented in a format that is accessible by machine learning algorithms. A Dataset is generally represented as a set of samples with quantitative features [11]. The size of a dataset is the number of samples in it, and its dimensionality refers to the number of features per sample. The works presented in this dissertation incorporate datasets of different dimensionalities ranging from one-dimensional datasets of only a single feature in each sample to high-dimensional datasets of thousands of features per sample.

### 1.1.2 Parallel Computing

Research in the fields of machine learning and data mining requires skills to gather, store, and analyze huge amounts of data. Most of these tasks require the capabilities that are beyond those of a desktop machine. Hence, high-performance computers are expected to play an increased role in this field of research. A parallel computer uses a group of processors that cooperate to solve computational problems. The problem in hand has to be decomposed into parts with each part being assigned to a processor. Partial computations then have to be gathered to get an accurate outcome [12].

Parallel algorithms are closely related to parallel hardware architectures. We have to consider parallel software implementations that drive the parallel hardware, and the hardware that supports parallel software implementations. Parallelism can be implemented at different levels in a computing system using hardware and software techniques [1]:

- **Data-level parallelism:** Simultaneous operations on multiple data. Examples of this are bit-parallel arithmetic operations, vector processors, and systolic arrays.
- **Instruction-level parallelism:** Processor simultaneously executes more than one instruction. An example of this is the use of instruction pipelining.
- **Thread-level parallelism:** A thread is a part of a program that shares some resources with other threads. At this level of parallelism, multiple software threads are executed on one processor, or more processors simultaneously.
- **Process-level parallelism:** A process is a running program on the computer. It reserves its own resources, like registers and memory space. This level of parallelism is the classic multitasking computing, in which multiple programs are running on one or more computers simultaneously.

The best-known classification scheme for parallel computers was proposed by Flynn [13]. In this classification, a machine's class depends on the parallelism it exhibits in its instruction and data streams. Flynn's taxonomy has four categories:

1. **Single instruction single data stream (SISD):** This category refers to computers with a single instruction stream and a single data stream. A uniprocessor computer is an example of this category.
2. **Single instruction multiple data stream (SIMD):** It refers to computers with a single instruction stream on different data streams. All the processors execute the same instruction on different data.
3. **Multiple instruction single data stream (MISD):** This category refers to a pipeline of multiple functional units operating on a single stream of data, and forwarding results from one unit to the next [14].
4. **Multiple instruction multiple data stream (MIMD):** Different processors can simultaneously run their own instructions on local data. In general, multicore processors and multithreaded multiprocessors fit into this category.

An overview of common parallel computer architectures are presented in more details in the following subsections [1].

#### 1.1.2.1 Shared-Memory Multiprocessors - Uniform Memory Access (UMA)

All processors in this architecture can access the same address space of the shared memory through an interconnection network as shown in Figure 1.1. The interconnection network may be a bus in simple systems. For larger systems, memory bandwidth becomes the bottleneck, and the shared bus has to be replaced with an interconnection network so that several processors can simultaneously access the shared memory.

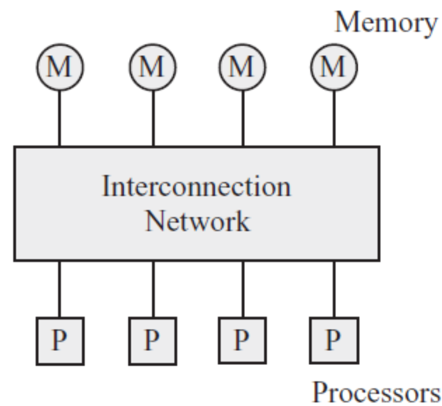



Figure 1.1: Shared-Memory Multiprocessor Architecture [1].

#### 1.1.2.2 Distributed-Memory Multiprocessors - Non-Uniform Memory Access (NUMA)

In this architecture, each processor has its own local memory that can be accessed directly, as shown in Figure 1.2 . A message passing (MP) mechanism is used in order to allow a processor to access other memory modules associated with other processors. Memory access is not uniform since it depends on whether the memory module to be accessed is local to the processor, or has to be accomplished through the interconnection network.

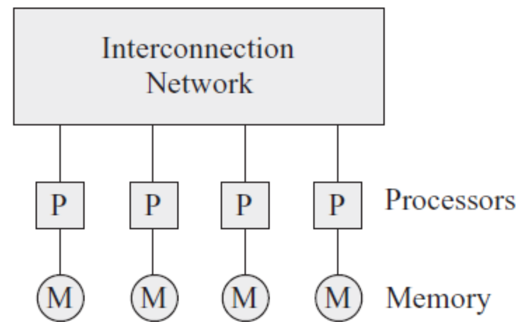



Figure 1.2: Distributed-Memory Multiprocessor Architecture [1].

#### 1.1.2.3 Systolic Processors

A systolic processor consists of an array of processing elements (PEs). It could be 1-D, 2-D, or even higher. Usually, all PEs perform the same task. Interconnections among the PEs are generally neighbor to neighbor with some global interconnections. Each PE is provided with a small memory to store data. Systolic processors are designed to implement a specific regular algorithm with simple data dependencies.

#### 1.1.2.4 Stream Multiprocessors

A stream multiprocessor is a SIMD or a MIMD system that consists of processors called streaming processors (SPs) or thread processors. As the name indicates, these processors deal with data streams [15]. This concept is closely related to the graphics processing unit (GPU) where the GPU is used to perform general-purpose intensive computations. New generations of GPUs from NVIDIA [16] is an example of successful stream multiprocessor.

## 1.2 Motivation

Recently, machine learning and data mining have been effectively utilized to solve several real world compute-intensive problems. Existing serial implementations of data mining algorithms are not sufficient to analyze and process this enormous amount of data in an effective and efficient manner. To satisfy performance constraints and requirements associated with data mining applications, some sort of acceleration is necessary. Different acceleration platforms are available for big data analysis such as High Performance Clusters (HPC), Multicore processors, Graphics Processing Units (GPU), and Field Programmable

Gate Arrays (FPGA). Among these acceleration platforms, FPGA-based hardware accelerators are more suitable for applications that require high I/O data rates and real-time processing [17]. The computational complexity of machine learning and data mining algorithms that are frequently used in embedded applications such as handwritten analysis, fingerprint/iris/signature verification, and face recognition, makes the design of efficient hardware accelerators for these algorithms a challenging issue.

Several approaches have been proposed to improve the performance of software implementations of the K-means clustering algorithm. Such approaches can be adopted in a hardware implementations of the algorithm to improve its performance and efficiency. Most of the proposed hardware implementations have not adopted these approaches due to the complex hardware structures required to implement them especially, for high-dimensional data. However, these algorithm-level approaches can be used for clustering one-dimensional data to obtain faster and more area-efficient hardware architectures.

Processor array architectures have been employed, as an accelerator, to compute similarity distance found in a variety of data mining algorithms. However, most of the proposed architectures in existing literature are designed in an ad hoc manner. Furthermore, data dependencies have not been analyzed and often only one design choice is considered for the scheduling and mapping of computational tasks. We believe a systematic approach is necessary to design processor array architectures to perform design space exploration and optimize the performance according to certain specifications while satisfying design constrains.

## 1.3 Research Objectives

In this work, we aim to optimize the performance of hardware accelerators for data mining applications in terms of speed and area. Our research objectives are:

1. Investigate algorithm-level approaches to design efficient hardware architectures for clustering one-dimensional data.

2. Explore parallelism in similarity distance computation, and systematically design parallel processing architectures to accelerate it.

3. Design scalable hardware architectures for processing large-scale data using nonlinear scheduling and projection operations.

## 1.4 Contributions

### 1.4.1 Efficient Hardware Implementation of K-Means Clustering for 1-D Data

We investigate the approach of continuous cluster centroids update [18], that results in faster convergence of software implementations of the K-Means algorithm, to enhance the speed of hardware accelerators of the algorithm for clustering one-dimensional data, and applications that rely on analyzing data elements in individual dimensions first, and then use the results of this analysis to perform full dimension-wide processing [19], [20]. We also propose to approximate the area-consuming division operation involved in updating cluster centroids to design an area-efficient hardware implementation of the algorithm while maintaining the quality of clustering results.

### 1.4.2 2-D Processor Array Architectures for Similarity Distance Computation

We present a systematic methodology that allows for design space exploration of 2-D processor array architectures for the computation of similarity distance matrices that is required by several machine learning and data mining algorithms to measure the degree of similarity between data samples. In traditional approach, data dependencies are analyzed in dependence graphs, by showing how output variables depend on input variables. In this work, however, data dependencies are analyzed using dependence matrices that show how input and output variables depend on the algorithm indices. Linear scheduling functions are obtained to determine the computational load to be performed by the system at each time step. Linear projection is then used to map several points of the computation domain to a single processing element in the projected 2-D processor arrays.

### 1.4.3 Linear Processor Array Architectures for Similarity Distance Computation

Compared to 2-D processor arrays, linear (1-D) arrays generally require less hardware resources and I/O bandwidth at the cost of slower computation. The same methodology employed to explore the design space of 2-D arrays is extended to explore the design space of 1-D processor array architectures for the similarity distance problem. To meet area and

bandwidth constrains, more time restrictions are introduced on input and output variables. The projection operation is also modified to map points in the computation domain to processing elements in the projected 1-D processor arrays.

### 1.4.4 Scalable and Parameterizable Processor Array Architecture for Similarity Distance Computation.

Given the complexity of today's data, machine learning and data mining algorithms are expected to be able to handle huge and high-dimensional datasets. The size and dimensionality of input datasets have not been taken into consideration in most previous 2-D processor array architectures. For high-dimensional data, feeding all the features of a single data element simultaneously is not practical due to limited I/O pins and bandwidth constraints. To overcome this problem, and to obtain practical designs that are amenable for hardware implementation, 1-D arrays have been proposed in the literature. Although 1-D processor arrays are more area-efficient than 2-D arrays, they are much slower. In this work, nonlinear scheduling and projection are introduced to allow for more control on the workload executed at each time step and the number of processing elements in the projected processor array. The employed scheduling and projection operations result in a fully parameterized architecture that scales better for large and high-dimensional data with improved compromise between area and speed.

## 1.5 Dissertation Organization

The rest of this dissertation is organized as follows.

In Chapter 2, a fast and area-efficient hardware implementation of the K-Means algorithm is proposed for clustering one-dimensional data. Continuous update of cluster centroids is adopted in the proposed architecture to reduce the number of iterations required by the algorithm to converge. Centroids update equations are also modified to approximate the slow and area-consuming division operation while maintaining the quality of clustering results.

Chapter 3 presents a systematic methodology for exploring the design space of similarity distance computation in machine learning algorithms. The methodology presented in this work is used to obtain the 3-D computation domain of the similarity distance computation algorithm. Four linear scheduling functions are presented, and six possible 2-D processor array architectures are obtained and classified based on the size and dimension-

ality of the input datasets. The obtained designs are analyzed in terms of speed and area, and compared with previously obtained designs.

In Chapter 4, the methodology used in Chapter 3 is extended to explore the design space of linear (1-D) processor array architectures for the similarity distance computation problem. Linear scheduling and projection operations are also used to obtain and analyze six linear processor array architectures.

Nonlinear scheduling and projection operations are introduced in Chapter 5 to design scalable processor array architecture for the computation of similarity distance matrices. The techniques presented in this chapter improve the scalability of the proposed architecture to accommodate large and high-dimensional data with improved compromise between area and speed.

Chapter 6 summarizes the dissertation and outlines some potential future work.

# Chapter 2

# Efficient Hardware Implementation of K-Means Clustering for 1-D Data

## 2.1 Introduction

K-Means algorithm [21] is a clustering algorithm [22] that is commonly used for data analysis in many fields like machine learning, pattern recognition, bioinformatics, and image processing. The algorithm aims to partition data elements of an input dataset into $K$ subgroups called clusters such that data elements within a cluster share some degree of similarity while being dissimilar to data in other clusters.

Because of its importance and high computational requirements, various FPGA-based hardware accelerators of the K-Means algorithm have already been proposed in the literature [23–28]. In [24], the authors proposed a software/hardware co-design of the K-Means algorithm. In the proposed design, distance calculation was implemented in hardware while new centroids were calculated by the host computer. The proposed hardware implementation benefited from truncating the bit width of the input data, and achieved a speedup of 50x over the implementation on a 500 MHz Pentium III host processor. The author of [25] proposed a systolic array architecture of the K-Means clustering algorithm. The aim was to accelerate the distance calculation unit by calculating the distance between the input data and the centroids of the $K$ clusters in parallel. The cluster index is obtained at the end of the array, and results are sent back to the host computer to calculate the new centroids. In [26], the authors proposed an efficient FPGA implementation of the K-Means algorithm by utilizing a floating point divider [27] for the calculation of new centroids within the FPGA. The proposed approach required extra blocks to convert between fixed-point and

floating-point data representations. The speedup of the hardware implementation using this floating point divider was compared with a hardware implementation that has the division performed in the host computer. No speedup or advantages, other than freeing the host for other tasks while the new centroids are being calculated in hardware, was gained. The authors of [28] fully implemented the K-Means algorithm in hardware to cluster Microarray genomic data. The objective of their work was to have multiple cores of the K-Means algorithm operating on the same chip to cluster multiple datasets in a server solution.

Several approaches have been proposed to improve the performance of software implementations of the K-means clustering algorithm. Most of the proposed hardware implementations of the algorithm have not adopted such approaches due to the complex hardware structures required to implement them especially, for high-dimensional data. However, these algorithm-level approaches can be used for clustering one-dimensional data to obtain faster and more area-efficient hardware architectures. In this chapter, we investigate the approach of continuous cluster centroids update [18], that results in faster convergence of software implementations of the K-Means algorithm, to enhance the speed of hardware accelerators of the algorithm for clustering one-dimensional data, and applications that rely on analyzing data elements in individual dimensions first, and then use the results of this analysis to perform full dimension-wide processing [19], [20]. We also propose to approximate the area-consuming division operation involved in updating cluster centroids to design an area-efficient hardware implementation of the algorithm while maintaining the quality of clustering results.

## 2.2 K-Means Clustering Algorithm

The K-Means clustering algorithm aims to partition an input dataset of $m$ data elements into $K$ subgroups called clusters such that data elements within a cluster share some degree of similarity while being dissimilar to data in other clusters.

The pseudo code of the standard K-Means clustering algorithm is shown in Algorithm 2.1. The algorithm proceeds in iterations, each iteration begins with $K$ centroids corresponding to the $K$ clusters. For the first iteration, clusters centroids are initialized with random numbers in the range of values in the input dataset as in line 1 in Algorithm 2.1. Each data element is then assigned to one of the $K$ clusters whose centroid is at minimal distance to the data element based on a distance metric [24] such as the Euclidean distance:

**Algorithm 2.1** Pseudo code of the standard K-Means algorithm.

**Inputs:**
***e***=[ $e_1$ $e_2$ . . . $e_m$ ]: Input dataset of m data elements
***K***: Number of Clusters
**Outputs:**
***c***=[$c_1$ $c_2$ . . . $c_K$]: Cluster centroids
***l***=[$l_1$ $l_2$ . . . $l_m$]: Cluster labels of ***e***
***n***=[$n_1$ $n_2$ . . . $n_K$]: Number of data elements in each cluster
***s***=[$s_1$ $s_2$ . . . $s_K$]: Sum of all data elements in each cluster

```
1:  Initialize_Centroids(c);
2:  done = false;
3:  repeat
4:      s=[0 0 ... 0];
5:      n=[0 0 ... 0];
6:      changed = false;
7:      for i=1 to m do
8:          min_dist = ∞;
9:          for j=1 to K do
10:             if D_ij<min_dist then
11:                 min_dist= D_ij;
12:                 label = j;
13:                 changed = true;
14:             end if
15:         end for
16:         l_i = label;
17:         s_label+=e_i;
18:         n_label++;
19:     end for
20:     for k=1 to K do
21:         c_k=s_k/n_k;
22:     end for
23: until changed == false
24: done = true;
```

$$D_{ij} = \sqrt{\sum_{k=1}^{f} (e_{ik} - c_{jk})^2} \tag{2.1}$$

or Manhattan distance:

$$D_{ij} = \sum_{k=1}^{f} |e_{ik} - c_{jk}| \tag{2.2}$$

where $D_{ij}$ is the distance between data element $i$ and the centroid of cluster $j$, $e_{ik}$ is the value of feature $k$ of data element $i$, $c_{jk}$ is the value of feature $k$ of the centroid of the cluster $j$, and $f$ is the number of features or dimensions. In this work, Manhattan distance is used for clustering one-dimensional data. For $f$=1 in Equation (2.2), the distance between a data element and a cluster centroid is simply the absolute value of the difference between their values.

Cluster centroids are then updated according to the new partitioning of the dataset. Centroids update may take place either continuously after the assignment of each data element, or once at the end of each iteration after the assignment of all data elements. Continuous updating of centroids is adopted in this work since the more often the centroids are updated, the faster the algorithm will converge [18]. The algorithm repeats for a specific number of iterations or until cluster centroids remain unchanged as a result of data elements stop moving across clusters [25].

The K-Means algorithm can be formally represented as [29]:

$$X_j(t) = \{i : D_{ij} \leq D_{im} \forall\, m = 1, ..., K\} \tag{2.3}$$

where $X_j(t)$ is the set of data elements assigned to cluster $j$ at iteration $t$, $D_{ij}$ denotes a suitable distance metric, such as the Euclidean or Manhattan distance, between data element $i$ and $c_j(t-1)$, the centroid of cluster $j$ calculated in the previous iteration. Initial centroids $c_j(0)$ for the first iteration are assigned random values. It is also required that the sets $X_j$ are disjoint, i.e.,

$$X_j(t) \cap X_m(t) = \phi, \forall\, m \neq j. \tag{2.4}$$

New clusters centroids are calculated at the end of each iteration as a simple average

by dividing the summation of all data elements assigned to each cluster by the number of these elements:

$$s_j(t) = \sum_{i \in X_j(t)} e_i \tag{2.5}$$

$$c_j(t) = \frac{s_j(t)}{n_j(t)} \tag{2.6}$$

where $c_j(t)$ is the updated centroid of cluster $j$ at the end of iteration $t$, $e_i$ is the value of data element $i$, and $n_j$ is the number of elements in the set $X_j(t)$.

## 2.3 Conventional Implementation of the K-Means Algorithm

The conventional implementation of the standard K-Means clustering algorithm, described in Algorithm 2.1, differs from the proposed implementation, that is introduced in Section 2.4, in two points; the calculation of the new centroids, and the update frequency of clusters centroids. In the conventional implementation, as shown in lines 4 and 5 in Algorithm 2.1, the summation of all data elements assigned to each cluster along with the number of these data elements are stored in special registers. A general update of the cluster centroids are performed at the end of each iteration. The new centroid of each cluster is calculated by dividing the summation of all data elements assigned to this cluster by the number of these elements, according to line 21 in Algorithm 2.1 and Eq. (2.6).

Figure 2.1 shows the functional blocks of the conventional hardware design of the K-Means algorithm as implemented in [26], [28], [2]. The Distance Calculation unit and the Minimum Distance unit are the same in both the conventional and proposed designs. The first unit is responsible for calculating the distances between the data element and all cluster centroids. These distances are passed to the second unit to find the minimum distance among them. The index of the cluster with nearest centroid to the data element is passed to the Accumulation unit. The value of the data element is added to the sum of all data elements assigned to the cluster with the index passed from the Minimum Distance unit. The register that holds the number of elements assigned to this cluster is also incremented by one. This process repeats to assign all data elements to their nearest clusters. The Centroids Update unit then calculates the new centroid of each cluster by dividing the summation of

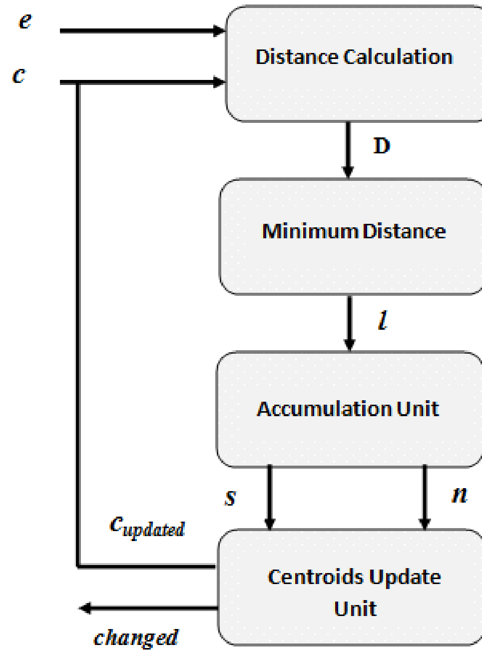



Figure 2.1: Functional blocks of the conventional K-Means algorithm hardware architecture.

all data elements assigned to this cluster by the number of these data elements.

## 2.4 The Proposed Approach

In this section, we introduce our approach to enhance the conventional hardware design of the K-Means algorithm in terms of speed and area efficiency. Algorithm 2.2 shows the pseudo code of the proposed implementation of the K-Means algorithm. Initialization step in line 1 is performed by randomly assigning all data elements to the $K$ clusters and initialize $c$, $l$, and $n$ based on this random assignment. To achieve faster convergence of the algorithm, clusters centroids are updated continuously every time a data element is assigned to a cluster, according to lines 14-23 in Algorithm 2.2. The centroids of two clusters need to be updated; the source cluster, from which the data element is removed, and the destination cluster, to which the data element is assigned. Implementation details of the proposed hardware design, along with the differences between the conventional and proposed implementations are described in Section 2.5.

New centroid update equations, that are derived in the following subsections, are written

in a recursive form. New centroids are calculated as the sum of the current centroid value and the change in this value that results from adding or removing one data element to or from the cluster. In the new centroid update equations, division operation appears only in the term that represents this change. The proposed implementation approximates only the value of this change by replacing the slow and area-consuming division operation with a shift operation. The main advantage of rewriting these equations in this form, as will be shown in Section 2.6, is to minimize the effect of this approximation on the quality of clustering results. Another advantage of using this recursive form is that new centroids are calculated without the need to accumulate the summation of all data elements in each cluster, as in the conventional accumulation-based implementation of the algorithm. The following subsections introduce the proposed approach to update the centroids of the source and destination clusters.

### 2.4.1 Source Cluster

As a result of removing a data element from its source cluster, the number of elements assigned to the source cluster is decremented by one, and the centroid is updated after subtracting the value of data element from the sum of data elements in the source cluster. We can write the following iterative equations:

$$n_{src}(t) = n_{src}(t-1) - 1 \tag{2.7}$$

$$c_{src}(t) = \frac{[n_{src}(t) + 1]c_{src}(t-1) - e_i}{n_{src}(t)} \tag{2.8}$$

After simple algebra, we obtain the simplified equation:

$$c_{src}(t) = c_{src}(t-1) + \frac{c_{src}(t-1) - e_i}{n_{src}(t)} \tag{2.9}$$

where:
$e_i$: Value of data element $i$.
$c_{src}(t-1)$: Current centroid of the source cluster.
$c_{src}(t)$: Updated centroid of the source cluster.
$n_{src}(t-1)$: Current number of elements in the source cluster.
$n_{src}(t)$: Updated number of elements in the source cluster.

**Algorithm 2.2** Pseudo code of the proposed K-Means algorithm implementation.

**Inputs:**

***e***=[ $e_1$ $e_2$ . . . $e_m$ ]: Input dataset of m data elements

***K***: Number of Clusters

**Outputs:**

***c***=[$c_1$ $c_2$ . . . $c_K$]: Cluster centroids

***l***=[$l_1$ $l_2$ . . . $l_m$]: Cluster labels of ***e***

***n***=[$n_1$ $n_2$ . . . $n_K$]: Number of data elements in each cluster

```
 1: Initialize(c,l,n);
 2: done = false;
 3: repeat
 4:     changed = false;
 5:     for i=1 to m do
 6:         src = l_i;
 7:         min_dist = ∞;
 8:         for j=1 to K do
 9:             if D_ij < min_dist then
10:                 min_dist=D_ij;
11:                 dest = j;
12:             end if
13:         end for
14:         if src ≠ dest then
15:             l_i = dest;
16:             changed = true;
17:             n_src--;
18:             n_dest++;
19:             X=Nearest_Power_of_2(n_src);
20:             Y=Nearest_Power_of_2(n_dest);
21:             c_src+=(c_src-e_i)>>X;
22:             c_dest+=(e_i-c_dest)>>Y;
23:         end if
24:     end for
25: until changed == false
26: done = true;
```

### 2.4.2 Destination Cluster

After a data element is assigned to its destination cluster, the number of elements in the destination cluster is incremented by one, and the centroid is updated after adding the value of the data element to the sum of data elements in the destination cluster. We can write the following iterative equations:

$$n_{dest}(t) = n_{dest}(t-1) + 1 \tag{2.10}$$

$$c_{dest}(t) = \frac{[n_{dest}(t) - 1]c_{dest}(t-1) + e_i}{n_{dest}(t)} \tag{2.11}$$

After simple algebra, we obtain the simplified equation:

$$c_{dest}(t) = c_{dest}(t-1) + \frac{e_i - c_{dest}(t-1)}{n_{dest}(t)} \tag{2.12}$$

where:
$e_i$: Value of data element $i$.
$c_{dest}(t-1)$: Current centroid of the destination cluster.
$c_{dest}(t)$: Updated centroid of the destination cluster.
$n_{dest}(t-1)$: Current number of elements in the destination cluster.
$n_{dest}(t)$: Updated number of elements in the destination cluster.

The values of $n_{src}$ and $n_{dest}$ are rounded to their nearest power of 2 integer, and the division is performed as a shift right by the rounded values according to lines 19-22 in Algorithm 2.2.

## 2.5 Hardware Design and Implementation

The proposed hardware architecture of the K-Means clustering algorithm, shown in Figure 2.2, consists of four fully pipelined units. Both Distance Calculation and Minimum Distance units have the same hardware implementation as in the conventional design of the algorithm. The main differences between the conventional and proposed designs are in the Accumulation/Count, and Centroids Update units. Hardware design and implementation details of these units are presented in the following subsections.

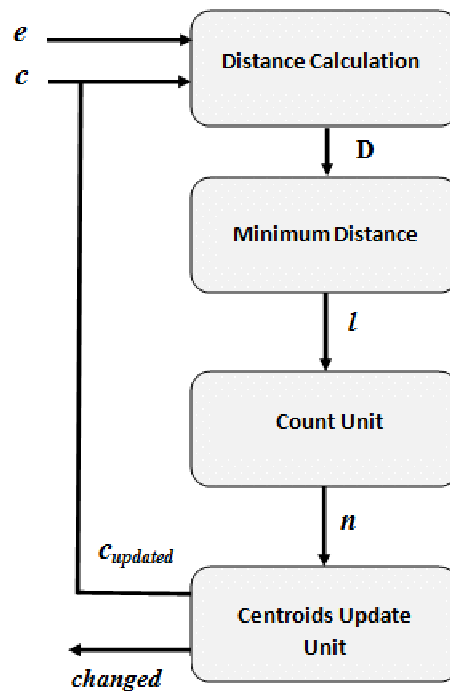



Figure 2.2: Functional blocks of the proposed K-Means algorithm hardware architecture.

### 2.5.1 Distance Calculation Unit

The distance calculation unit, shown in Figure 2.3, reads one data element every clock cycle, and calculates the $K$ distances between a data element and the centroids of the $K$ clusters simultaneously.

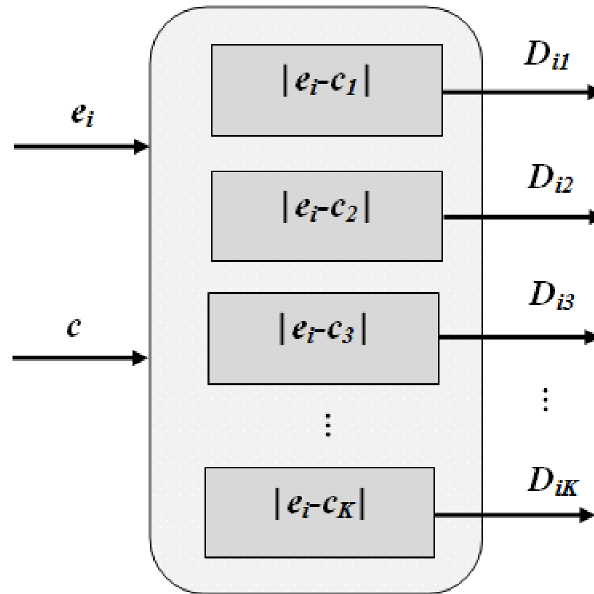



Figure 2.3: Distance Calculation unit.

### 2.5.2 Minimum Distance unit

The $K$ distances from the previous stage go through a compare tree to find the index of the cluster with the minimum distance to the data element as shown in Figure 2.4. This unit is pipelined, and the number of stages is equal to the number of levels in the compare tree, which is equal to $\lceil \log_2(K) \rceil$, where $K$ is the number of clusters. The output of this unit represents the label of the nearest cluster to the current data element. As shown in Figure 2.4, the data element $e$ is passed through the pipeline stages since it is needed in next stages to update clusters centroids.

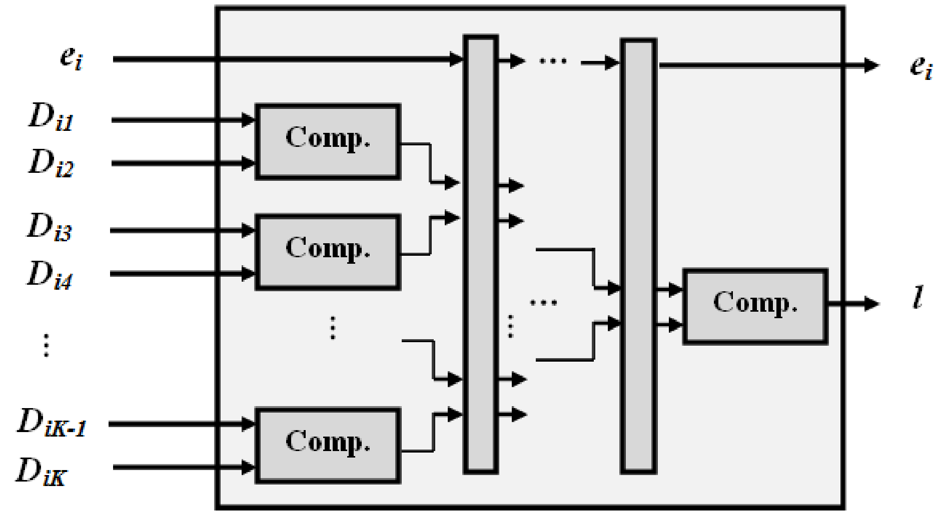



Figure 2.4: Minimum Distance unit.

### 2.5.3 Count Unit

This unit keeps track of the number of data elements in each cluster. The Accumulation unit of the conventional design requires 2$K$ registers. $K$ registers to accumulate the sum of all data elements assigned to each cluster, and another $K$ registers to store the numbers of data elements in each cluster. The proposed design, on the other hand, calculates the new centroids recursively without the need to accumulate the sums of all data elements in each cluster. Hence, only $K$ registers are required by the Count unit of the proposed design. Figure 2.5 shows the hardware design of the Count unit. Inputs *src* and *dest* represent the labels of the source and destination clusters of the data element $e$, respectively. A comparator is used to check for the equality of *src* and *dest* to make sure that the data element is assigned to a new cluster other than its current cluster. If *src* and *dest* are not equal, the counter associated with the source cluster is decremented by one and the counter associated with the destination cluster is incremented by one, according to equations (2.7) and (2.10) and lines 17 and 18 in Algorithm 2.2. The values of the two registers $n_{src}$ and $n_{dest}$ are passed to the Centroids Update unit.

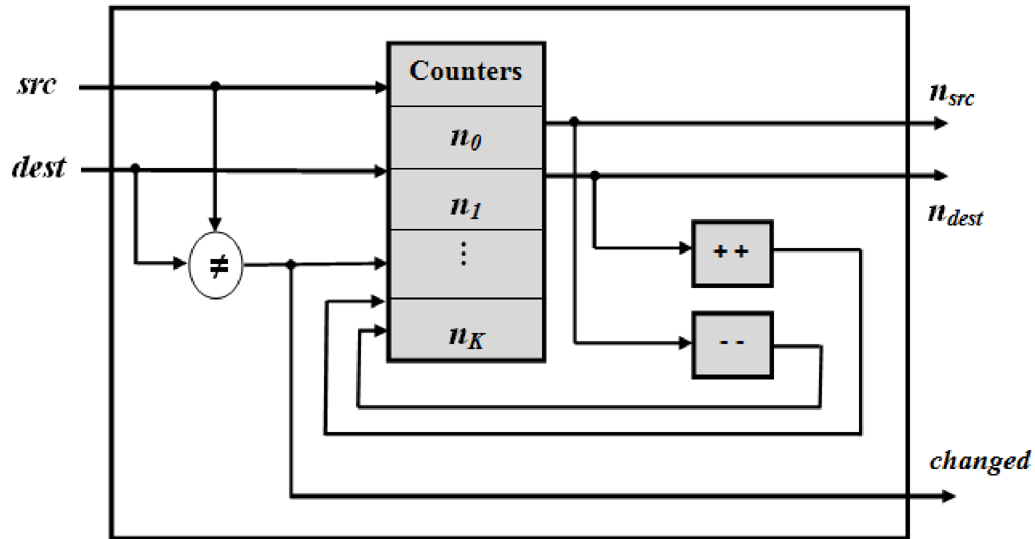



Figure 2.5: Count unit.

### 2.5.4 Centroids Update Unit

As its name indicates, this unit is responsible for updating the centroids of the source and destination clusters of data elements. In the conventional design, centroids update takes place only after the assignment of all data elements in the input dataset. New centroids are calculated by dividing the sum of all data elements assigned to a cluster by the number of these data elements, which are calculated and stored in the Accumulation unit. In the proposed design, centroids of the source and destination clusters are updated continuously after the assignment of each data element according to equations (2.9) and (2.12) and lines 14-23 in Algorithm 2.2.

As shown in Figure 2.6, the Centroids Update unit is pipelined with three pipeline stages. In the first stage, the two inputs $n_{src}$ and $n_{dest}$ are rounded either up or down to their nearest power of 2 integer. The value of the inputs are checked against a set of intervals, and the rounded outputs are determined based on the existence of the input in one of these intervals. Shift registers in the second stage are implemented in Verilog using the shift right operator with variable shift amount equals to the rounded value of $n_{src}$ or $n_{dest}$. This implementation infers a barrel shifter, which is a combinational circuit that performs the shift operation $(x >> sh_amount)$ in one clock cycle, where $sh_amount$ is the shift amount. Two adders are used to calculate the updated values of the source and destination clusters according to lines 21 and 22 in Algorithm 2.2. The algorithm proceeds in iterations,

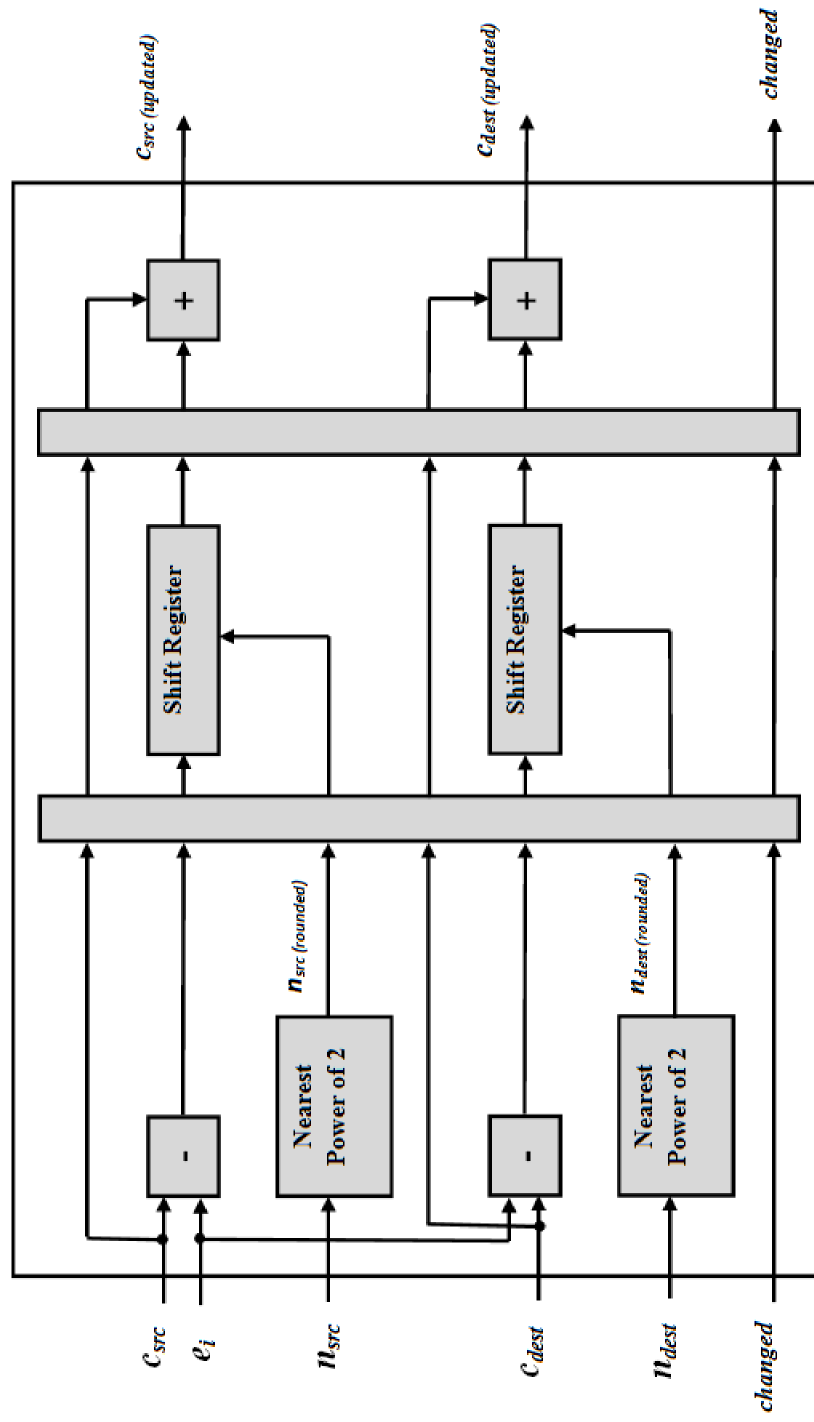



Figure 2.6: Centroids Update unit.

and the value of the output $changed$ is used to check for algorithm convergence at the end of each iteration. A value of zero for this output means that data elements stop moving between clusters, and hence the algorithm converges.

The total number of stages in the pipelined datapath of the proposed design is:

$$N_{Stages} = 1 + \lceil \log_2(K) \rceil + 1 + 3 \tag{2.13}$$

$$N_{Stages} = \lceil \log_2(K) \rceil + 5 \tag{2.14}$$

where *K* is the number of clusters.

## 2.6 Results and Discussion

The proposed design was implemented in Verilog hardware description language using Xilinx ISE Design Suite 9.2 to target Xilinx Virtex4 XC4VLX25. Table 2.1 shows implementation results for clustering a one-dimensional image dataset, that was used in previous FPGA [2] and GPU [3] implementations of the algorithm, into 8 Clusters. The implementation occupies 8% of the available slices with a maximum clock frequency of 121 MHz.

Table 2.1: Implementation results for clustering one-dimensional dataset into 8 clusters on Xilinx XC4VLX25.

| **Logic Utilization** | **Available** | **Used** | **Utilization** |
|---|---|---|---|
| No. of Slices | 10752 | 918 | 8% |
| No. of Slice Flip Flops | 21504 | 629 | 3% |
| No. of 4-input LUTs | 21504 | 1598 | 7% |
| Max. Clock Frequency | 121 MHz | | |

Figure 2.7 shows the effect of adopting the continuous update of cluster centroids on the speed of convergence of the proposed design compared to the general update approach used in the conventional design. The speedup is calculated as the ratio of the number of iterations required by the conventional design to converge, to the number of iterations required by the proposed design. Behavioral simulation results for ten runs of the two implementations, described in Algorithm 2.1 and Algorithm 2.2, are shown. Each run starts with a different set of random initial clusters centroids. Both implementations were fed with the same initial centroids in each run. On each box, the central line indicates the median,

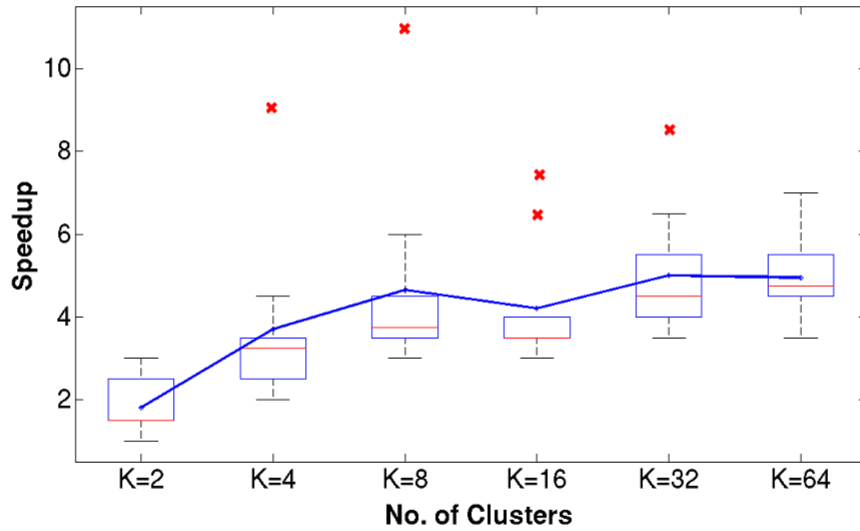



Figure 2.7: Speedup of the proposed design over the conventional design calculated as the ratio of number of iterations required to converge.

and the bottom and top edges of the box indicate the 25th and 75th percentiles, respectively. The outliers are plotted individually using the ✖ symbol, and the continuous line on the box plot connects the average speedups of the 10 runs for different number of clusters. Simulation results show that the proposed design converges faster than the conventional design using less number of iterations. It is clear from the variation of speedup values that the convergence of the algorithm is highly dependent on the initial cluster centroids. To avoid biased results when comparing the two implementations, we use the same initial centroids in our experiments.

In [2], a conventional FPGA implementation of the K-Means algorithm is compared with a GPU implementation of the algorithm presented in [3] for an image processing application. The results were based on a 0.4 Mega Pixel image dataset clustered into 16, 32, and 64 clusters. GPU results in [3] were based on 2.2 GHz Intel Core 2 Duo, with 4GB RAM and Nvidia GeForces 9600MGT graphics card. In [2], the targeted FPGA device was Xilinx XC4VSX35 running at a maximum clock frequency of 141 MHz. A comparison between the proposed FPGA implementation of the K-Means algorithm and the previous FPGA and GPU implementations in terms of speed is shown in Table 2.2. The table shows the time per single iteration, and the complete execution time for the three implementations.

For the implementation of the proposed design, time measurements are based on a clock frequency of 121 MHz. The execution time for a single iteration is:

$$T_{Single} = \frac{C_{Single}}{F} \tag{2.15}$$

where $C_{Single}$ is the number of clock cycles required to complete one iteration of the algorithm, and $F$ is the clock frequency.

The complete execution time of the algorithm is the time required to complete $N_{iter}$ iterations of the algorithm before it converges:

$$T_{Complete} = T_{Single} \times N_{iter} \tag{2.16}$$

As shown in Table 2.2, both FPGA implementations were faster than the GPU implementation for all values of $K$. The GPU implementation is used as a reference implementation to compare the speedup of the two FPGA implementations over it. The proposed implementation took more time to complete a single iteration compared to the conventional implementation in [2]. One reason for this is because of the continuous update approach used in the proposed design, that requires more number of updates compared to the general update approach used in the conventional design. The second reason is that the conventional implementation in [2] achieved a higher maximum clock frequency compared to the maximum frequency achieved in the proposed implementation. However, and as shown if Figure 2.7, the continuous update adopted in the proposed design results in the algorithm to converge after less number of iterations, and hence reducing the complete execution time of the algorithm. The speedups of both FPGA implementations over the GPU implementation is shown in Figure 2.8.

Table 2.2: Execution times of GPU, conventional FPGA, and proposed FPGA implementations of the K-Means algorithm.

| **K** | **GPU [3]** | | **FPGA [2]** | | **Proposed** | |
|---|---|---|---|---|---|---|
| | Single Iteration Time (ms) | Avg. Complete Execution Time (ms) | Single Iteration Time (ms) | Avg. Complete Execution Time (ms) | Single Iteration Time (ms) | Avg. Complete Execution Time (ms) |
| **16** | 21 | 443 | 2.8 | 39.2 | 3.3 | 9.57 |
| **32** | 20 | 421 | 2.8 | 42 | 3.3 | 10.395 |
| **64** | 23 | 508 | 2.8 | 45.4 | 3.3 | 12.37 |

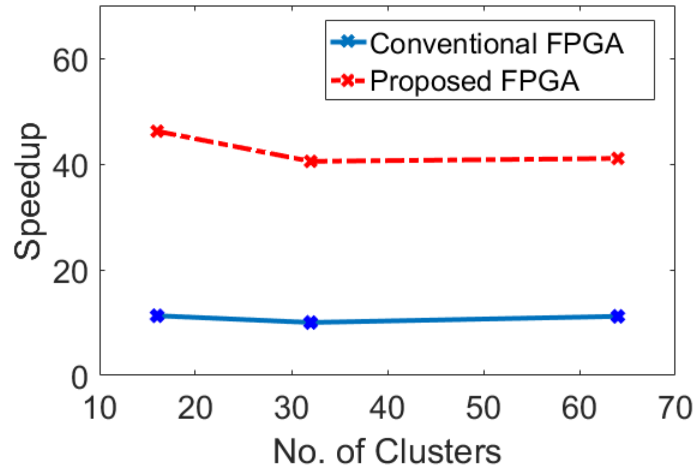



Figure 2.8: Speedup of conventional [2] and proposed FPGA implementations of the K-Means algorithm over GPU implementation [3].

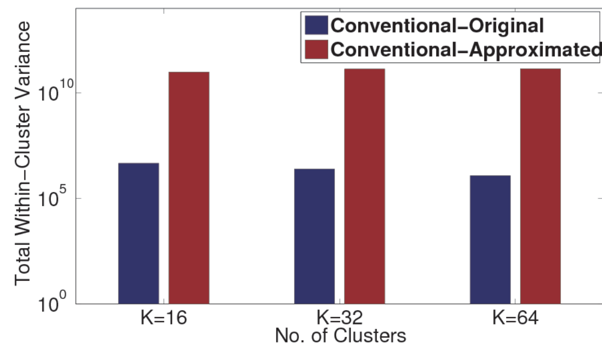
To compare our work with the conventional hardware design in terms of area efficiency, the proposed design is implemented on Xilinx XC4VLX25, the same FPGA device used in [28]. Table 2.3 Shows the number of slices occupied by the Centroids Update unit of the proposed design, and those occupied by the Divider used to update clusters centroids in the conventional design in [28]. To have a valid comparison, we do not compare the complete implementations since the proposed design targets one-dimensional datasets, while the conventional design in [28] is implemented to cluster a multi-dimensional dataset. As discussed in Section 5, the two designs differ in the Accumulation/Count and Centroids Update units. The proposed design requires half the number of registers in the Count unit compared to Accumulation unit of the conventional design. As shown in Table 2.3, the divider used in the conventional design occupied 8% of the available slices compared to 3% occupied by the Centroids Update unit of the proposed design. The area occupied only by divider is equal to the total area occupied by the proposed design, as shown in Table 2.1.

Table 2.3: Area occupied by the Divider and Centroids Update units in the conventional and proposed designs, Respectively.

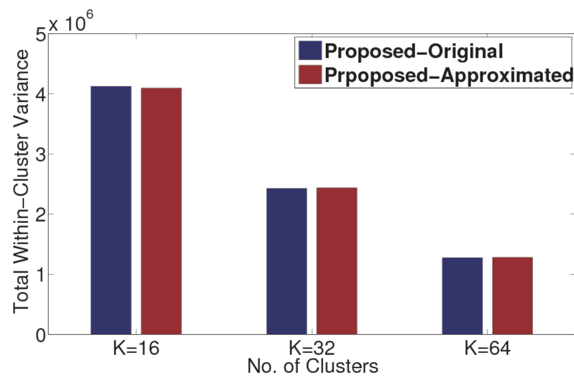
| **Area** | **Divider Conventional Design [28]** | **Proposed Centroids Update Unit** |
|---|---|---|
| No. of Slices | 967 | 332 |
| Utilization | 8% | 3% |

To determine the effect of the approximation adopted in the proposed design on the

quality of clustering results, we calculated the total within-cluster variance, a commonly used quality measure, for both the approximated and original implementations of the proposed and conventional designs. In the approximated implementations, the division operation in centroid update equations (2.6) for the conventional design, (2.9) and (2.12) for the proposed design, is implemented as a shift right by the nearest power of 2 integer to $n_j$. As shown in Figure 2.9, the proposed design is less sensitive to the error results from approximating $n_j$ compared to the conventional design. For the proposed design, the quality of clustering results using the approximated implementation is very close to that of the original division-based implementation.




(a) Conventional design.




(b) Proposed design.

Figure 2.9: Total Within-Cluster Variance for the approximated, shift-based, and original, division-based, Centroids Update units.

# Chapter 3

# 2-D Processor Array Architectures for Similarity Distance Computation

## 3.1 Introduction

Several machine learning and data mining algorithms require calculating certain types of distances such as Euclidean, Manhattan, and Cosine distances as a similarity measure between feature vectors [22]. In these algorithms, distance calculation consumes a significant portion of the computation time that is required to process large and high-dimensional datasets in applications involving big data analytics [5]. This task is intensively used in a wide range of algorithms such as Support Vector Machine (SVM) [30], $K$-Nearest Neighbours (KNN) [9], and $K$-Means [31] algorithms. The performance of such algorithms can be significantly improved by accelerating the distance calculation part.

The advances in VLSI technology has made it possible to implement compute-intensive kernels of machine learning algorithms in hardware. Several hardware architectures for such kernels have been proposed in the literature [32–36]. As one of the important operations in machine learning and data mining, similarity distance computation has been considered for hardware acceleration. In [4], a linear (1-D) processor array architecture for similarity distance computation has been proposed as part of a serial input VLSI clustering analyzer. The speed of their proposed architecture is slower than other 2-D serial input architectures since input data points are applied in a feature-serial format. The target was to reduce the circuit complexity of the distance calculation module. Another linear processor array has been proposed in [37]. Three similarity measures have been implemented, and an algorithm to assign computation tasks to the processing elements has been developed.

Compared to 2-D processor arrays, linear arrays generally require less area. On the other hand, 2-D processor arrays are expected to be much faster as a result of increasing the number of processing elements that work in parallel. In [38], a distance calculation module has been proposed as part of a VLSI architecture for a clustering analyzer. The distance calculation module has been designed as a 2-D processor array architecture of $K \times N$ processing elements, where $K$ is the number of clusters and $N$ is the number of data points in the input dataset. For large datasets, a huge number of processing elements are required and all $N$ data points must be fed simultaneously to the system; hence, the proposed design is not amenable for hardware implementation. To overcome the drawbacks of this architecture, the authors of [39] proposed a serial-input 2-D processor array architecture of $K \times M$ processing elements, where $K$ is the number of clusters and $M$ is the number of features. In the proposed architecture, the features of one data point is applied to the distance calculation module at a time.

In this chapter, we present a systematic methodology for exploring the design space of 2-D processor array architectures for similarity distance computation. Previous architectures proposed in the literature have been obtained using ad hoc techniques that do not allow for design space exploration. To obtain practical designs that are amenable for hardware implementation, the size and dimensionality of the input datasets are taken into consideration. The methodology presented in this chapter is used to obtain the 3-D computation domain of the similarity distance computation algorithm. A scheduling function determines whether an algorithm variable is pipelined or broadcast. Four linear scheduling functions are presented, and six possible 2-D processor array architectures are obtained and classified based on the size and dimensionality of the input datasets.

## 3.2 Problem Formulation

Given two M-dimensional datasets $\mathbf{X}$ of $N$ data points and $\mathbf{Y}$ of $K$ data points, that are represented as two matrices of dimensions $M \times N$ and $K \times M$, respectively. Distance matrix $\mathbf{D}$ of dimension $K \times N$ can be calculated using any distance measure where element $D(k,n)$ of the distance matrix represents the distance between the $k^{th}$ data point of dataset $\mathbf{Y}$ and the $n^{th}$ data point of dataset $\mathbf{X}$. As a special case, distance matrix can be calculated between data points of the same dataset taken pairwise. In this case, the distance matrix is a symmetric square matrix of dimension $N \times N$ with zero entries on the diagonal, and elements on the upper triangle need only to be calculated [37]. In this work, and without loss of generality, we will use Manhattan distance as the similarity measure between data

points of the two datasets $\mathbf{X}$ and $\mathbf{Y}$, which is calculated as:

$$D(k,n) = \sum_{m=0}^{M-1} |X(m,n) - Y(k,m)| \tag{3.1}$$

$$0 \leq n < N, \quad 0 \leq k < K,$$

where $N$ is the number of data points of dataset $\mathbf{X}$, $K$ is the number of data points of dataset $\mathbf{Y}$, and $M$ represents the number of features or dimensions.

A well-known example of algorithms that require calculating the distance matrix in the same way is the $K$-means clustering algorithm [31]. In this algorithm, distances between $N$ data points of an input dataset and the centroids of $K$ clusters have to be calculated to assign data points to their closest cluster. Algorithm 3.1 shows the standard procedure for calculating the Manhattan distance matrix between two matrices $\mathbf{X}$ and $\mathbf{Y}$. The computation can be implemented as three nested loops, and hence the algorithm has three indices $k$, $m$, and $n$.

**Algorithm 3.1** Standard procedure for Manhattan distance calculation.

**Inputs:** Two matrices $\mathbf{X}$ and $\mathbf{Y}$ of dimensions $M \times N$ and $K \times M$, respectively.
**Output:** Distance matrix $\mathbf{D}$ of dimension $K \times N$.

1: **for** $n = 0 \quad to \quad N - 1$ **do**
2:     **for** $k$=0 `to` $K - 1$ **do**
3:       $D(k,n) = 0;$
4:         **for** $m$=0 `to` $M - 1$ **do**
5:             $D(k,n) += |X(m,n) - Y(k,m)|$
6:         **end for**
7:     **end for**
8: **end for**

## 3.3 A Systematic Methodology for Processor Array Design

Systematic methodologies to design processor arrays allow for design space exploration and performance optimization according to certain design specifications and constraints. Several methodologies were proposed in the literature [40], [41], [42], [43]. Most of these methodologies are not suitable for mapping high-dimensional algorithms onto processor arrays. The starting point of the majority of these methodologies is the development of a

data dependence graph (DG) between the input and output variables. A dependence graph is suitable for algorithms with computational domains of dimension 3 at most. F. Gebali et al. have proposed a systematic methodology that is able to deal with algorithms of arbitrary dimensions [43]. A formal algebraic procedure was used to map a 6-dimensional algorithm of 3-D digital filter onto an efficient processor array architecture. The proposed methodology thereafter has been used to systematically design processor array architectures for algorithms from different fields such as pattern matching [44], motion estimation [45], and computer arithmetic [46–52]. In this work, we develop processor array architectures for similarity distance computation in machine learning algorithms. The remaining of this chapter presents the procedure employed to explore the design space of the problem to get efficient processor array implementations.

### 3.3.1 The 3-D Computation Domain and Domain Boundaries

The computation domain $\mathcal{D}$ for the distance calculation algorithm is defined by the algorithm indices and the limits imposed on these indices [1]. Since the algorithm has three indices, the computation domain is a volume in the 3-D integer space $\mathbb{Z}^3$. The three indices $k$, $m$, and $n$ represent the coordinates of a point in the computation domain. We organize the indices in the form of a vector or point $\mathbf{p} \in \mathbb{Z}^3$:

$$\mathbf{p} = \begin{bmatrix} k & m & n \end{bmatrix}^t . \tag{3.2}$$

The limits imposed on the algorithm indices define the upper and lower hulls of the computation domain, as shown in Figure 3.1.

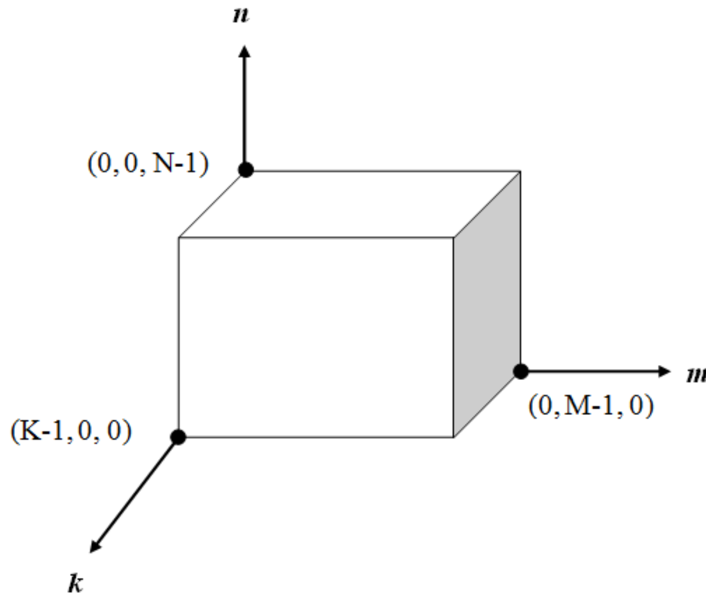



Figure 3.1: The 3-D computation domain $\mathcal{D}$ for the distance calculation algorithm.

### 3.3.2 The Dependence Matrices

Rather than analyzing data dependencies of the algorithm using the traditional way of studying how output variables of the algorithm depend on the inputs, we study how each variable depends on the algorithm indices. The dependence matrix of a variable $v$ that depends on $i$ out of $j$ indices of the algorithm is an integer matrix $\mathbf{A}_v$ of dimension $i \times j$ where $i \leq j$. The set of nullvectors of $\mathbf{A}_v$ defines a subdomain $\mathcal{B} \subset \mathcal{D}$ that is called the *broadcast subdomain* since every point in the subdomain $\mathcal{B}$ sees the same instance of variable $v$. The basis vectors of the broadcast subdomain are the nullspace of the dependence matrix [1].

Output variable $\mathbf{D}$ depends on indices $k$ and $n$ of the algorithm. Hence, its dependence matrix is given by the $2 \times 3$ integer matrix:

$$\mathbf{A_D} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.3}$$

The three elements in each row of the dependence matrix refer to the ordered algorithm indices $k$, $m$, and $n$. The first row shows that variable $\mathbf{D}$ depends on index $k$, and the second row shows that the variable depends on index $n$. The nullspace basis vector of matrix $\mathbf{A_D}$ could be given by:

$$\mathbf{e_D} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^t. \tag{3.4}$$

As shown in Figure 3.2(a), the broadcast subdomain for the output instance $D(c_1, c_2)$ is calculated using all the points in the computation domain $\mathcal{D}$ whose indices are $(c_1, m, c_2)$, where $0 \leq m < M$.
The dependence matrix of the input variable $X(m, n)$ is given by:

$$\mathbf{A_X} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.5}$$

and the associated nullspace basis vector could be given by:

$$\mathbf{e_X} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^t. \tag{3.6}$$

The dependence matrix of the input variable $Y(k, m)$ is given by:

$$\mathbf{A_Y} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \tag{3.7}$$

and the associated nullspace basis vector could be given by:

$$\mathbf{e_Y} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^t. \tag{3.8}$$

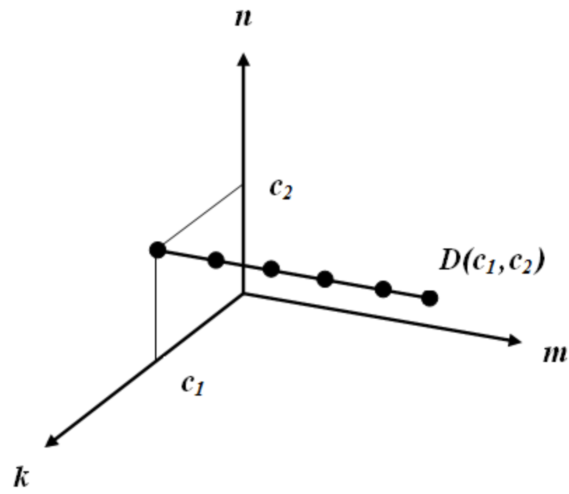
The broadcast subdomains for input variables $X(m, n)$ and $Y(k, m)$ are shown in Figure 3.2(b) and Figure 3.2(c), respectively.

### 3.3.3 Data Scheduling

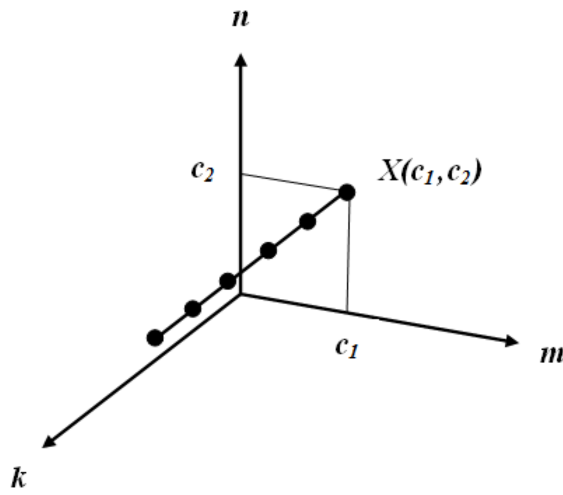
Algorithm variables can be either pipelined or broadcast by determining a scheduling function that assigns each point in the computation domain a time value. When a variable is broadcast, all points that belong to its subdomain are assigned the same time value. A pipelined variable, on the other hand, has all points in the computation domain that belong to its subdomain assigned different time values.

A simple scheduling function that can be used for scheduling the algorithm tasks is the affine scheduling function of the form [1]:

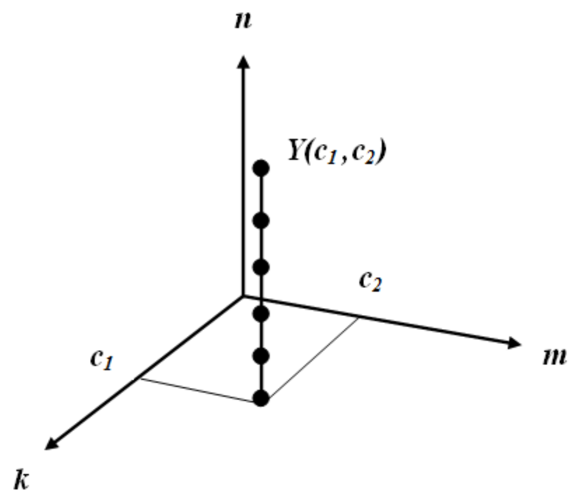
$$t(\mathbf{p}) = \mathbf{sp}, \tag{3.9}$$




(a) Subdomain for output variable $D(c_1, c_2)$.




(b) Subdomain for input variable $X(c_1, c_2)$.




(c) Subdomain for input variable $Y(c_1, c_2)$.

Figure 3.2: The broadcast subdomains for algorithm variables.

where the function $t(\mathbf{p})$ assigns a time value to a point $\mathbf{p} \in \mathcal{D}$, and $\mathbf{s} = \begin{bmatrix} s_1 & s_2 & s_3 \end{bmatrix}$ is the scheduling vector.

If we choose to broadcast a variable whose nullvector is $\mathbf{e}$, we must have [1]:

$$\mathbf{se} = 0. \tag{3.10}$$

To pipeline a variable whose nullvector is **e**, the following inequality must be satisfied [1]:

$$\mathbf{se} \neq 0. \tag{3.11}$$

These two restrictions provide the minimum constraints that must be satisfied to obtain a valid scheduling function. The broadcast restriction (3.10) preclude any broadcast directions that coincide with the projection directions defined in the following subsection.

In this work, we choose to pipeline our output variable **D** since broadcasting an output variable results in slower architectures that require calculating the output value from all partial results in the same clock cycle. From (3.4) and (3.11), we can write

$$\begin{bmatrix} s_1 & s_2 & s_3 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \neq 0, \tag{3.12}$$

which implies $s_2 \neq 0$. Let us choose $s_2 = 1$. Our scheduling vector so far is given by:

$$\begin{bmatrix} s_1 & 1 & s_3 \end{bmatrix}. \tag{3.13}$$

As one design alternative, let us broadcast the two input variables **X** and **Y**. To satisfy broadcast restriction (3.10), and from (3.6) and (3.8), we must have:

$$\begin{bmatrix} s_1 & 1 & s_3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 0, \tag{3.14}$$

which implies that $s_1 = 0$, and:

$$\begin{bmatrix} 0 & 1 & s_3 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0. \tag{3.15}$$

which implies that $s_3 = 0$. Thus, a valid scheduling vector that satisfies the above restric-

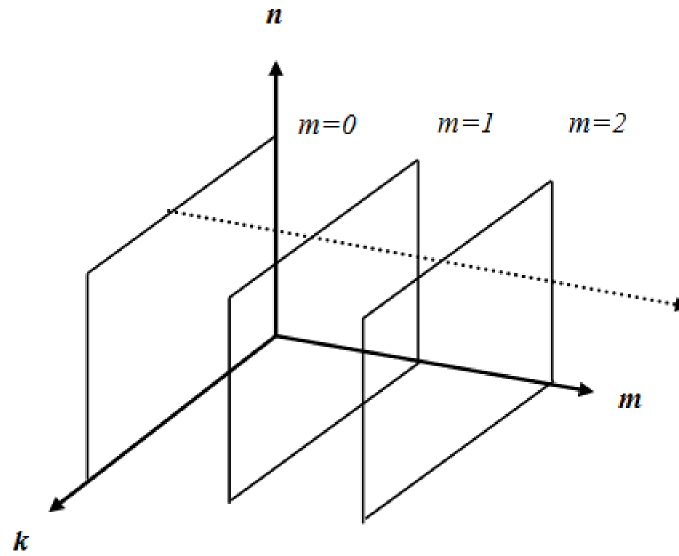



Figure 3.3: Equitemporal zones using linear scheduling and scheduling vector is [0 1 0].

tions can be given by:

$$\mathbf{s_1} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}. \tag{3.16}$$

The scheduling function determines the computational load to be performed by the system at each time step. The above calculated scheduling vector given by (3.16) assigns all points with the same value of coordinate $m$ the same time value. All points in each plane in Figure 3.3 are assigned the same time value and said to belong to the same equitemporal zone [1]. The following scheduling vectors can be calculated the same way for other design choices as shown in Table 3.1:

$$\mathbf{s_2} = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \tag{3.17}$$

$$\mathbf{s_3} = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \tag{3.18}$$

$$\mathbf{s_4} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}. \tag{3.19}$$

### 3.3.4 Projection Operation

The computations associated with different points in the computation domain are executed in different time steps. Mapping each point in the computation domain to a single PE results in poor hardware utilization since a PE is active only for one time instant and idle the rest of the time [1]. To improve the utilization of hardware resources, a projection operation is used to map several points of the computation domain to a single processing element (PE). In [43], the authors explained how to obtain a projection matrix $\mathbf{P}$ and use it to perform the projection operation. To find the projection matrix, a projection direction vector $\mathbf{d}$ has to be defined. A valid projection direction vector belongs to the null space of matrix $\mathbf{P}$ and must satisfy the inequality [43]:

$$\mathbf{sd} \neq 0. \tag{3.20}$$

The following vectors represent valid projection directions that satisfy (3.20) for the scheduling function in (3.16):

$$\mathbf{d_1} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^t \tag{3.21}$$

$$\mathbf{d_2} = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix}^t \tag{3.22}$$

$$\mathbf{d_3} = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix}^t \tag{3.23}$$

$$\mathbf{d_4} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^t. \tag{3.24}$$

To keep the control hardware of the processor array architectures simple, we only choose projection directions along the basis vectors of $\mathcal{D}$. Only projection direction vectors with a single 1 in their elements are chosen. Table 3.1 shows the valid scheduling vectors and their associated possible projection directions. Design space exploration for these scheduling and projection direction vectors will be discussed in the following section.

Table 3.1: Design Space Exploration of 2-D Processor Array Architectures for Similarity Distance Computation.

| **Scheduling Vector** | **Possible Projection Directions** | **Design Choices** | | |
|---|---|---|---|---|
| | | **Output D** | **Input X** | **Input Y** |
| $s_1$=[0 1 0] | $d_{11}$=[0 1 0] | Pipelined | Broadcast | Broadcast |
| $s_2$=[0 1 1] | $d_{21}$=[0 0 1]<br>$d_{22}$=[0 1 0] | Pipelined | Broadcast | Pipelined |
| $s_3$=[1 1 0] | $d_{31}$=[0 1 0]<br>$d_{32}$=[1 0 0] | Pipelined | Pipelined | Broadcast |
| $s_4$=[1 1 1] | $d_{41}$=[0 0 1]<br>$d_{42}$=[0 1 0]<br>$d_{43}$=[1 0 0] | Pipelined | Pipelined | Pipelined |

## 3.4 Design Space Exploration

Previous works in [38] and [39] have not taken into consideration the size and dimensionality of the input datasets. To achieve practical designs that are feasible for hardware implementation, we classify the resulting processor array architectures for the similarity distance computation problem into two groups depending on the value of $N$ relative to $M$.

### 3.4.1 Case 1: When $N \gg K, M$

In this case, we perform design space exploration for processor array architectures that are suitable for calculating the similarity distance between dataset $\mathbf{X}$ of large size $N$ compared to the size of dataset $\mathbf{Y}$ and the datasets dimension $M$. From Table 3.1, we choose projection directions $\mathbf{d_{21}}$ and $\mathbf{d_{41}}$ of value $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$. These projection direction vectors ensure that the $n$-axis will be eliminated after the projection operation, and the projected computation domain will have $K \times M$ points. Hence, the resulting processor array is a 2-D array of $K \times M$ PEs. The corresponding projection matrix $\mathbf{P}$ could be given by:

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}. \tag{3.25}$$

Hence, a point $\mathbf{p} = \begin{bmatrix} k & m & n \end{bmatrix}^t \in \mathcal{D}$ maps to point $\bar{\mathbf{p}}$ in the projected computation domain $\bar{\mathcal{D}}$:

$$\bar{\mathbf{p}} = \mathbf{P}\mathbf{p} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = \begin{bmatrix} k \\ m \end{bmatrix}. \tag{3.26}$$

#### 3.4.1.1 Design #1: Using $\mathbf{s_2} = [0 \quad 1 \quad 1]$ and $\mathbf{d_{21}} = [0 \quad 0 \quad 1]^t$

Using the scheduling vector $\mathbf{s_2} = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix}$, the time value associated with each point $\mathbf{p}$ in the computation domain is given by:

$$t(\mathbf{p}) = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = m + n. \tag{3.27}$$

Input variable $\mathbf{X}$ is broadcast. Input sample $X(m, n)$ is supplied at time $m + n$. This implies that all elements with the same sum of row and column indices are fed simultaneously. Element $X(0, 0)$ is fed at time 0. Elements $X(0, 1)$ and $X(1, 0)$ are fed at time 1,

and so on. $M(M-1)/2$ delay registers are required to ensure feeding data elements of matrix $\mathbf{X}$ as imposed by the scheduling function. The broadcast direction for the input data is mapped to the vector $\bar{\mathbf{e_X}}$ given by:

$$\bar{\mathbf{e_X}} = \mathbf{P}\mathbf{e_X} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{3.28}$$

Broadcast lines of variable $\mathbf{X}$ will map to horizontal lines along the $k$-axis in the projected architecture, as shown in Figure 3.4.

According to Table 3.1, input variable $\mathbf{Y}$ is pipelined. The pipeline direction for the input data is mapped to the vector $\bar{\mathbf{e_Y}}$ given by:

$$\bar{\mathbf{e_Y}} = \mathbf{P}\mathbf{e_Y} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{3.29}$$

This means that input variable $\mathbf{Y}$ is localized. Each processing element PE$(k, m)$ in the projected architecture will store a single element, $Y(k, m)$, of the input matrix $\mathbf{Y}$.

Output variable $\mathbf{D}$ is Pipelined. The pipeline direction for the output data is mapped to the vector $\bar{\mathbf{e_D}}$ given by:

$$\bar{\mathbf{e_D}} = \mathbf{P}\mathbf{e_D} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{3.30}$$

Output data will map to vertical lines along the $m$-axis in the projected architecture, as shown in Figure 3.4. For each of these lines, we need to specify a feeding point, to initialize the pipeline, and an extraction point to extract the output from. We choose to initialize the pipelines at PEs with $m = 0$ and extract outputs from PEs with $m = M - 1$. Hence, the initialization point for output $D(c_1, c_2)$ is:

$$\begin{bmatrix} c_1 & 0 & c_2 \end{bmatrix}^t, \tag{3.31}$$

that will map to the point:

$$\bar{\mathbf{p}} = \mathbf{P}\mathbf{p} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ 0 \\ c_2 \end{bmatrix} = \begin{bmatrix} c_1 \\ 0 \end{bmatrix}, \tag{3.32}$$

and the extraction point is:

$$\begin{bmatrix} c_1 & M-1 & c_2 \end{bmatrix}^t, \tag{3.33}$$

that will map to the point:

$$\bar{\mathbf{p}} = \mathbf{P}\mathbf{p} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ M-1 \\ c_2 \end{bmatrix} = \begin{bmatrix} c_1 \\ M-1 \end{bmatrix}. \tag{3.34}$$

The extraction point for an instance $D(k,n)$ of this variable could be given by:

$$\mathbf{p} = \begin{bmatrix} k & M-1 & n \end{bmatrix}^t. \tag{3.35}$$

The time value for this element is:

$$t(\mathbf{p}) = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} k \\ M-1 \\ n \end{bmatrix} = n + M - 1. \tag{3.36}$$

This implies that all elements of the output matrix $\mathbf{D}$ on the same column are obtained from the processing elements at the same time. The first column, $n = 0$, is obtained at time $M-1$, the second column is obtained at time $M$, and so on. The number of time steps required to complete the computation of the distance matrix is $M + N$.

The resulting processor array architecture is shown in Figure 3.4, and the processing element structure is shown in Figure 3.5. Register $D(k,n)_m$ in the processing element stores the calculated partial result to be passed to PE($k, m+1$), and $D(k,n)_{m-1}$ is the partial result passed from PE($k, m-1$).

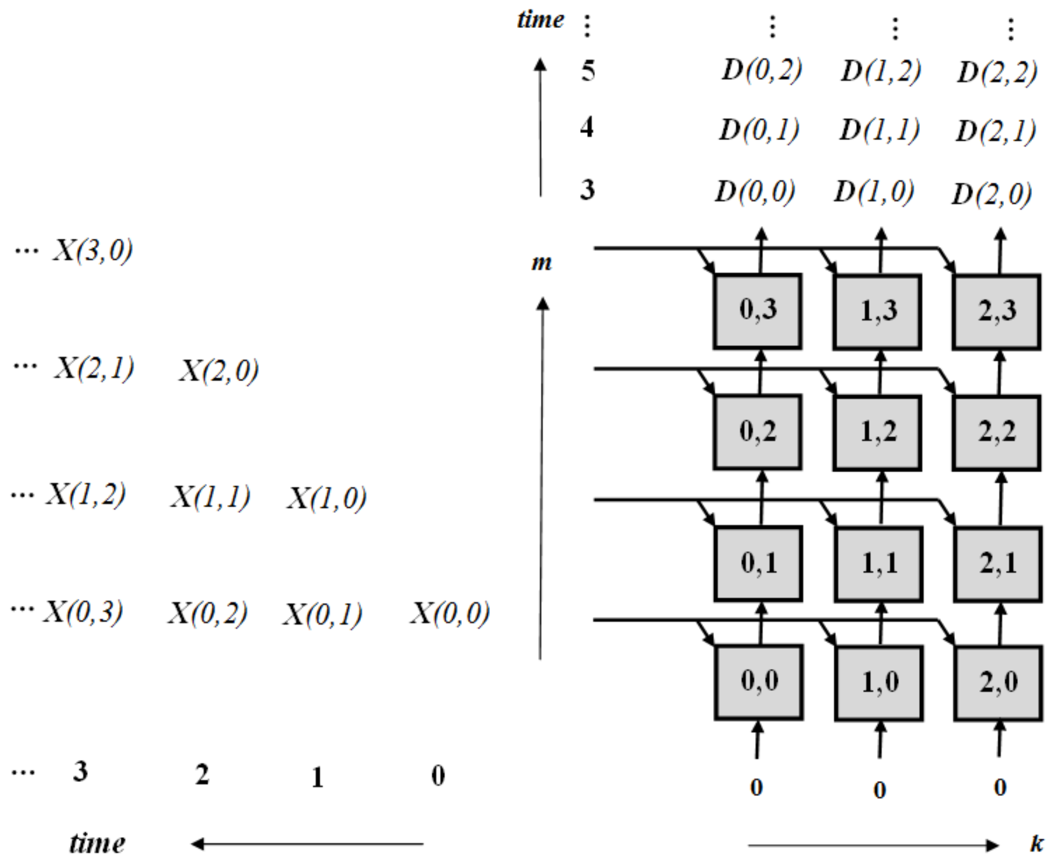



Figure 3.4: Processor array architecture for Design #1 when $K = 3$ and $M = 4$.

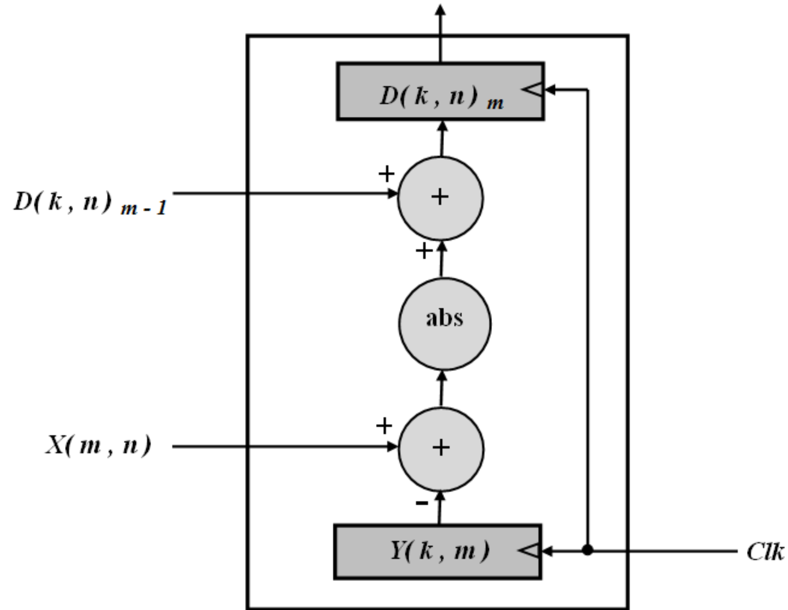



Figure 3.5: Processing element for Design #1 in Figure 3.4.

#### 3.4.1.2 Design #2: Using $\mathbf{s_4} = [1 \quad 1 \quad 1]$ and $\mathbf{d_{41}} = [0 \quad 0 \quad 1]$

According to Table 3.1, input variable $\mathbf{X}$ is pipelined. Since projection direction $\mathbf{d_{41}}$ equals projection direction $\mathbf{d_{21}}$ used in Design #1, the pipeline direction for the input data will map to horizontal lines along the $k$-axis in the projected architecture, as shown in Figure 3.6. We choose to feed the pipelined inputs at PEs with $k = 0$. Hence, the feeding point for input $X(c_1, c_2)$ is:

$$\begin{bmatrix} 0 & c_1 & c_2 \end{bmatrix}^t, \tag{3.37}$$

that will map in the projected architecture to the point:

$$\bar{\mathbf{p}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 \\ c_1 \end{bmatrix}. \tag{3.38}$$

The time value associated with each point in the computation domain is given by:

$$t(\mathbf{p}) = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = k + m + n. \tag{3.39}$$

The feeding point for an instance $X(m, n)$ could be given by the point:

$$\mathbf{p} = \begin{bmatrix} 0 & m & n \end{bmatrix}^t. \tag{3.40}$$

The time value assigned to this point is:

$$t(\mathbf{p}) = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ m \\ n \end{bmatrix} = m + n. \tag{3.41}$$

This implies that all elements with the same sum of row and column indices are fed simultaneously. Element $X(0, 0)$ is fed at time 0. Elements $X(0, 1)$ and $X(1, 0)$ are fed at time 1, and so on.

Input $\mathbf{Y}$ is Localized and output $\mathbf{D}$ is pipelined as in Design #1. The time value assigned to the extraction point $\mathbf{p} = \begin{bmatrix} k & M-1 & n \end{bmatrix}^t$ of output instance $D(k, n)$ is:

$$t(\mathbf{p}) = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} k \\ M-1 \\ n \end{bmatrix} = k + n + M - 1. \tag{3.42}$$

This implies that all elements of the output matrix $\mathbf{D}$ with the same sum of row and column indices are obtained from the processing elements at the same time. Element $D(0, 0)$ is obtained at time $M - 1$. Elements $D(0, 1)$ and $D(1, 0)$ are obtained at time $M$, and so on. The number of time steps required to complete the computation of the distance matrix is $K+M+N$. The resulting processor array architecture is shown in Figure 3.6. $M(M-1)/2$ delay registers are required to ensure feeding data elements of matrix $\mathbf{X}$ as imposed by the scheduling function. Processing element structure is similar to that of Design #1 shown in Figure 3.5 with an extra register to store pipelined data elements of input variable $\mathbf{X}$ to be passed to PE($k + 1, m$) in the next time step.

### 3.4.2 Case 2: When $M \gg K, N$

In this case, we perform design space exploration for processor array architectures that are suitable for high dimensional low sample size (HDLSS) data [53]. DNA microarray data is an example of such datasets that have typically a small number of samples with a large number of genes [54]. From Table 3.1, we choose designs with projection direction $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$. This projection direction ensures that the $m$-axis will be eliminated after the

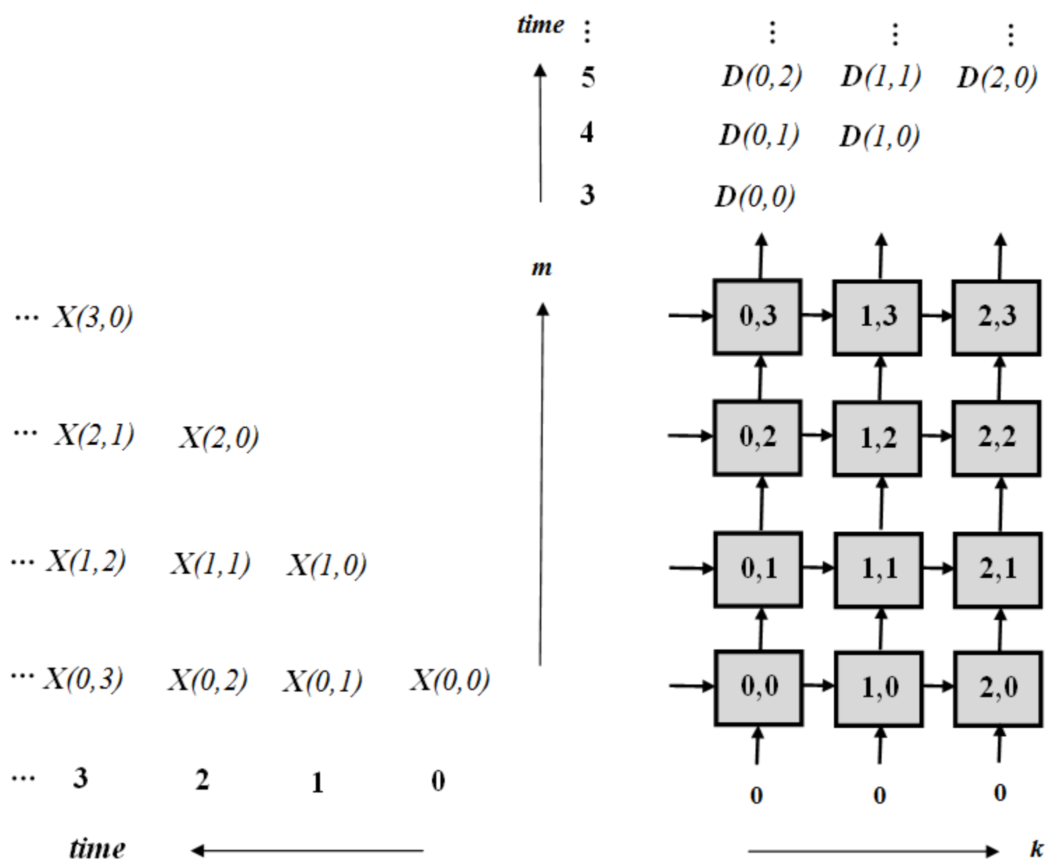



Figure 3.6: Processor array architecture for Design #2 when $K = 3$ and $M = 4$.

projection operation, and the projected computation domain will have $K \times N$ points. The corresponding projection matrix $\mathbf{P}$ could be given by:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.43}$$

As a result of this projection, any point $\mathbf{p} = \begin{bmatrix} k & m & n \end{bmatrix}^t$ in the computation domain $\mathcal{D}$ will map to the point:

$$\bar{\mathbf{p}} = \mathbf{P}\mathbf{p} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = \begin{bmatrix} k \\ n \end{bmatrix}. \tag{3.44}$$

Four processor array architectures are obtained using projection directions $\mathbf{d_{11}}$, $\mathbf{d_{22}}$, $\mathbf{d_{31}}$, and $\mathbf{d_{42}}$ from Table 3.1. Following the same algebraic mapping methodology used in the previous subsection, output variable $\mathbf{D}$ is found to be localized in all of the obtained architectures.

#### 3.4.2.1 Design #3: Using $\mathbf{s_1} = [0 \quad 1 \quad 0]$ and $\mathbf{d_{11}} = [0 \quad 1 \quad 0]$

The time value associated with each point in the computation domain is given by:

$$t(p) = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = m. \tag{3.45}$$

Both input variables $\mathbf{X}$ and $\mathbf{Y}$ are broadcast. $M$ time steps are required to calculate all elements of the distance matrix. The processing element structure is shown in Figure 3.7, and the resulting processor array architecture is shown in Figure 3.8.

#### 3.4.2.2 Design #4: Using $\mathbf{s_2} = [0 \quad 1 \quad 1]$ and $\mathbf{d_{22}} = [0 \quad 1 \quad 0]$

The time value associated with each point in the computation domain is given by:

$$t(p) = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = m + n. \tag{3.46}$$

Input variable $\mathbf{X}$ is broadcast and input variable $\mathbf{Y}$ is pipelined. $M + N$ time steps are required to calculate all elements of the distance matrix. The structure of the processing

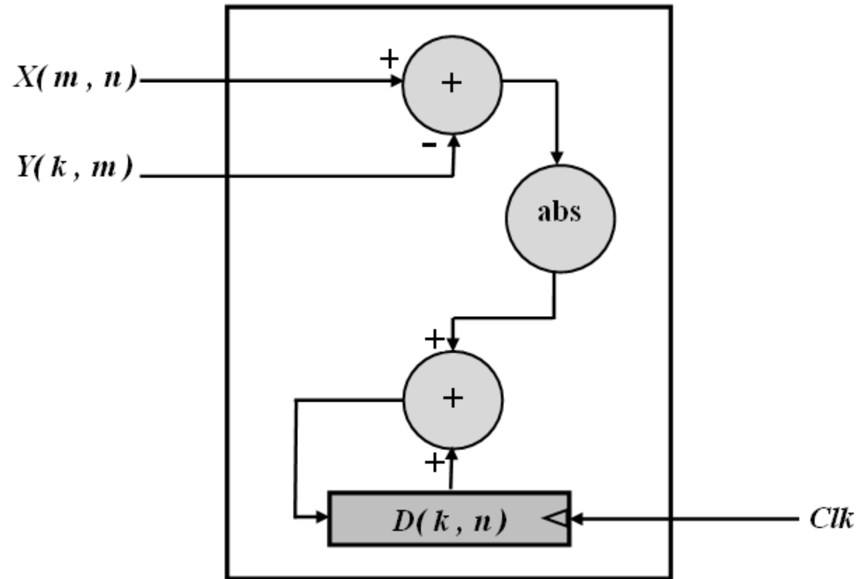



Figure 3.7: Processing element for Design #3 in Figure 3.8.

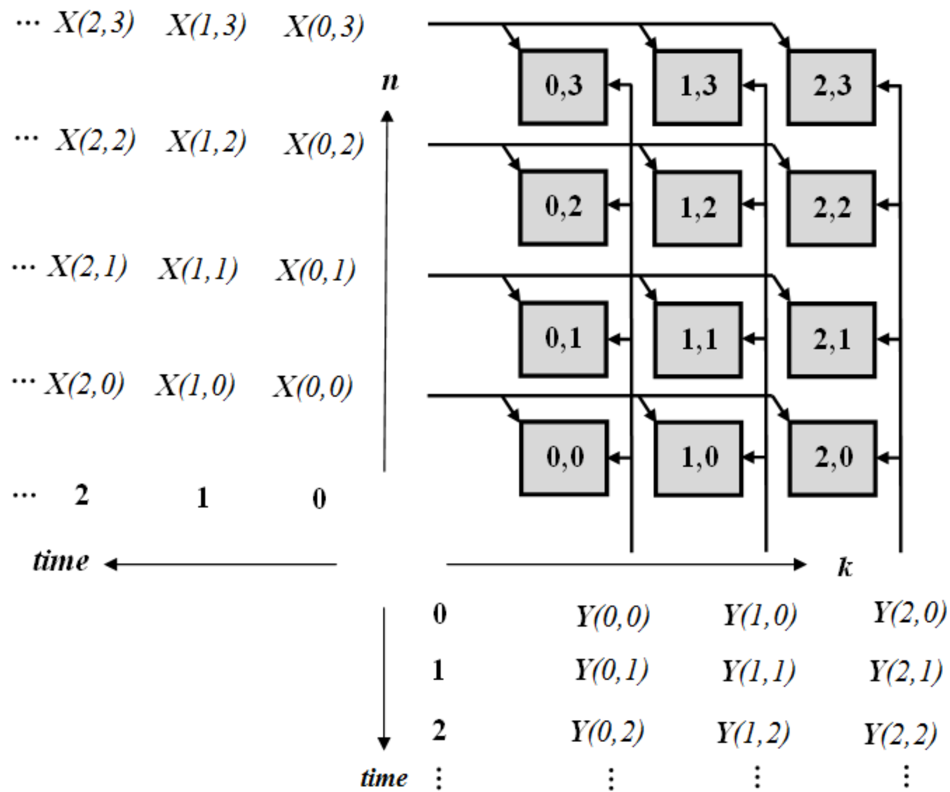



Figure 3.8: Processor array architecture for Design #3 when $K = 3$ and $N = 4$.

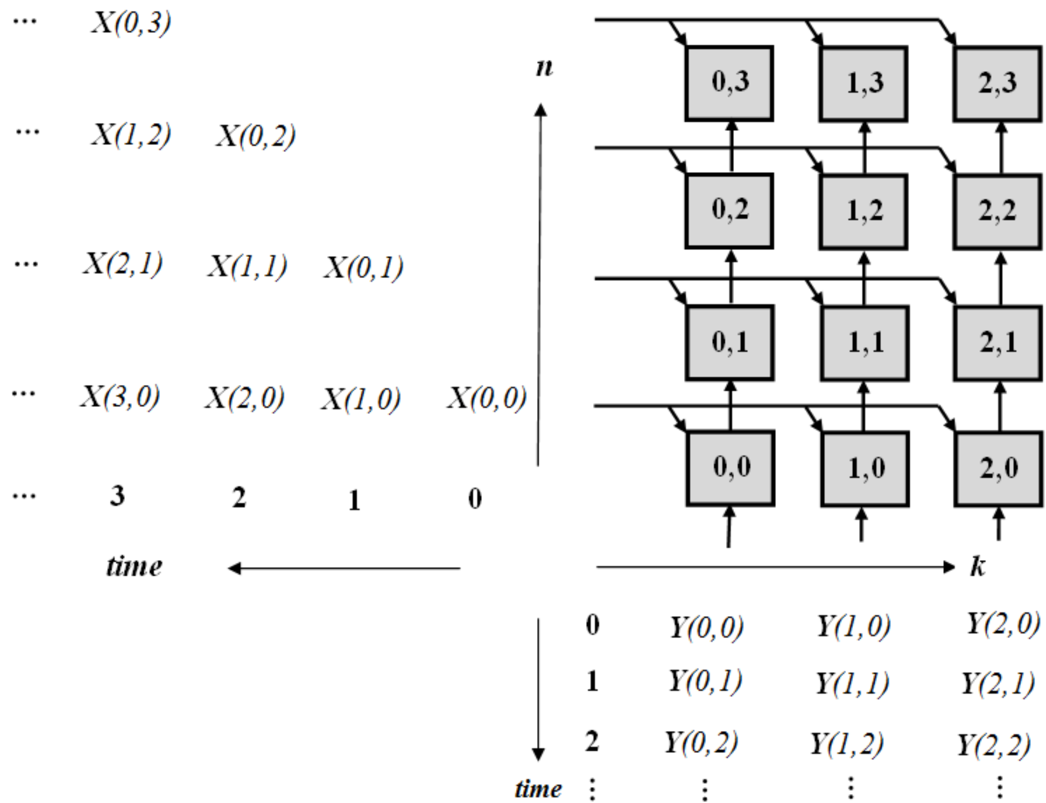



Figure 3.9: Processor array architecture for Design #4 when $K = 3$ and $N = 4$.

element is similar to that of Design #3 shown in Figure 3.7 with an extra register to store pipelined data elements of input variable $\mathbf{Y}$. The resulting processor array architecture is shown in Figure 3.9. $N(N-1)/2$ delay registers are required to feed data elements of matrix **X**.

#### 3.4.2.3 Design #5: Using $\mathbf{s_3} = [1 \quad 1 \quad 0]$ and $\mathbf{d_{31}} = [0 \quad 1 \quad 0]$

The time value associated with each point in the computation domain is given by:

$$t(p) = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = k + m. \tag{3.47}$$

Input variable $\mathbf{X}$ is pipelined and input variable $\mathbf{Y}$ is broadcast. $K + M$ time steps are required to calculate all elements of the distance matrix. The structure of the processing element is similar to that of Design #3 shown in Figure 3.7 with an extra register to store pipelined data elements of input variable $\mathbf{X}$. The resulting processor array architecture is

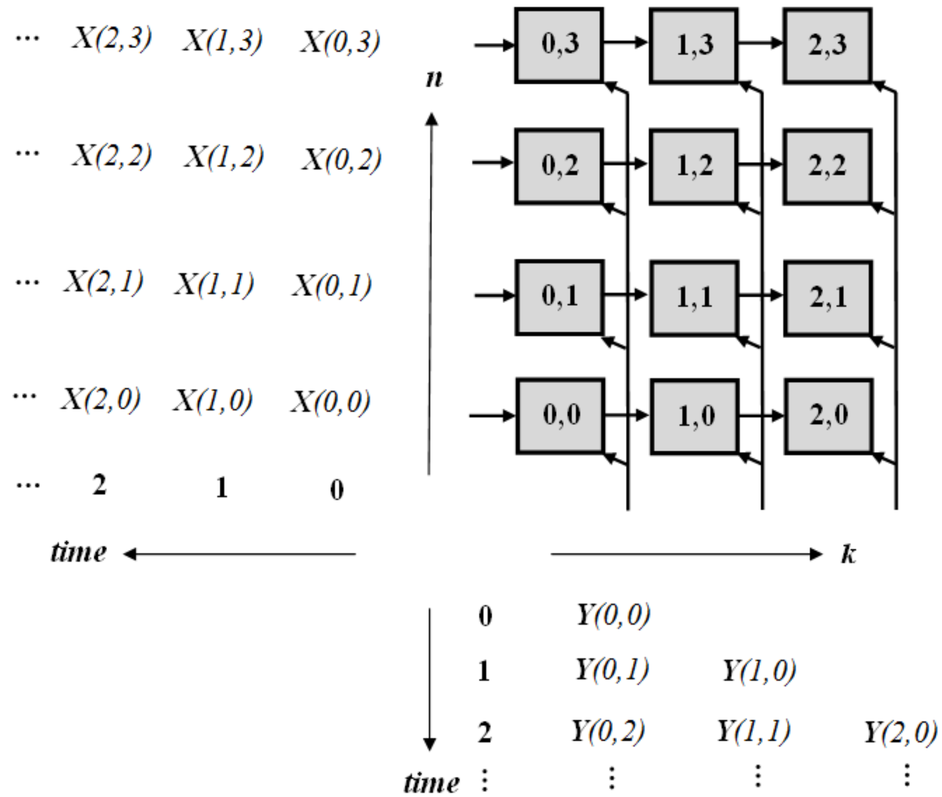



Figure 3.10: Processor array architecture for Design #5 when $K = 3$ and $N = 4$.

shown in Figure 3.10. $K(K-1)/2$ delay registers are required to feed data elements of matrix $\mathbf{Y}$.

#### 3.4.2.4 Design #6: Using $\mathbf{s}_4 = [1 \quad 1 \quad 1]$ and $\mathbf{d}_{42} = [0 \quad 1 \quad 0]$

The time value associated with each point in the computation domain is given by:

$$t(p) = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = k + m + n. \tag{3.48}$$

Both input variables $\mathbf{X}$ and $\mathbf{Y}$ are pipelined. $K+M+N$ time steps are required to calculate all elements of the distance matrix. The structure of the processing element is similar to that of Design #3 shown in Figure 3.7 with two extra registers to store pipelined data elements of input variables $\mathbf{X}$ and $\mathbf{Y}$. The resulting processor array architecture is shown in Figure 3.11. $N(N-1)/2$ delay registers and $K(\mathrm{K}{-}1)/2$ delay registers are required to feed data elements of matrices $\mathbf{X}$ and $\mathbf{Y}$, respectively.

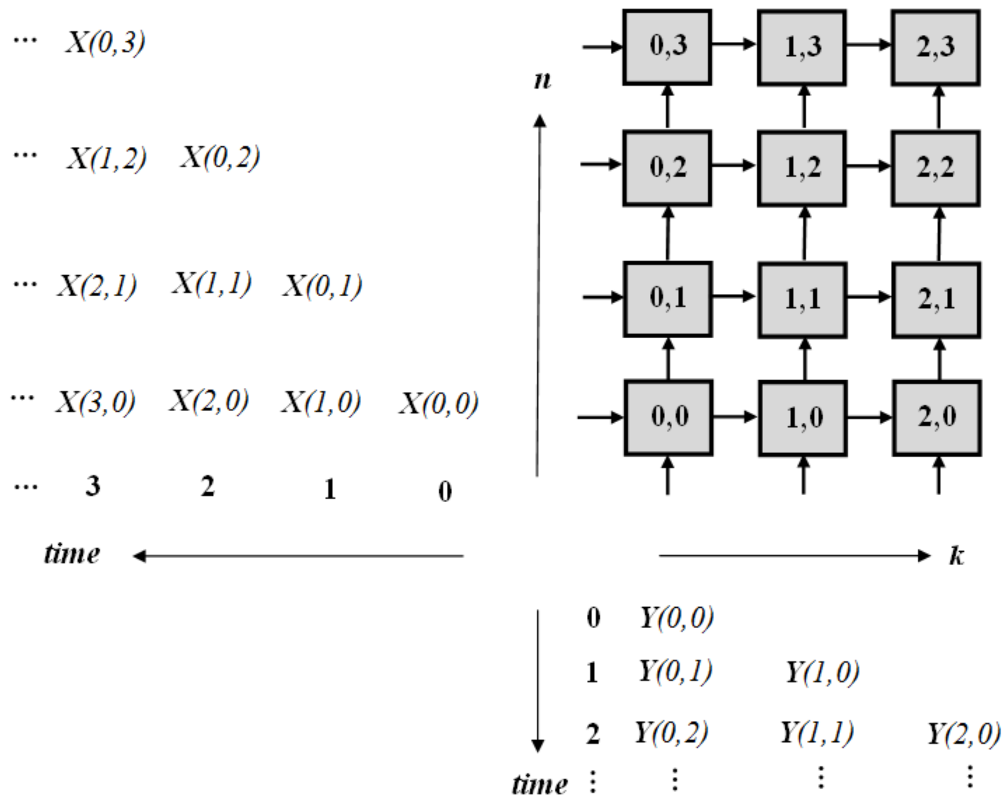



Figure 3.11: Processor array architecture for Design #6 when $K = 3$ and $N = 4$.

### 3.4.3 Case 3: When $K \gg M, N$

Two more designs that are similar to Designs #1 and #2 could be obtained for input datasets with $K >> M, N$. Projection direction vectors $\mathbf{d_{32}}$ and $\mathbf{d_{43}}$ of value $\begin{bmatrix}1 & 0 & 0\end{bmatrix}$ ensure that the $k$-axis will be eliminated with $M \times N$ points in the projected computation domain.

## 3.5 Design Comparison

The systematic methodology we employed to obtain our processor array architectures allowed us to explore possible timing and mapping options. The ad hoc techniques used to design previous architectures do not allow for design space exploration. Only one design has been developed without explaining how it was obtained. In addition to the new proposed architectures, the employed methodology was also able to obtain previously obtained architectures. Design #2 is identical to the design obtained in [39], and Design #6 is identical to the one obtained in [38]. Table 3.2 summarizes the circuit and time complexities of the six designs obtained in Section 4.

Design #1 has better time and circuit complexities than Design #2 obtained in [39]. Only $M + N$ time steps are required compared to $K + M + N$ steps required by Design #2. Each processing element in Design #2 has an extra register to store the pipelined data elements of input $\mathbf{X}$.

Among the four designs of Case 2 in Section 4, Design #3 has the optimum time and circuit complexities. Only $M$ time steps are required to complete the computation of the distance matrix. No delay registers are required, and the structure of the processing element is the simplest. Both input variables $\mathbf{X}$ and $\mathbf{Y}$ are not pipelined, and no more registers are required. Design #6 obtained in [38] has the worst time and circuit complexities among the four designs.

The clock cycle time $T_c$ for the processing elements shown in Figure 3.5 and Figure 3.7 can be modeled as the delay of the subtraction, absolute value, and addition operations which is given by:

$$T_c = 3w\tau, \tag{3.49}$$

where $w$ is the size of data elements in bits, and $\tau$ is the delay of a 1-bit full adder.

The number of time steps required to calculate the distance matrix on a single processor is $K \times M \times N$ steps, and the speedup of any of the obtained designs is given by:

Table 3.2: Circuit and time complexities of the obtained processor array architectures.

| **Design** | **Circuit Complexity** | **Time Complexity** |
|---|---|---|
| **Case 1:** $N \gg K, M$ | | |
| Design #1 | $K \times M$ PEs<br>$\frac{1}{2}M(M-1)$ Registers | $M + N$ |
| Design #2 | $K \times M$ PEs<br>$\frac{1}{2}M(M-1)$ Registers | $K + M + N$ |
| **Case 2:** $M \gg K, N$ | | |
| Design #3 | $K \times N$ PEs | $M$ |
| Design #4 | $K \times N$ PEs<br>$\frac{1}{2}N(N-1)$ Registers | $M + N$ |
| Design #5 | $K \times N$ PEs<br>$\frac{1}{2}K(\mathrm{K}-1)$ Registers | $K + M$ |
| Design #6 | $K \times N$ PEs<br>$\frac{1}{2}(K^2 + N^2 - \mathrm{K} - N)$ Registers | $K + M + N$ |

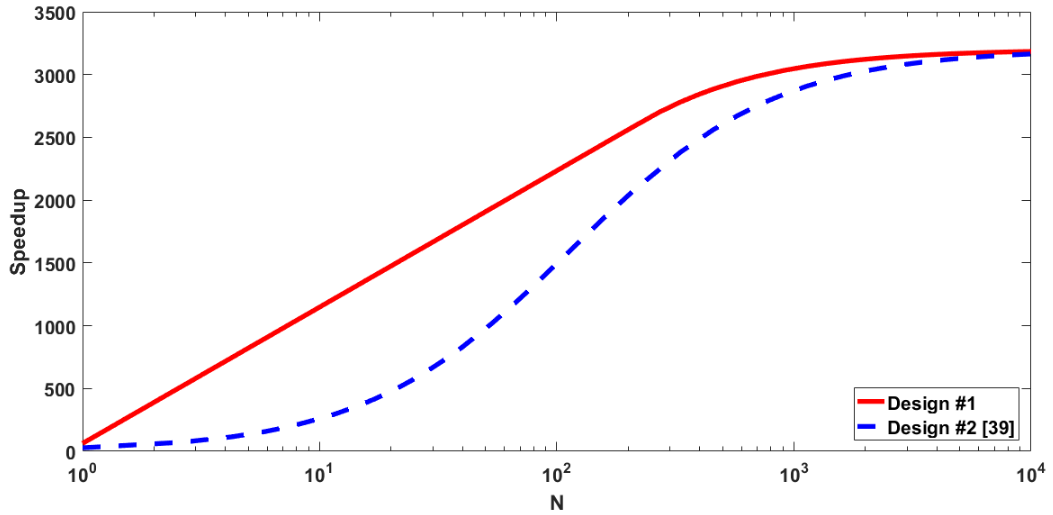



Figure 3.12: Analytical speedups of Case 1 architectures for $K = 64$ and $M = 50$.

$$Speedup = \frac{T_{Sequential}}{T_{Parallel}}, \tag{3.50}$$

where $T_{Sequential}$ is the time required to calculate the distance matrix on a single processor that is given by:

$$T_{Sequential} = T_c(K \times M \times N), \tag{3.51}$$

and $T_{Parallel}$ represents the time required to calculate the distance matrix on any of the proposed processor array architectures. The speedup of Design #1, for example, is given by:

$$Speedup = \frac{T_c(K \times M \times N)}{T_c(M + N)} = \frac{K \times M \times N}{M + N}. \tag{3.52}$$

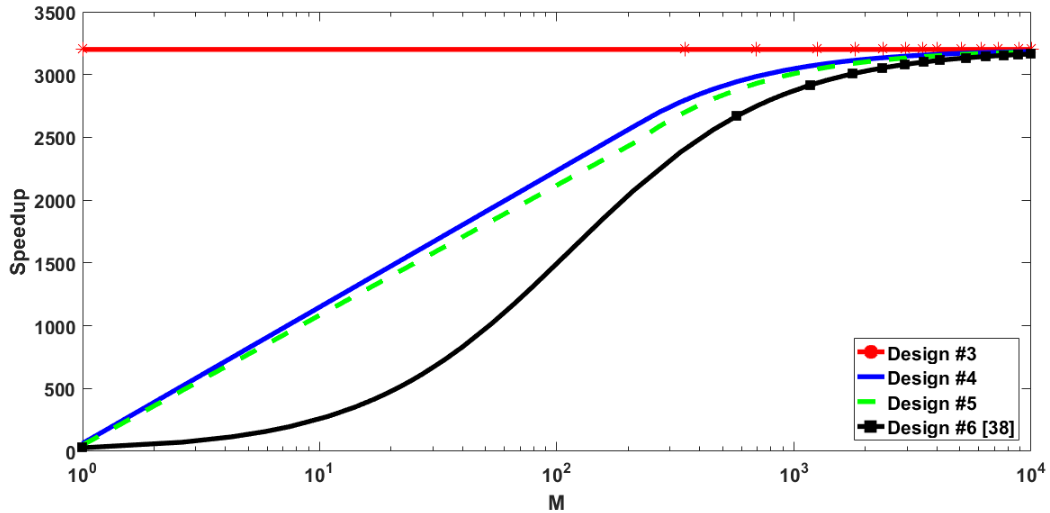



Figure 3.13: Analytical speedups of Case 2 architectures for $K = 64$ and $N = 50$.

Analytical speedups of the designs obtained for Case 1 and Case 2 in the previous section are shown in Figure 3.12 and Figure 3.13, respectively. As shown in the figures, Design #1 has higher speedup than Design #2 for variable sizes of input dataset $\mathbf{X}$, and Design #3 has the best speedup for variable dimensions of the input datasets.

One real-world application of similarity distance computation using the proposed architectures is cluster analysis for microarray gene expression data. A microarray dataset is represented by a real-valued matrix with a large number of rows, that represent genes, and a few number of columns, that represent samples. For instance, similarity distance computation involved in clustering the Leukemia dataset [55] using $K$-means algorithm can be accelerated using our proposed designs. This dataset consists of 7,129 genes and 72 samples. In this application, it is meaningful to cluster either genes or samples [56].

The gene-based clustering for the Leukemia dataset [55] using $K$-means clustering with $K = 2$ results in system parameters of $K = 2$, $M = 72$ and $N = 7,129$. For this situation we have $N \gg K, M$ and Case 1 in Table 3.2 applies and Design #1 gives the best performance. The corresponding analytical speedup of distance computation involved in one iteration of the algorithm is 142.

The sample-based clustering for the Leukemia dataset [55] using $K$-means clustering with $K = 2$ results in system parameters of $K = 2$, $M = 7,129$ and $N = 72$. For this situation we have $M \gg K, N$ and Case 2 in Table 3.2 applies and Design #3 gives the best performance. The corresponding analytical speedup of distance computation involved in one iteration of the algorithm is 144.

## 3.6 Discussion

Linear scheduling and projection operations employed in this work proved to be helpful for exploring the design space of the similarity distance computation problem. These linear operations allow us to control the number of active processors at a given time instance and also allow us to control the number of time steps needed to complete the algorithm.

The resulting processor array might require high I/O bandwidth, which affect the system buses and number of I/O pins. This could prove problematic for high-dimensional datasets that require a large number of features ($M$) or samples ($N$) being fed to the processor array simultaneously.

The linear scheduling and projection operations might require input data broadcast among the processors. Data broadcast for large values of $K$, $M$ or $N$ might imply high fanout and this will impact system clock rate. Clock rates of the actual implementations of the proposed architectures can be affected by the delay of long broadcast buses $T_b$ that can be modeled as [57]:

$$T_b = RC \times \frac{m(m+1)}{2}, \tag{3.53}$$

where $R$ and $C$ are the parasitic resistance and capacitance of one section of the bus between two adjacent processing elements, respectively, and $m$ is the number of processing elements attached to the bus [44].

# Chapter 4

# Linear Processor Array Architectures for Similarity Distance Computation

## 4.1 Introduction

Linear (1-D) processor arrays are generally more suitable for area, power, and bandwidth-constrained applications compared to 2-D arrays at the cost of slower speed. In [4], a linear processor array for the computation of similarity distance has been proposed. The proposed architecture is used to calculate similarity distances between data samples of an input dataset and clusters centroids in a VLSI clustering analyzer. Input data samples are fed in a feature-serial format along the $m$-axis. The proposed architecture has higher time complexity than other 2-D processor arrays in [38] and [39]. However, both area complexity and number of I/O pins have been reduced. In [37], the authors proposed an FPGA-based linear processor array for similarity measures computation. The proposed architecture is used to calculate three similarity measures among all samples of an input dataset. Most of the existing processor arrays in the literature have been designed in an ad hoc manner. Data dependencies have not been analyzed, and only one design alternative is considered for tasks scheduling and mapping.

In this chapter, we extend the systematic technique presented in Chapter 3 to design linear processor arrays for the computation of similarity distance matrices. The employed technique is used to define the computation domain of the algorithm. Time restrictions on input and output variables are introduced in order to meet area and bandwidth constraints. Six scheduling vectors are calculated, and six possible design alternatives are obtained and analyzed in terms of area and time.

## 4.2 Data Scheduling

Our strategy for arriving at suitable scheduling functions combines broadcast restriction (3.10) and pipelining restriction (3.11) presented in Chapter 3, page 37. These restrictions are the minimum constraints that can be used to get a valid scheduling function. We start by choosing to pipeline the evaluation of all points that lie in a plane perpendicular to one of the three $k$-, $m$-, or $n$-axes. Next we pipeline the evaluation of all points that lie on lines in the chosen plane. These lines are parallel to one of the remaining two axes in that plane. Finally we broadcast the evaluation of all points in the chosen line, as shown in Figure 4.1 for the first scheduling vector discussed in the following subsection. In total, we have three axes to choose the planes and two directions to choose the lines in the planes. This gives rise to six possible scheduling functions. Subsection 4.2.1 illustrates how this technique is used to derive our first scheduling vector $\mathbf{s}_1$.

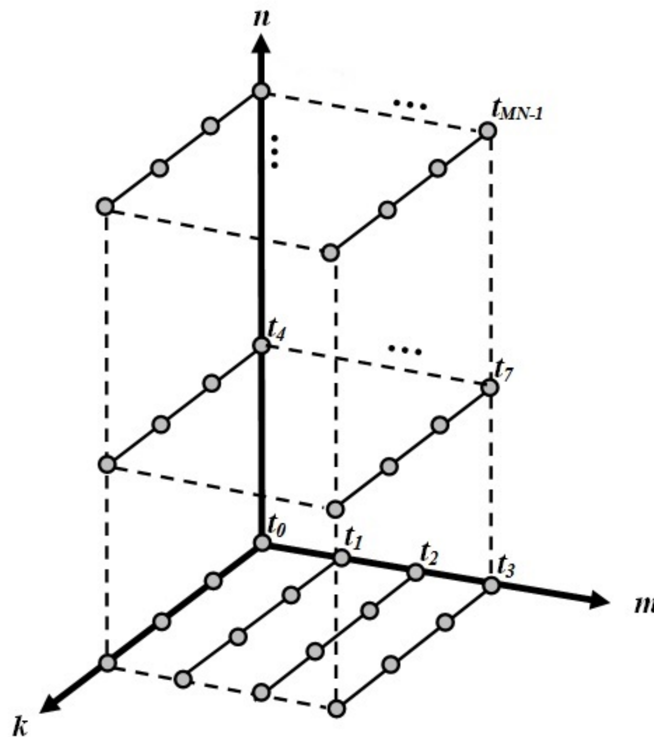



Figure 4.1: Equitemporal zones for scheduling vector $\mathbf{s_1}$

### 4.2.1 Calculation of the first scheduling vector $\mathbf{s}_1$

Let us choose to broadcast input variable $\mathbf{X}$. From (3.6) and (3.10), we have:

$$\begin{bmatrix} s_1 & s_2 & s_3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 0 \tag{4.1}$$

which implies $s_1$ = 0.

To avoid feeding large number of features simultaneously, we choose to supply input variable $\mathbf{X}$ in a feature-serial format (i.e., along the $m$-axis). This implies that for any data sample $n$, the time between the calculations for feature $m$ and feature $m+1$ is one time step:

$$\begin{bmatrix} 0 & s_2 & s_3 \end{bmatrix} \begin{bmatrix} k \\ m+1 \\ n \end{bmatrix} - \begin{bmatrix} 0 & s_2 & s_3 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = 1 \tag{4.2}$$

which implies $s_2$ = 1.

We choose to start the first calculation for sample $n+1$ after the last calculation for sample $n$. The time between these two calculations is also one time step:

$$\begin{bmatrix} 0 & 1 & s_3 \end{bmatrix} \begin{bmatrix} k \\ 0 \\ n+1 \end{bmatrix} - \begin{bmatrix} 0 & 1 & s_3 \end{bmatrix} \begin{bmatrix} k \\ M-1 \\ n \end{bmatrix} = 1 \tag{4.3}$$

which implies $s_3$ = $M$. Hence, the first valid scheduling vector is given by:

$$\mathbf{s_1} = \begin{bmatrix} 0 & 1 & M \end{bmatrix} \tag{4.4}$$

Referring to Figure 4.1, the calculated scheduling vector $\mathbf{s_1}$ results in assigning all points on each of the continuous lines the same time value. These lines are called equitemoporal zones since the computations for all points on each line are performed simultaneously [1]. From the geometric perspective, scheduling vector $\mathbf{s_1}$ results in executing all points in a plane with a fixed value of coordinate $n$ before points in the plane with next value of $n$. Within each plane, all points on a line with a fixed value of coordinate $m$ are executed before points on the line with next value of $m$. Points on each of these lines are executed in parallel.

### 4.2.2 Calculation of the remaining scheduling vectors

The remaining five scheduling vectors, out of the six possible scheduling vectors described earlier in this section, can be calculated using the same procedure employed to calculate $\mathbf{s_1}$ with different orders of execution along the three axes. In Figure 4.2, the equitemporal zones are lines along the $k$-axis with another order of execution. The associated scheduling vector is given by:

$$\mathbf{s_2} = \begin{bmatrix} 0 & N & 1 \end{bmatrix} \tag{4.5}$$

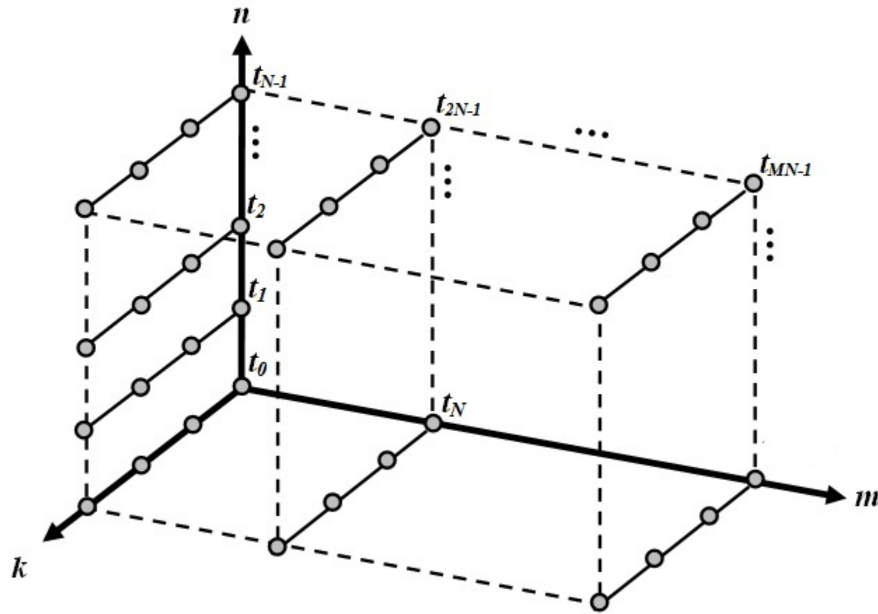



Figure 4.2: Equitemporal zones for scheduling vector $\mathbf{s_2}$

Another two timing alternatives with equitemporal zones along the $m$-axis are shown in Figure 4.3 and Figure 4.4. The associated scheduling vectors for these timing alternatives are given by:

$$\mathbf{s_3} = \begin{bmatrix} 1 & 0 & K \end{bmatrix} \tag{4.6}$$

and

$$\mathbf{s_4} = \begin{bmatrix} N & 0 & 1 \end{bmatrix} \tag{4.7}$$

Figure 4.5 and Figure 4.6 show two timing alternatives with equitemporal zones along the $n$-axis. The associated scheduling vectors are given by:

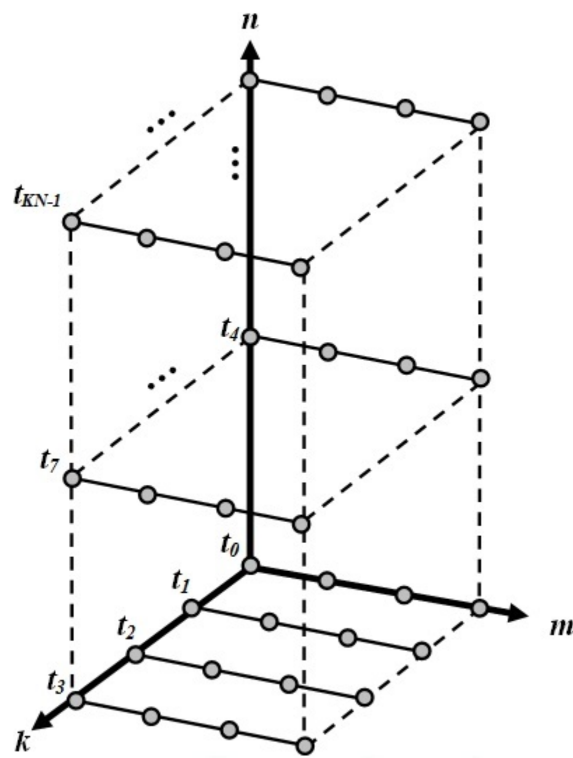



Figure 4.3: Equitemporal zones for scheduling vector $\mathbf{s_3}$

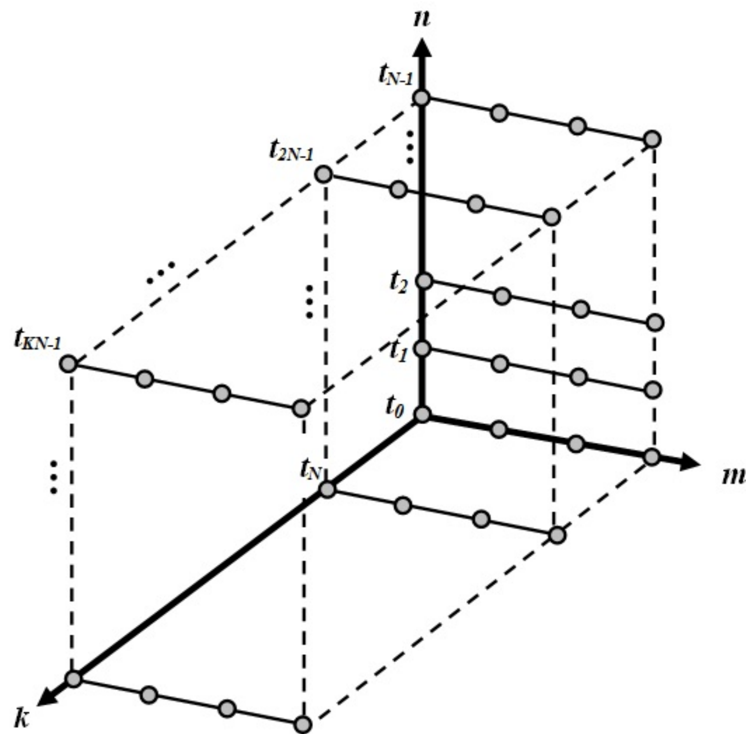



Figure 4.4: Equitemporal zones for scheduling vector $\mathbf{s_4}$

$$\mathbf{s_5} = \begin{bmatrix} M & 1 & 0 \end{bmatrix} \tag{4.8}$$

and

$$\mathbf{s_6} = \begin{bmatrix} 1 & K & 0 \end{bmatrix} \tag{4.9}$$

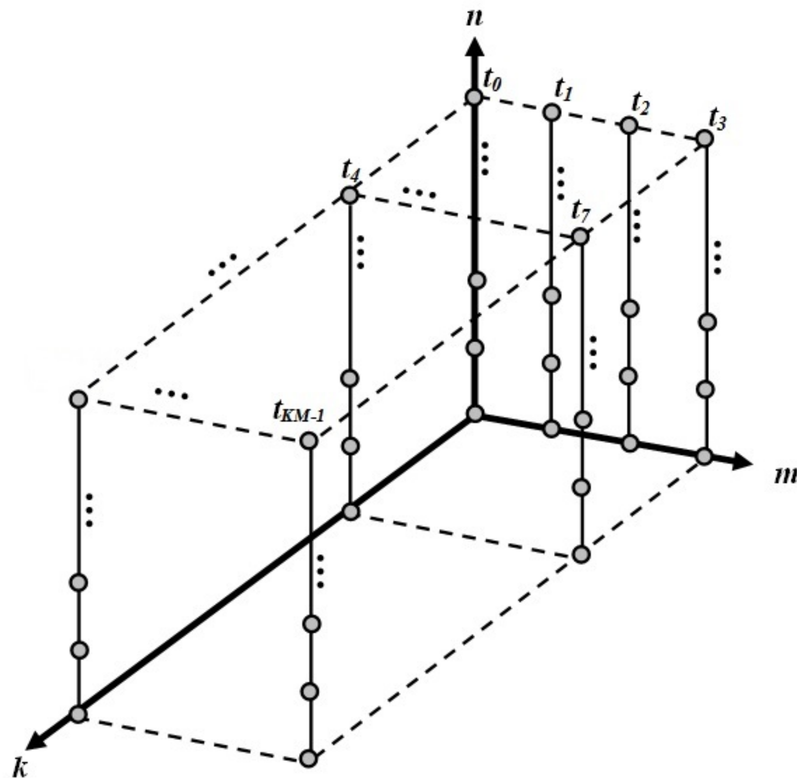



Figure 4.5: Equitemporal zones for scheduling vector $\mathbf{s_5}$

## 4.3 Projection Operation

Linear projection is defined as the mapping of several points in the $n$-dimensional computation domain $\mathcal{D}$ to a single point in a $k$-dimensional domain $\tilde{\mathcal{D}}$, where $k \leq n$. A projection matrix $\mathbf{P}$ that can be used to perform the projection operation can be obtained using a set of $l = (n - k)$ projection direction vectors that belong to the null space of the projection matrix and satisfy broadcast condition (3.20) presented in Chapter 3 page 39 [43].

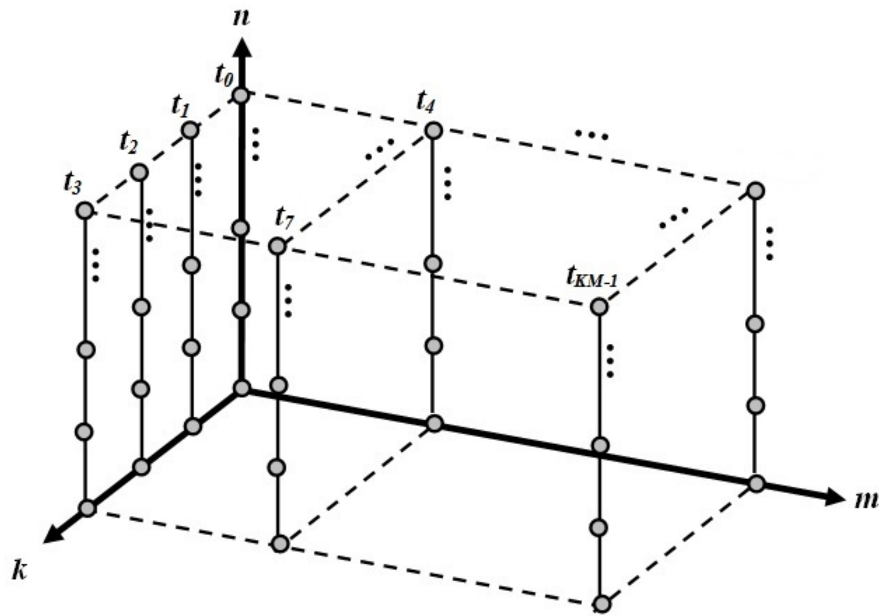



Figure 4.6: Equitemporal zones for scheduling vector $s_6$

In this chapter, our goal is to map the points in the 3-D computation domain shown in Figure 3.1 on page 34 to a 1-D domain. Hence, two projection direction vectors have to be specified for each of the six scheduling vectors presented in the previous section. For the scheduling vector $\mathbf{s_1} = \begin{bmatrix} 0 & 1 & M \end{bmatrix}$ and according to (3.20), two possible projection directions could be given by:

$$\mathbf{d_{11}} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^t \tag{4.10}$$

and

$$\mathbf{d_{12}} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^t \tag{4.11}$$

These projection directions are then used to calculate the associated projection matrix according to the procedure described in [43]:

$$\mathbf{P_1} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^t \tag{4.12}$$

Table 4.1 shows the chosen projection directions and the associated project matrices for the six obtained scheduling vectors.

## 4.4 Design Space Exploration

In this section, we will explore the design space of linear processor arrays for similarity distance computation using the calculated scheduling vectors and projection matrices in Table 4.1.

### 4.4.1 Design #1: using $\mathbf{s_1}$ = [0 1 $M$] and $\mathbf{P_1}$ = [1 0 0]

In this design option, each point $\mathbf{p} = \begin{bmatrix} k & m & n \end{bmatrix}^t \in \mathcal{D}$ is assigned a time value using the scheduling function:

$$t(\mathbf{p}) = \begin{bmatrix} 0 & 1 & M \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = m + Mn \tag{4.13}$$

The projection matrix $\mathbf{P}_1 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$ maps any point $\mathbf{p}$ in the computation domain to the point:

Table 4.1: Possible projection directions and associated projection matrices

| **Scheduling Vector** | **Chosen Projection Directions** | **Associated Projection Matrix** |
|---|---|---|
| $\mathbf{s_1} = [0 \;\; 1 \; M]$ | $\mathbf{d_{11}} = [0 \;\; 1 \;\; 0]^t$<br>$\mathbf{d_{12}} = [0 \;\; 0 \;\; 1]^t$ | $\mathbf{P}_1 = [1 \;\; 0 \;\; 0]$ |
| $\mathbf{s_2} = [0 \; N \; 1]$ | $\mathbf{d_{21}} = [0 \;\; 1 \;\; 0]^t$<br>$\mathbf{d_{22}} = [0 \;\; 0 \;\; 1]^t$ | $\mathbf{P}_2 = [1 \;\; 0 \;\; 0]$ |
| $\mathbf{s_3} = [1 \;\; 0 \; K]$ | $\mathbf{d_{31}} = [1 \;\; 0 \;\; 0]^t$<br>$\mathbf{d_{32}} = [0 \;\; 0 \;\; 1]^t$ | $\mathbf{P}_3 = [0 \;\; 1 \;\; 0]$ |
| $\mathbf{s_4} = [N \; 0 \;\; 1]$ | $\mathbf{d_{41}} = [1 \;\; 0 \;\; 0]^t$<br>$\mathbf{d_{42}} = [0 \;\; 0 \;\; 1]^t$ | $\mathbf{P}_4 = [0 \;\; 1 \;\; 0]$ |
| $\mathbf{s_5} = [M \; 1 \;\; 0]$ | $\mathbf{d_{51}} = [1 \;\; 0 \;\; 0]^t$<br>$\mathbf{d_{52}} = [0 \;\; 1 \;\; 0]^t$ | $\mathbf{P}_5 = [0 \;\; 0 \;\; 1]$ |
| $\mathbf{s_6} = [1 \; K \; 0]$ | $\mathbf{d_{61}} = [1 \;\; 0 \;\; 0]^t$<br>$\mathbf{d_{62}} = [0 \;\; 1 \;\; 0]^t$ | $\mathbf{P}_6 = [0 \;\; 0 \;\; 1]$ |

$$\tilde{\mathbf{p}} = \mathbf{P}_1\mathbf{p} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = k \tag{4.14}$$

which implies that the resulting processor array is a linear array along the $k$-axis with $K$ PEs. All points in the computation domain with the same $k$ coordinate will map to the same point, or PE in the projected computation domain. Input variable $\mathbf{X}$ is broadcast since the broadcast condition in (3.10) is satisfied and the broadcast direction is mapped to the vector:

$$\tilde{\mathbf{e}}_{\mathbf{X}} = \mathbf{P}_1\mathbf{e}_{\mathbf{X}} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 1 \tag{4.15}$$

which implies that input data is fed using broadcast lines along the $k$-axis in the projected architecture. Input variable $\mathbf{Y}$ is pipelined since the pipeline condition in (3.11) is satisfied. The pipeline direction is mapped to the vector:

$$\tilde{\mathbf{e}}_{\mathbf{Y}} = \mathbf{P}_1\mathbf{e}_{\mathbf{Y}} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0 \tag{4.16}$$

which implies that input $\mathbf{Y}$ is localized in the projected architecture. The $k^{th}$ PE only uses the $M$ features of the $k^{th}$ data sample of input matrix $\mathbf{Y}$. Output variable $\mathbf{D}$ is also pipelined, and the pipeline direction is mapped to the vector:

$$\tilde{\mathbf{e}}_{\mathbf{D}} = \mathbf{P}_1\mathbf{e}_{\mathbf{D}} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = 0 \tag{4.17}$$

which implies that output $\mathbf{D}$ is also localized. Every clock cycle, each PE calculates the absolute difference between the corresponding features of datasets $\mathbf{X}$ and $\mathbf{Y}$, and accumulates the result in a register. For every $M$ cycles, each PE generates the distance $\mathbf{D}(k, n)$ between the $n^{th}$ sample of dataset $\mathbf{X}$ and the $k^{th}$ sample of dataset $\mathbf{Y}$. $K$ distances are calculated in parallel by the $K$ PEs. Hence, the total number of time steps is $MN$ clock cycles. The time complexity of the proposed architecture can also be determined by calculating the time value assigned by the scheduling function in (4.13) to the point with upper limits coordinates:

$$t(\mathbf{p}_{max}) = \begin{bmatrix} 0 & 1 & M \end{bmatrix} \begin{bmatrix} K-1 \\ M-1 \\ N-1 \end{bmatrix} = MN - 1 \tag{4.18}$$

Since the first time value is zero, the total number of time steps is $t(\mathbf{p}_{max}) + 1 = MN$ steps. The resulting processor array and the structure of each PE are shown in Figure 4.7 and Figure 4.8, respectively.

The remaining processor array architectures are obtained in the following subsections using the same procedure utilized in this subsection to obtain Design #1.

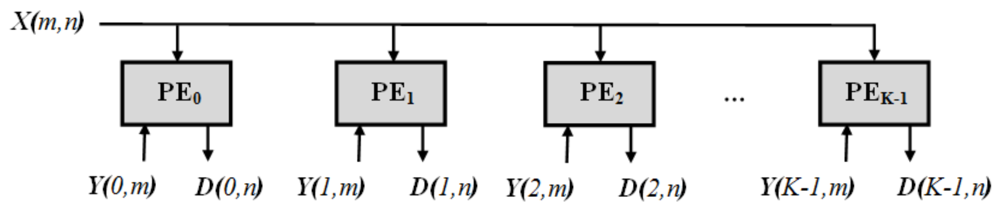



Figure 4.7: Processor array architecture for Design #1

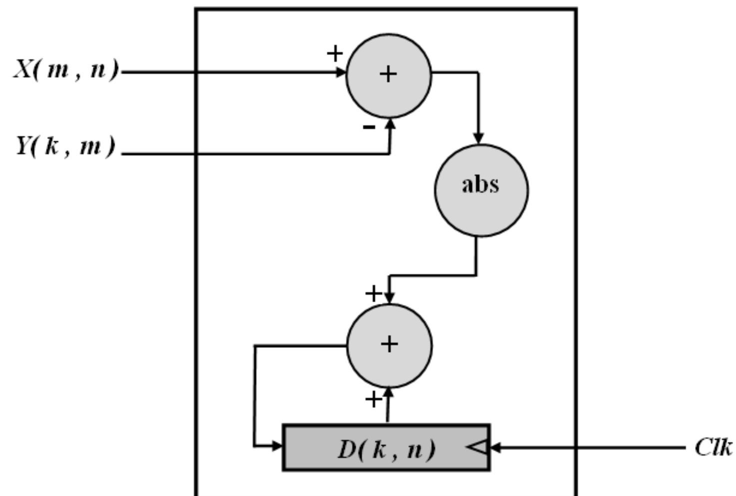



Figure 4.8: Processing element for Design #1 in Figure 4.7

### 4.4.2 Design #2: using $\mathrm{s}_2$ = [0 $N$ 1] and $\mathrm{P}_2$ = [1 0 0]

The projection matrix is the same as that of Design #1. Hence, all points in the computation domain will be mapped to a linear processor array of $K$ PEs similar to that in Figure 4.7 with variable $\mathbf{X}$ being broadcast and variables $\mathbf{Y}$ and $\mathbf{D}$ are localized. The chosen scheduling vector results in assigning each point in the computation domain the time value:

$$t(\mathbf{p}) = \begin{bmatrix}0 & N & 1\end{bmatrix} \begin{bmatrix}k \\ m \\ n\end{bmatrix} = Nm + n \tag{4.19}$$

The total number of time steps is also $MN$ steps. However, the scheduling vector imposes a different order of execution. $N$ computations for feature $m$ of all data samples are performed before the $N$ computations for feature $m+1$. Hence, $N$ registers are required by each PE to store the intermediate results compared to only one register in Design #1.

### 4.4.3 Design #3: using $\mathrm{s}_3$ =[1 0 $K$] and $\mathrm{P}_3$ =[0 1 0]

The scheduling function for this design alternative is:

$$t(\mathbf{p}) = \begin{bmatrix}1 & 0 & K\end{bmatrix} \begin{bmatrix}k \\ m \\ n\end{bmatrix} = k + Kn \tag{4.20}$$

Accordingly, the total number of time steps is $KN$ steps. Both input variables $\mathbf{X}$ and $\mathbf{Y}$ are localized, and output $\mathbf{D}$ is broadcast with its broadcast direction mapped to a line along the $m$-axis. The PE structure is shown in Figure 4.9.

Broadcasting an output variable requires that partial results from all PEs are used concurrently to generate one data element of the output matrix every clock cycle. An adder tree of depth $\lceil \log_2 M \rceil$ and $M-1$ adders can be used to calculate the sum of all partial results from the $M$ PEs, as shown in Figure 4.10. Compared to PE structure of Design #1 in Figure 4.8, there is no need to store partial results in a register since absolute differences from all PEs are added together in the same cock cycle.

### 4.4.4 Design #4: using $\mathrm{s}_4$ =[$N$ 0 1] and $\mathrm{P}_4$ =[0 1 0]

The scheduling function for this design choice is:

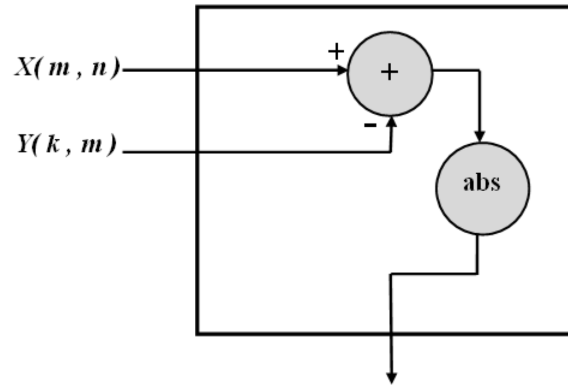



Figure 4.9: Processing element for Design #3

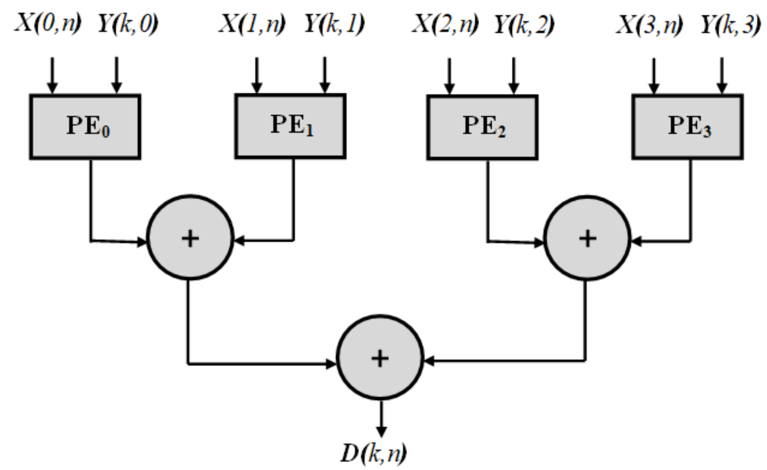



Figure 4.10: Processor array architecture for Design #3 with $M$=4.

$$t(\mathbf{p}) = \begin{bmatrix} N & 0 & 1 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = Nk + n \tag{4.21}$$

The time complexity for this design is equivalent to that of Design #3 which is $KN$ time steps. The PE structure and the processor array architecture are the same in Figure 4.9 and Figure 4.10, respectively. The main difference between the two designs is in the order of execution that results in generating elements of the output matrix $\mathbf{D}$ in a different order.

### 4.4.5 Design #5: using $\mathbf{s}_5$ = [$M$ 1 0] and $\mathbf{P}_5$ =[0 0 1]

The scheduling function for this design option is:

$$t(\mathbf{p}) = \begin{bmatrix} M & 1 & 0 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = Mk + m \tag{4.22}$$

The total number of time steps is $KM$ steps. Both input variable $\mathbf{X}$ and output variable $\mathbf{D}$ are localized. Input variable $\mathbf{Y}$ is broadcast with its broadcast direction mapped to a line along the $n$-axis. For every $M$ cycles, each PE generates the distance $\mathbf{D}(k, n)$ between the $n^{th}$ sample of dataset $\mathbf{X}$ and the $k^{th}$ sample of dataset $\mathbf{Y}$. $N$ distances are calculated in parallel by the $N$ PEs. The processor array architecture is shown in Figure 4.11 with the PE structure being the same as that of Design #1 in Figure 4.8.

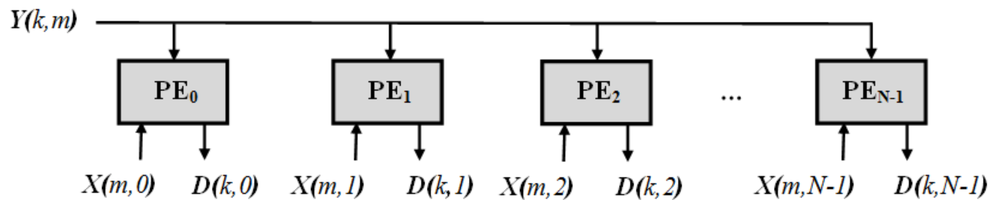



Figure 4.11: Processor array architecture for Design #5

### 4.4.6 Design #6: using $\mathbf{s}_6$ = [1 $K$ 0] and $\mathbf{P}_6$ = [0 0 1]

The scheduling function for this design alternative is:

$$t(\mathbf{p}) = \begin{bmatrix} 1 & K & 0 \end{bmatrix} \begin{bmatrix} k \\ m \\ n \end{bmatrix} = k + Km \tag{4.23}$$

The processor array architecture and its time complexity are the same as of Design #5. However, the order of execution that is imposed by the chosen scheduling vector dictates that $K$ registers are required by each PE to store the intermediate results compared to only one register in Design #5.

## 4.5 Comparison and Results

The systematic approach adopted in this chapter facilitates design space exploration of linear processor arrays for the similarity distance computation problem. The obtained architectures provide us with the flexibility to choose the one that meets hardware constraints for specific values of system parameters $K$, $M$, and $N$.

### 4.5.1 Design Comparison

Design #1 and Design #2 are suitable for parallelizing distance calculation tasks involved in processing dataset $\mathbf{X}$ of large size $N$ and dimension $M$, compared to $K$ that represents the size of dataset $\mathbf{Y}$. Distance calculation that is required for clustering data samples of a large-scale dataset $\mathbf{X}$ using K-Means algorithm, for example, fits these design options since the size of dataset $N$ and the number of features $M$ are much larger than the number of clusters $K$. Compared to Design #1, Design #2 is not practical since it has the same time complexity with a large number of additional registers to store intermediate results.

The systematic methodology adopted in this work can be used to obtain a previously devised architecture in [4]. This architecture is similar to Design #1 with input $\mathbf{X}$ being pipelined rather than broadcast. Scheduling vector $\mathbf{s_1}$ can be modified to reflect this change by applying pipeline restriction in (3.11) instead of broadcast restriction in (3.10). The modified scheduling vector is:

$$\mathbf{s_7} = \begin{bmatrix} 1 & 1 & M \end{bmatrix} \tag{4.24}$$

The total number of time steps is $(K + MN - 1)$ as compared to $(MN)$ steps for Design #1. The resulting processor array architecture is shown in Figure 4.12. A total of

$K(K-1)/2$ delay registers are required to feed features of input dataset $\mathbf{Y}$. The structure of PE is the same as of Design #1 shown in Figure 4.8 with one more register for the pipelined input $\mathbf{X}$.

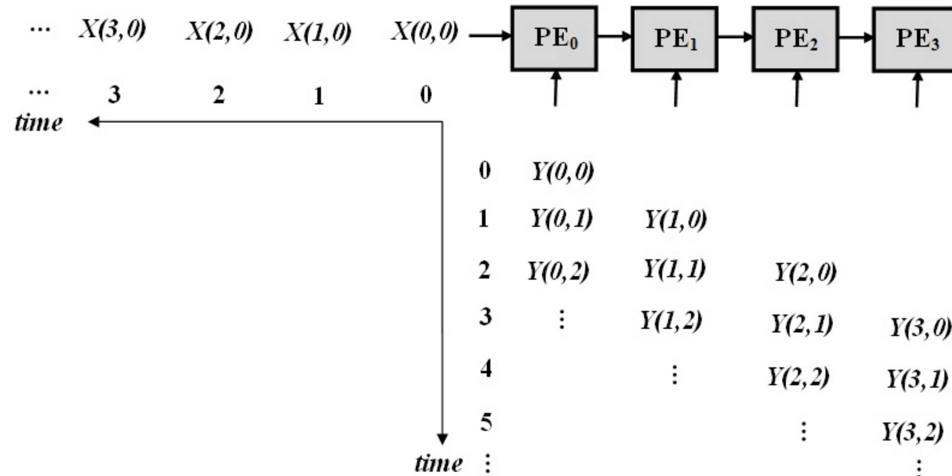



Figure 4.12: Inputs timing for Design of [4] with $K$=4

The main drawback of Design #3 and Design #4 is their slow clock rate compared to other architectures since $M$ partial results have to be added within a single clock cycle.

In [58], a distance calculation unit has been proposed to calculate similarity distances between data samples and cluster centroids in a hardware implementation of the K-Means clustering algorithm. The proposed design calculates $K$ distances between a data sample of $M$ features and the $K$ cluster centroids concurrently using $K$ adder trees of $M-1$ adders each. A total of $KM$ registers are required to store the cluster centroinds. Since large-scale datasets are usually stored in off-chip or on-chip RAMs with limited bandwidth and number of data ports, this design is not practical for high-dimensional data with large number of features $M$ as it requires feeding large number of features simultaneously. The results reported for this design are for a small dataset with number of features $M=4$. Clock speed of this design can be enhanced by pipelining the adder trees with each level represents a pipeline stage and a total of extra $K(M-1)$ pipeline registers, as proposed in [2].

Design #5 is another option that is similar to Design #1. The main differences between the two architectures are in the choice of broadcasting or localizing input variables $\mathbf{X}$ and $\mathbf{Y}$, and the number of PEs. Design #5 is not amenable for hardware implementation when

the value of $N$ is very large since it results in a huge number of PEs. However, this design option is suitable for processing high dimensional, low sample size (HDLSS) datasets [53]. One example of these datasets is the gene expression microarray datasets. These datasets typically have a small number of samples $N$ and a large number of genes that represent the features [54]. The time complexity for Design #6 is the same as of Design #5 with an extra $K \times N$ registers to store intermediate results.

Table 4.2 summarizes circuit and time complexities of the six proposed processor array architectures obtained in this chapter and previous architectures proposed in [4], [58], and [2]. Critical path delays are also presented with $T_a$ refers to the delay of a $w$-bit adder, where $w$ is the data width.

### 4.5.2 Implementation Results

As discussed in the previous subsection, Design #1 is the best design for calculating similarity distances in the K-Means clustering algorithm. Design #1 and previous designs proposed in [4], [58], and [2] are implemented on the same FPGA device to accelerate distance computation involved in clustering 4,096 samples of an image dataset (Bridge) [59], with each sample consists of 16 features. The four architectures are implemented in Verilog hardware description language with Xilinx ISE Design Suite 13.4 targeting Xilinx Virtex7 XC7VX330T. Table 4.3 and Figure 4.13 show implementation results for distance calculation involved in one iteration of the K-Means clustering algorithm with $N$=4,096 samples, $M$=16 features, and different number of clusters $K$.

Implementation results show that Design #1 outperforms Design of [4] in terms of area and speed for all values of $K$. Design of [4] occupies more slices due to the delay registers used to feed features of input dataset $\mathbf{Y}$. Execution time is determined by the number of clock cycles required to calculate all elements of distance matrix $\mathbf{D}$ and the clock rate. For all values of $K$, Design of [4] has a slower clock speed and requires $(K-1)$ more clock cycles than Design #1. As shown in Table 4.3, as the number of PEs increases, the maximum clock rate for Design #1 decreases due to the higher delay of longer broadcast buses. However, Design #1 still attains higher clock rate than Design of [4] due to higher clock skew as inspected by the Xilinx tool. The effect of clock skew and long broadcast buses can be minimized by using clock distribution networks and buffer insertion for Design of [4] and Design #1, respectively at the cost of more area and power consumption.

As expected based on time complexities of the four architectures in Table 4.2, Designs of [58] and [2] require less time to complete the computation of all elements of distance

Table 4.2: Design comparison.

| Design | Circuit Complexity | | | Number of Inputs | Time Complexity | Critical Path Delay |
|---|---|---|---|---|---|---|
| | Absolute Difference | Adder | Register | | | |
| **Design #1** | $K$ | $K$ | $K$ | $K+1$ | $MN$ | $3T_a$ |
| **Design #2** | $K$ | $K$ | $KN$ | $K+1$ | $MN$ | $3T_a$ |
| **Design #3** | $M$ | $M-1$ | – | $2M$ | $KN$ | $(2+\lceil \log_2 M \rceil)T_a$ |
| **Design #4** | $M$ | $M-1$ | – | $2M$ | $KN$ | $(2+\lceil \log_2 M \rceil)T_a$ |
| **Design #5** | $N$ | $N$ | $N$ | $N+1$ | $KM$ | $3T_a$ |
| **Design #6** | $N$ | $N$ | $KN$ | $N+1$ | $KM$ | $3T_a$ |
| **Design of [4]** | $K$ | $K$ | $\frac{1}{2}(K^2+K)$ | $K+1$ | $K+MN-1$ | $3T_a$ |
| **Design of [58]** | $KM$ | $K(M-1)$ | $KM$ | $M(K+1)$ | $N$ | $(2+\lceil \log_2 M \rceil)T_a$ |
| **Design of [2]** | $KM$ | $K(M-1)$ | $K(2M-1)$ | $M(K+1)$ | $\lceil \log_2 M \rceil + N$ | $2T_a$ |

matrix $\mathbf{D}$. However, Design #1, achieves an average decrease of 75% in area and 58% in area-delay product while being scalable to high-dimensional datasets of any dimension.

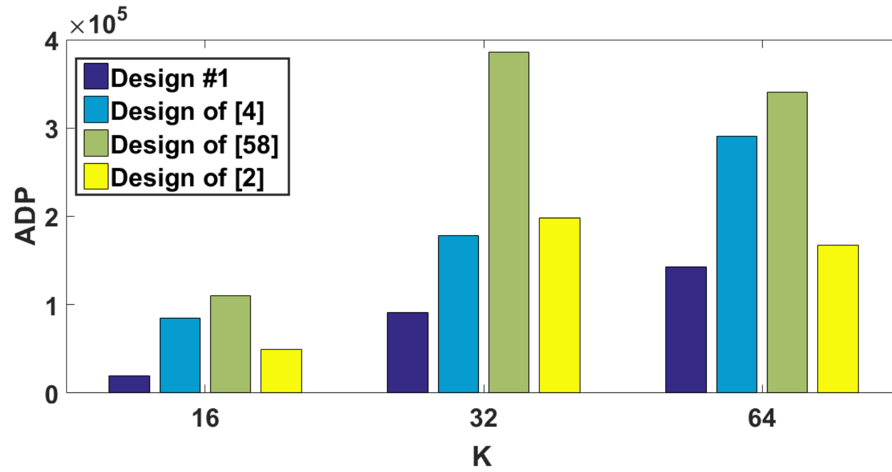



Figure 4.13: Area-delay product for different values of $\mathbf{K}$

Table 4.3: Implementation results.

| **Design** | **Area (#Slices)** | | | **Max. Frequency (MHz)** | | | **Delay ($\mu$s)** | | |
|---|---|---|---|---|---|---|---|---|---|
| | $K = 16$ | $K = 32$ | $K = 64$ | $K = 16$ | $K = 32$ | $K = 64$ | $K = 16$ | $K = 32$ | $K = 64$ |
| **Design #1** | 177 | 384 | 768 | 585.8 | 276.3 | 353.1 | 111.8 | 237.1 | 185.6 |
| **Design of [4]** | 295 | 643 | 1,020 | 228.7 | 237.0 | 230.5 | 286.6 | 276.6 | 284.5 |
| **Design of [58]** | 4,148 | 11,910 | 13,132 | 154.2 | 126.4 | 157.9 | 26.5 | 32.4 | 25.9 |
| **Design of [2]** | 3,548 | 12,564 | 13,087 | 294.8 | 259.4 | 320.7 | 13.9 | 15.8 | 12.7 |

# Chapter 5

# Scalable and Parameterizable Processor Array Architecture for Similarity Distance Computation

## 5.1 Introduction

Linear [4] and 2-D [38] [39] processor array architectures have been proposed to accelerate the computation of similarity distance matrices. 2-D processor arrays are generally faster than linear arrays as more processing elements (PEs) are used to perform the computation in parallel. On the other hand, linear arrays are more suitable for area, power, and bandwidth-constrained applications. To the best of our knowledge, data dimensionality and size have not been considered in previous 2-D processor arrays. For high-dimensional data, feeding all the features of a single data element simultaneously is not practical due to I/O bandwidth constraints. Linear processor arrays, on the other hand, are generally more area-efficient and meet I/O bandwidth constraints at the cost of slower execution times

Given the complexity of today's data, machine learning and data mining algorithms are expected to handle big and high-dimensional data. In [60], an optimized OpenCL high-level implementation of the K-Means clustering algorithm has been presented. The authors reported that the maximum number of features that could fit on Startix V A7 FPGA is around 160. Even partitioning the computation and caching partial results in local memory to accommodate lager sizes was not efficient due to excessive global memory transactions.

In this chapter, we present a systematic approach for developing scalable processor array architecture for similarity distance computation based on our recent work [61], that is

presented in Chapter 3, with more control on area and I/O requirements and better compromise between area and speed.

## 5.2 Methodology of processor array design

In our work presented in Chapters 3 and 4, we have systematically explored the design space of processor array architectures for similarity distance computation. Six 2-D and seven 1-D processor array architectures were obtained using linear scheduling and projection operations. In this work, nonlinear scheduling and projection are employed to develop a scalable and parameterized processor array architecture starting from the best architecture among these six architectures.

### 5.2.1 Data Scheduling

A scheduling function assigns each point in the computation domain a time value. Based on the choice of broadcasting or pipelining algorithm variables, we were able to obtain a number of linear scheduling functions in Chapters 3 and 4. Linear scheduling can be used to perform design space exploration for the problem in hand. Depending on the values of $K$, $M$, and $N$, in order to obtain scalable architectures that are amenable for hardware implementation, more control on the computational load at each time step is required. Figure 3.3 on page 38 shows one of the obtained timing options for 2-D processor arrays. In this timing option, all points on any given plane with the same value of coordinate $m$ are assigned the same time value and said to belong to a single equitomporal zone [1]. For large-scale data, having a large number of points executing at the same time results in an impractical hardware architecture that requires a huge number of PEs and the feeding of a large number of data inputs simultaneously.

Nonlinear scheduling can be used for more control on the computational load to be performed at each time step. In this chapter, our approach is to choose one of the linear scheduling alternatives obtained in Chapter 3 and develop a nonlinear scheduling function with smaller and parameterized equitemporal zones. Analyzing the four scheduling functions presented in Chapter 3, we choose to partition the equitemporal zones of the scheduling function shown in Figure 3.3 for the following reasons:

- The resulting processor array architecture has the best time complexity among the six obtained architectures.

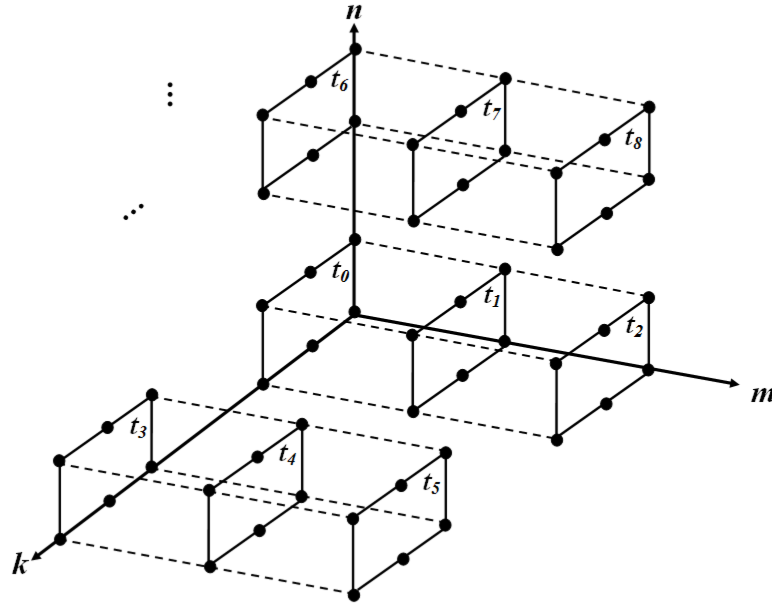



Figure 5.1: 2-D equitemporal zones using nonlinear scheduling function (5.1), K=6, M=3, $w_k$= 3, and $w_n$= 2.

- Simple feeding of data without any delay registers.
- No communication between PEs. Hence, easier partitioning without any feedback connections.
- Partial outputs are stored in local registers in the PEs. Hence, straightforward accumulation of partial results at each time step.

Planes in Figure 3.3 that represent equitemporal zones can be partitioned into smaller zones with parameterized dimensions. Rather than assigning all points on a plane the same time value, only points in the new smaller zones are assigned the same time value. The partitioning adopted in this work along with the order of execution are shown in Figure 5.1. Rather than calculating all $N \times K$ distances at each time step using linear scheduling, nonlinear scheduling allows for more control to calculate only $w_k \times w_n$ distances at each time step. The nonlinear scheduling function that assigns time values to points in the computation domain is given by:

$$t(\mathbf{p}) = \left\lfloor \frac{k}{w_k} \right\rfloor + H_k H_n m + H_n \left\lfloor \frac{n}{w_n} \right\rfloor \tag{5.1}$$

where:

$$H_k = \left\lfloor \frac{K-1}{w_k} \right\rfloor + 1 \tag{5.2}$$

$$H_n = \left\lfloor \frac{N-1}{w_n} \right\rfloor + 1 \tag{5.3}$$

### 5.2.2 Projection Operation

Linear projection is defined as the mapping of several points in the $n$-dimensional computation domain $\mathcal{D}$ to a single point in a $k$-dimensional domain $\tilde{\mathcal{D}}$, where $k \leq n$ [43]. It simply eliminates coordinates of axes along the projection direction vectors with no control on the size of the projected processor array. Previously obtained architectures in the literature are 1-D or 2-D processor arrays of size $K$, $M$, or $N$ in each dimension. For more control on the size of the resulting processor array and to achieve full utilization of hardware resources, we choose to assign each point in the equitemporal zones described in the previous subsection to a PE in the projected processor array. This ensures that all PEs are always busy with more control on the size of the resulting architecture by choosing parameters $w_k$ and $w_n$ to meet area and I/O bandwidth constraints. Each point $\mathbf{p} = \begin{bmatrix} k & m & n \end{bmatrix}^t \in \mathcal{D}$ will be mapped to processing element $PE_{i,j}$ where:

$$i = k \bmod w_k \tag{5.4}$$

$$j = n \bmod w_n \tag{5.5}$$

## 5.3 The proposed processor array architecture

The processor array architecture corresponding to the nonlinear scheduling and projection described in the previous section is shown in Figure 5.2. This architecture is obtained by partitioning the original $K \times N$ processor array we proposed in [61] using linear scheduling and projection operations. The proposed architecture is a parameterized processor array of $w_k \times w_n$ PEs. Both inputs $\mathbf{X}$ and $\mathbf{Y}$ are broadcast and output $\mathbf{D}$ is localized. As shown in Figure 5.3, partial results of output variable $\mathbf{D}$ calculated at each time step for a given value of $m$ are stored in local registers and hence, simple accumulation of outputs at each time step is possible.

In the case of K-Means clustering algorithm, for instance, a total of $w_k \times w_n$ distances between $w_n$ samples of input data $\mathbf{X}$ and $w_k$ cluster centroids are generated every $M$ clock cycles by the proposed architecture. The total number of cycles required to calculate all

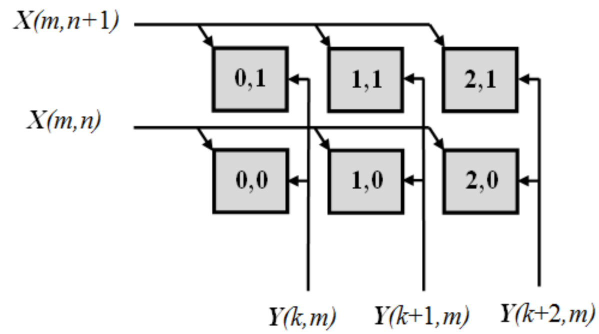



Figure 5.2: Proposed processor array architecture with $w_k$= 3, and $w_n$= 2.

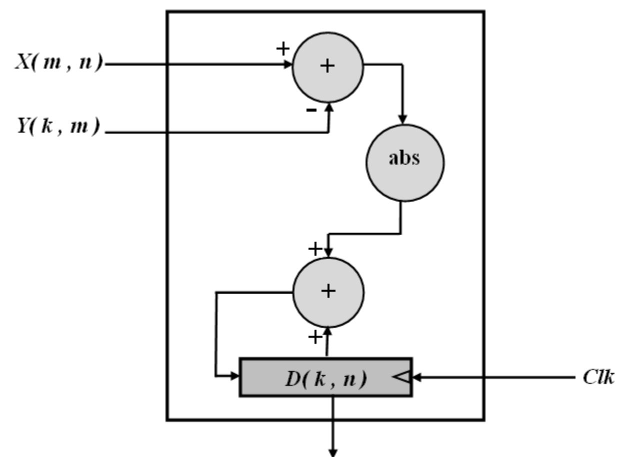



Figure 5.3: Processing element structure for the proposed architecture.

elements of distance matrix $\mathbf{D}$ is:

$$C_{total} = \lceil K/w_k \rceil \lceil N/w_n \rceil M \tag{5.6}$$

## 5.4 Design Comparison

The 2-D processor array proposed in [38] has $K \times N$ PEs. For datasets of thousands or even millions of samples $N$, the proposed architecture is not feasible for hardware implementation as it consists of a huge number of PEs with large number of data inputs being fed simultaneously. To overcome these limitations, the authors of [39] have proposed another 2-D processor array architecture of $K \times M$ PEs. However, the proposed architecture is not practical for high-dimensional datasets with large number of features $M$ per sample as it requires feeding all features of each sample simultaneously. In both architectures, data is fed in a skewed pattern and hence, a large number of delay registers are required for large and high-dimensional datasets. The linear processor array architecture proposed in [4] requires less area and I/O pins compared to 2-D architectures. However, the time complexity is much higher since lesser number of PEs are involved, with only one feature of each sample of dataset $\mathbf{X}$ being fed at a time. Table 5.1 summarizes circuit and time complexities of the proposed architecture and previous architectures in [38] and [39].

As discussed in Section 4.5, the non-systolic architectures of [58] and [2] are not amenable for hardware implementation when dealing with high-dimensional data. The proposed architecture, on the other hand, scales better for high-dimensional data since it deals only with one dimension of data at each time step. As shown in Table 5.1, the proposed architecture is fully parameterized. Parameters $w_k$ and $w_n$ can be chosen to determine the number of PEs so that area, speed, and I/O bandwidth constraints are met.

## 5.5 Implementation Results

The proposed processor array architecture is implemented in Verilog hardware description language using Xilinx ISE Design Suite 13.4 to target Xilinx Virtex7 XC7VX330T FPGA to accelerate distance computation involved in clustering data samples of the letter recognition dataset from UCI Repository [62]. The dataset consists of 20,000 samples for the 26 alphabets with each having 16 numerical attributes represented as integer values in the range from 0 to 15. Table 5.2 shows implementation results for distance calculation involved in one iteration of the K-Means clustering algorithm with $K$ = 26, $M$ =16,

Table 5.1: Design Comparison.

| **Design** | **Circuit Complexity** | | | **Number of Inputs** | **Time Complexity** |
|---|---|---|---|---|---|
| | **Absolute Difference** | **Adder** | **Register** | | |
| **[38]** | $KN$ | $KN$ | $\frac{1}{2}(K^2 + N^2 + 2KN - K - N)$ | $K + N$ | $K + M + N$ |
| **[39]** | $KM$ | $KM$ | $\frac{1}{2}(M^2 + 2KM - M)$ | $M$ | $K + M + N$ |
| **Proposed** | $w_k w_n$ | $w_k w_n$ | $w_k w_n$ | $w_k + w_n$ | $\lceil K/w_k \rceil \lceil N/w_n \rceil M$ |

Table 5.2: Implementation results of the proposed architecture with $K = 26$, $w_k$= 13 ($K/2$), and different values of $w_n$.

| $\mathbf{w_n}$ | **Area (# Slices)** | **Max. Frequency (MHz)** | **Delay ($\mu$ Sec)** | **ADP** |
|---|---|---|---|---|
| **2** | 277 | 703.7 | 454.7 | 125,962 |
| **4** | 576 | 341.7 | 468.2 | 269,710 |
| **8** | 1,156 | 354.4 | 225.7 | 260,948 |
| **16** | 2,514 | 285.0 | 140.3 | 352,842 |

$N$=20,000, $w_k$=13, and different values of $w_n$. As shown in this table, the parameterized architecture allow for more control on area and speed with $w_n$=2 gives the best Area-Delay product.

Table 5.3 and Figure 5.4 show implementation results for previous architectures in [4], [39], [58], [2], and the proposed architecture (with $w_k = K/2$ and $w_n$=2) for the same dataset and different number of clusters $K$. Delays are calculated using the maximum frequencies achieved after implementing each architecture and the required clock cycles shown in the last column of Table 5.1.

In addition of its scalability to any number of features $M$, implementation results show that the proposed architecture achieves the best compromise between area and speed. Architecture of [58] has the worst Area-Delay product due to its slow clock rate even compared to similar architecture of [2] with pipelined adder trees. The proposed architecture achieves an ADP that is comparable and even slightly better than that of the 2-D architecture in [39] using an average of only 9% of its area. An average decrease of 82% in ADP compared to the linear architecture of [4] is also achieved with 3x average speedup.

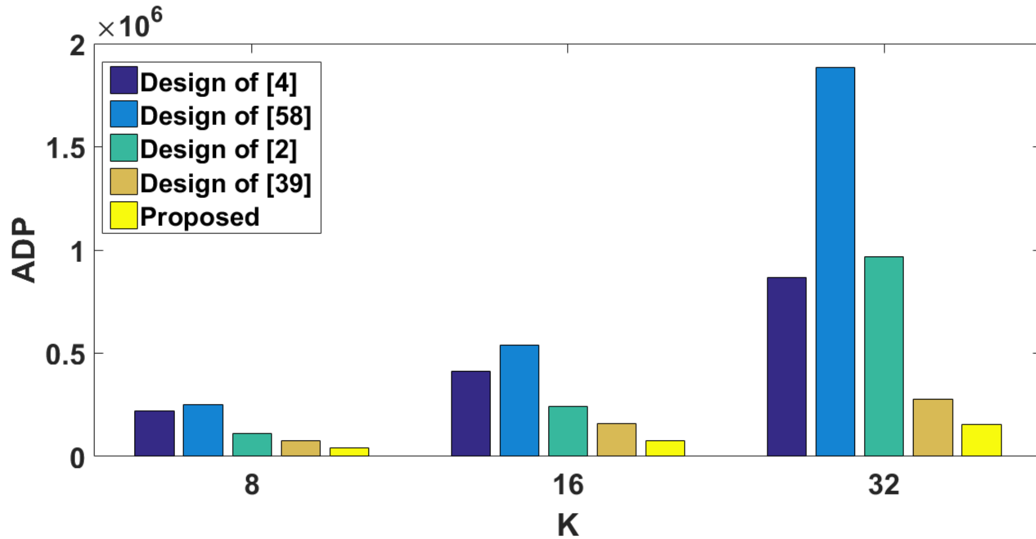



Figure 5.4: Area-Delay Product.

Table 5.3: Implementation results of previous architectures and the proposed architecture ($w_n = 2$, $w_k = K/2$) for different values of $K$.

| **Design** | **Area (#Slices)** | | | **Max. Frequency (MHz)** | | | **Delay ($\mu$s)** | | |
|---|---|---|---|---|---|---|---|---|---|
| | $K = 8$ | $K = 16$ | $K = 32$ | $K = 8$ | $K = 16$ | $K = 32$ | $K = 8$ | $K = 16$ | $K = 32$ |
| **[4]** | 166 | 295 | 643 | 238.8 | 228.7 | 237.0 | 1,340 | 1,399 | 1,350 |
| **[58]** | 1,977 | 4,148 | 11,910 | 156.7 | 154.2 | 126.4 | 127 | 129 | 158 |
| **[2]** | 1,727 | 3,548 | 12,564 | 316.2 | 294.8 | 259.4 | 63 | 67 | 77 |
| **[39]** | 944 | 1,992 | 3,614 | 252.7 | 252.0 | 262.3 | 79 | 79 | 76 |
| **Proposed** ($w_n$=2) | 90 | 169 | 336 | 710.2 | 710.2 | 698.3 | 450 | 450 | 458 |

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

This section summarizes the research work presented in the dissertation.

In Chapter 2, we proposed a fast and area-efficient FPGA implementation of the K-means algorithm for clustering one-dimensional data. In the proposed implementation, centroids update equations are rewritten to calculate the new centroids recursively. The division operation appears in these equations is replaced with a shift operation. Experimental results show that this approximation results in a more area-efficient hardware implementation while maintaining the quality of clustering results. Experimental results also show that the continuous update of clusters centroids adopted in the proposed design results in faster convergence of the algorithm using less number of iterations compared to the general update approach used in the conventional design.

A systematic algebraic methodology is presented in Chapter 3 to design 2-D processor array architectures for similarity distance computation. Previously obtained architectures have been obtained using ad hoc techniques that do not allow for design space exploration. To achieve practical designs that are amenable for hardware implementation, the proposed designs are classified based on the size and dimensionality of the input datasets. The methodology presented in this chapter is used to obtain the 3-D computation domain of the similarity distance computation algorithm. Four linear scheduling functions are presented, and six possible 2-D processor array architectures are obtained and analyzed in terms of speed and area. Compared to previously obtained designs, the proposed designs achieve better time and area complexities.

The methodology presented in Chapter 3 is employed in Chapter 4 to explore the design

space of 1-D processor array architectures for the computation of similarity distance matrices. To meet area and bandwidth constrains, more time restrictions are introduced on input and output variables. Six new processor arrays, in addition to a previously devised one, are obtained systematically. Time and area complexities of these seven architectures are compared and analyzed. Implementation results for the previously obtained architecture and one of our proposed architectures show that the proposed architecture achieves better performance in terms of speed and area.

In Chapter 5, nonlinear scheduling and projection operations are introduced to systematically develop scalable processor array architecture for similarity distance computation is presented. The fully parameterized architecture proposed achieves better compromise between area and speed with more control on the number of PEs and I/O pins. Moreover, the methodology introduced makes scalability possible in the designed architecture.

## 6.2 Future Work

The research work presented in this dissertation can be extended in the future along the following research directions:

1. The approach of continuously updating cluster centroids presented in Chapter 2 results in faster convergence of the K-Means algorithm for clustering one-dimensional data. This approach can be investigated for clustering multidimensional data with techniques to minimize the complexity of hardware structures required to implement it.

2. More accurate approximations can be investigated to improve the accuracy of the shift-based division approximation in the modified centroids update equations, considering floating point data.

3. The systematic methodology employed to develop processor array architectures in Chapter 3 and Chapter 4 can be used to build automated tools to generate high-level templates, such as OpenCL code, with full description of the explored processor arrays.

4. Scheduling and projection operations introduced in this work allow for control on time and area complexities of the proposed architectures. Hence, the emphasis was to enhance the performance of hardware accelerators in terms of speed and area.

We intend to analyze the proposed architectures in terms of power, and extend the proposed methodologies to design power-efficient architectures that are critical for embedded data mining applications.

5. The significance of employing systematic methodologies to design processor array architectures is more obvious when dealing with high-dimensional algorithms. We intend to employ the same approach to explore the design space of processor array architectures for deep convolutional neural networks. The computations involved in a convolutional layer is implemented as six nested loops, which means a computation domain of dimension 6.

6. The delay and fan-out of broadcast buses utilized in the proposed architectures can be enhanced using buffer insertion techniques, and a model for the fan-out of these buses can be derived.

7. Another interesting future research direction is to investigate the extension of the employed methodology to support analyzing data dependencies in multi-dimensional algorithms to generate optimized multi-threaded software implementations.

# Appendix A

# Publications

The works presented in Chapters 2 through 5 have been published/submitted in the following research articles, respectively:

1. Awos Kanan, Fayez Gebali, and Atef Ibrahim, **”Fast and Area-Efficient Hardware Implementation of the K-means Clustering Algorithm”**, *WSEAS Transactions on Circuits and Systems*, Vol. 15, pp 133-142, 2016.

2. Awos Kanan, Fayez Gebali and Atef Ibrahim, **”Design Space Exploration of 2-D Processor Array Architectures for Similarity Distance Computation”**, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 28, No. 8, pp 2218-2228, Aug 2017.

3. Awos Kanan, Fayez Gebali, Atef Ibrahim, and Kin Fun Li, **”Design Space Exploration of Resource-Constrained Architectures for Similarity Distance Computation”**, submitted, *Journal of Parallel and Distributed Computing*, Elsevier, Jan 2018.

4. Awos Kanan, Fayez Gebali, Atef Ibrahim, and Kin Fun Li, **”Scalable and Parameterizable Processor Array Architecture for Similarity Distance Computation”**, to be submitted, *IET Computers and Digital Techniques*, 2018.

# Bibliography

[1] F. Gebali, *Algorithms and Parallel Computing*. John Wiley & Sons, 2011.

[2] H. M. Hussain, K. Benkrid, A. Ebrahim, A. T. Erdogan, and H. Seker, "Novel dynamic partial reconfiguration implementation of k-means clustering on FPGAs: comparative results with GPPs and GPUs," *International Journal of Reconfigurable Computing*, vol. 2012, pp. 1–15, 2012.

[3] G. Karch, "GPU-based acceleration of selected clustering techniques," *Master's thesis, Silesian*, 2010.

[4] M. F. Hsieh and C. H. Lai, "A serial input VLSI systolic architecture for a clustering analyser," *International journal of electronics*, vol. 84, no. 3, pp. 269–284, 1998.

[5] A. Choudhary, R. Narayanan, B. Ö. Ikyılmaz, G. Memik, J. Zambreno, and J. Pisharath, "Optimizing data mining workloads using hardware accelerators," in *Proc. of the Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, 2007.

[6] D. Xu and Y. Tian, "A comprehensive survey of clustering algorithms," *Annals of Data Science*, vol. 2, no. 2, pp. 165–193, 2015.

[7] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[8] A. K. Jain and R. C. Dubes, *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[9] S. Kotsiantis, "Supervised machine learning: a review of classification techniques," *Informatica*, vol. 31, no. 3, pp. 249–269, 2007.

[10] R. Gentleman and V. Carey, "Unsupervised machine learning," in *Bioconductor Case Studies*, 2008, pp. 137–157.

[11] A. K. Nandi, R. Fa, and B. Abu-Jamous, *Integrative Cluster Analysis in Bioinformatics*. John Wiley & Sons, 2015.

[12] A. Y. Zomaya, *Parallel computing for bioinformatics and computational biology: models, enabling technologies, and case studies*. John Wiley & Sons, 2006, vol. 55.

[13] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[14] M. J. Flynn and K. W. Rudd, "Parallel architectures," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 67–70, 1996.

[15] B. K. Khailany, T. Williams, J. Lin, E. P. Long, M. Rygh, D. W. Tovey, and W. J. Dally, "A programmable 512 GOPS stream processor for signal, image, and video processing," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 202–213, 2008.

[16] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *ISMM*, 2007, pp. 103–104.

[17] D. Singh and C. K. Reddy, "A survey on platforms for big data analytics," *Journal of Big Data*, vol. 2, no. 1, p. 1, 2014.

[18] V. Faber, "Clustering and the continuous k-means algorithm," *Los Alamos Science*, vol. 22, pp. 138–144, 1994.

[19] E. Aksehirli, B. Goethals, and E. Müller, "Efficient cluster detection by ordered neighborhoods," in *Big Data Analytics and Knowledge Discovery*, 2015, pp. 15–27.

[20] T. Yun, T. Hwang, K. Cha, and G.-S. Yi, "CLIC: clustering analysis of large microarray datasets with individual dimension-based clustering." *Nucleic Acids Research*, vol. 38, no. Web-Server-Issue, pp. 246–253, 2010.

[21] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, 1967, pp. 281–297.

[22] R. Xu, D. Wunsch *et al.*, "Survey of clustering algorithms," *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645–678, 2005.

[23] Y. Choi and H. K. So, "Map-reduce processing of k-means algorithm with FPGA-accelerated computer cluster," in *IEEE 25th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2014, pp. 9–16.

[24] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski, "Algorithmic transformations in the implementation of k-means clustering on reconfigurable hardware," in *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, 2001, pp. 103–110.

[25] D. Lavenier, "FPGA implementation of the k-means clustering algorithm for hyperspectral images," in *Los Alamos National Laboratory LAUR 00-3079*, 2000.

[26] X. Wang and M. Leeser, "K-means clustering for multispectral images using floating-point divide," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines FCCM*, April 2007, pp. 151–162.

[27] X. Wang and B. Nelson, "Tradeoffs of designing floating-point division and square root on virtex FPGAs," in *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2003*, April 2003, pp. 195–203.

[28] 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 *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, June 2011, pp. 248–255.

[29] K. J. Kohlhoff, V. S. Pande, and R. B. Altman, "K-means for parallel architectures using all-prefix-sum sorting and updating steps," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 8, pp. 1602–1612, 2013.

[30] T.-M. Huang, V. Kecman, and I. Kopriva, *Kernel based algorithms for mining huge data sets*. Springer, 2006.

[31] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM computing surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[32] Z. K. Baker and V. K. Prasanna, "Efficient hardware data mining with the apriori algorithm on FPGAs," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, 2005, pp. 3–12.

[33] R. D. Chamberlain, R. K. Cytron, M. A. Franklin, R. S. Indeck *et al.*, "The mercury system: Exploiting truly fast hardware for data search," in *Proc. of Intl Workshop on Storage Network Architecture and Parallel I/Os*, 2003, pp. 65–72.

[34] S. Fushimi and M. Kitsuregawa, "GREO: a commercial database processor based on a pipelined hardware sorter," in *ACM SIGMOD Record*, vol. 22, no. 2, 1993, pp. 449–452.

[35] U. Inoue, T. Satoh, H. Hayami, H. Takeda, T. Nakamura, and H. Fukuoka, "RINDA: a relational database processor with hardware specialized for searching and sorting," *IEEE Micro*, vol. 11, no. 6, pp. 61–70, 1991.

[36] B. West, R. D. Chamberlain, R. S. Indeck, and Q. Zhang, "An FPGA-based search engine for unstructured database," in *Proc. of 2nd Workshop on Application Specific Processors*, 2003, pp. 25–32.

[37] D. G. Perera and K. F. Li, "Parallel computation of similarity measures using an FPGA-based processor array," in *22nd International Conference on Advanced Information Networking and Applications, 2008. AINA 2008*, 2008, pp. 955–962.

[38] H. Cheng and C. Tong, "Clustering analyzer," *IEEE Transactions on Circuits and Systems*, vol. 38, no. 1, pp. 124–128, 1991.

[39] M. F. Lai, M. Nakano, Y. P. Wu, and C. H. Hsieh, "VLSI design of clustering analyser using systolic arrays," *IEE Proceedings - Computers and Digital Techniques*, vol. 142, no. 3, pp. 185–192, 1995.

[40] S. K. Rao and T. Kailath, "Regular iterative algorithms and their implementation on processor arrays," *Proceedings of the IEEE*, vol. 76, no. 3, pp. 259–269, 1988.

[41] S. Kung, *VLSI Array Processors*. NJ: Prentice Hall, 1988.

[42] E. Abdel-Raheem, "Design and VLSI implementation of multirate filter banks," Ph.D. dissertation, University of Victoria, 1995.

[43] F. El-Guibaly and A. Tawfik, "Mapping 3-d IIR digital filter onto systolic arrays," *Multidimensional Systems and Signal Processing*, vol. 7, no. 1, pp. 7–26, 1996.

[44] F. Gebali and A. Rafiq, "Processor array architectures for deep packet classification," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 3, pp. 241–252, 2006.

[45] M. Rehan, M. W. El-Kharashi, and F. Gebali, "A hierarchical design methodology for full-search block matching motion estimation," *Multidimensional Systems and Signal Processing*, vol. 17, no. 4, pp. 327–341, 2006.

[46] A. Ibrahim, F. Gebali, H. El-Simary, and A. Nassar, "High-performance, low-power architecture for scalable radix 2 montgomery modular multiplication algorithm," *Canadian Journal of Electrical and Computer Engineering*, vol. 34, no. 4, pp. 152–157, 2009.

[47] A. Ibrahim, F. Gebali, H. Elsimary, and A. Nassar, "Processor array architectures for scalable radix 4 montgomery modular multiplication algorithm," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 7, pp. 1142–1149, 2011.

[48] A. Ibrahim, H. Elsimary, and F. Gebali, "Low-power, high-speed unified and scalable word-based radix 8 architecture for montgomery modular multiplication in GF (P) and GF ($2^n$)," *Arabian Journal for Science and Engineering*, vol. 39, no. 11, pp. 7847–7863, 2014.

[49] A. Ibrahim, F. Gebali, and T. F. Al-Somani, "Systolic array architectures for sunar–koç optimal normal basis type II multiplier," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 10, pp. 2090–2102, 2015.

[50] F. Gebali and A. Ibrahim, "Efficient scalable serial multiplier over GF($2^m$) based on trinomial," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 10, pp. 2322–2326, 2015.

[51] A. Ibrahim and F. Gebali, "Low power semi-systolic architectures for polynomial-basis multiplication overGF ($2^m$) using progressive multiplier reduction," *Journal of Signal Processing Systems*, vol. 82, no. 3, pp. 331–343, 2016.

[52] F. Gebali and A. Ibrahim, "Low space-complexity and low power semi-systolic multiplier architectures over GF($2^m$) based on irreducible trinomial," *Microprocessors and Microsystems*, vol. 40, pp. 45–52, 2016.

[53] Y. Terada, "Clustering for high-dimension, low-sample size data using distance vectors," *arXiv preprint arXiv:1312.3386v2*, 2013. [Online]. Available: http://arxiv.org/abs/1312.3386

[54] A. Hochstein, H. I. Ahn, Y. T. Leung, and M. Denesuk, "Survival analysis for HDLSS data with time dependent variables: Lessons from predictive maintenance at a mining service provider," in *IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*, July 2013, pp. 372–381.

[55] T. R. Golub, D. K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J. P. Mesirov, H. Coller, M. L. Loh, J. R. Downing, M. A. Caligiuri *et al.*, "Molecular classification of cancer: class discovery and class prediction by gene expression monitoring," *science*, vol. 286, no. 5439, pp. 531–537, 1999.

[56] D. Jiang, C. Tang, and A. Zhang, "Cluster analysis for gene expression data: a survey," *IEEE Transactions on knowledge and data engineering*, vol. 16, no. 11, pp. 1370–1386, 2004.

[57] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley, 1993.

[58] R. Ratnakumar and S. J. Nanda, "A FSM based approach for efficient implementation of K-Means algorithm," in *20th International Symposium on VLSI Design and Test (VDAT)*, May 2016, pp. 1–6.

[59] P. Franti, "Clustering datasets," 2015. [Online]. Available: http://cs.uef.fi/sipu/datasets/

[60] Q. Y. Tang and M. A. S. Khalid, "Acceleration of K-means algorithm using Altera SDK for OpenCL," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 10, no. 1, pp. 6:1–6:19, 2016.

[61] A. Kanan, F. Gebali, and A. Ibrahim, "Design space exploration of 2-D processor array architectures for similarity distance computation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2218–2228, Aug 2017.

[62] C. Blake and C. Merz, "UCI repository of machine learning databases." *[Online]. Available: http://www. ics. uci. edu/˜ mlearn/ML-Repository. html*, 1998.