Design Exploration of Hardware Accelerators For The K-NN Algorithm

by

Dunia Jamma

A thesis
presented to
the University of Guelph

In partial fulfillment of requirements
for the degree of
Master of Applied Science
in
Engineering

Guelph, Ontario, Canada

ABSTRACT

Design Exploration Of Hardware Accelerators For The K-NN Algorithm

Dunia Jamma
University of Guelph, 2016

Advisor:
Professor Shawki Areibi, Professor Gary Grewal

Increasingly, machine-learning algorithms are playing an important role in the context of embedded and real-time systems. Applications such as wireless sensor networks, security, and commercial enterprises rely increasingly on machine-learning algorithms to efficiently make predictive decisions based on the large volumes of data these systems collect. Most supervised machine-learning algorithms, however, require relatively large amounts of runtime to perform training and/or classification due to the size and dimensionality of the data they must work with. Therefore, there is a need to accelerate the runtime of these algorithms, especially for real-time applications. In this thesis, several different hardware accelerators are proposed and compared for the K-Nearest Neighbor (K-NN) classification algorithm. These accelerators are developed using Xilinx Vivado High-Level Synthesis (HLS) and Cadence Tensilica tools, and represent different (tightly coupled versus semi-tightly coupled) architectures. The experimental results, based on several benchmarks, show that hardware speedups for an HLS range from 48x-168x, while those obtained using Cadence Tensilica tools range from 86x-650x.

Acknowledgements

In the end we will conserve only what we love; we will love only what we understand; and we will understand only what we have been taught. - Baba Dioum

First and foremost, I would like to thank my greatest teacher of all: God. I know that I am here and that I am able to write all of this for a reason. I will do my best to never forget how fortunate I am to be here, and that this comes with lessons and a responsibilities. I hope I am doing the work You have planned me to do.

I would like to express my deep gratitude and special appreciation to my advisor Dr. Shawki Areibi for his great support and encouragement during my studies. Thank you for all the guidance and feedback you have provided me. Thank you for your open door policy and the patience you have shown whenever I needed your help. I appreciate you putting pressure on me and pushing me toward what I have become through learning, and working on this research.

I would also like to express my deep gratitude to my co-advisor Dr. Gary Grewal for all the valuable advice, feedback and support he provided. Without these, this thesis would hardly be completed. Thank you for the patience you showed all the time while I was carrying out this research.

I would like to express a special thanks to my colleague and friend Omar Ahmed. One of the best things that has happened to me was working with you. Thank you for all the moral and academic support you provided.

I owe a very special thank to my parents, you sacrificed a great deal, suffered and worked hard all your life to permit me to be in a such position. I hope I am making you proud.

Many thanks to my brothers and sisters, especially to my sister-in-law Rania Hanna. Without your daily support, I am sure that I would not have been able to reach to the point I am at. To my daughters Mirel and Melinda - thank you for forgiving me for being away most of the time during my research.

The best is left to the last. Words cannot express how grateful I am to my husband Maher Azar. You have always been a close friend who provides me with support and encouragement even if it puts more pressure and hard work on yourself. Your support for me was felt when dealing with every single problem I faced during my study.

Contents

1 Introduction **1**

1.1 Problem Definition 1

1.2 Challenges and Motivation 2

1.3 Thesis Statement 3

1.4 Contribution 3

1.5 Thesis Summary 4

2 Background **5**

2.1 Preprocessing Methods 5

2.2 Supervised Learning Algorithms 7

2.2.1 The K-NN Algorithm 8

2.3 Field Programmable Gate Arrays (FPGAs) 11

2.4 Tools 12

2.4.1 Xilinx Vivado HLS Tool 12

2.4.2 Xtensa Tensilica Tool 22

2.5 Summary 25

3 Literature Review **26**

3.1 Accelerating Supervised Learning Algorithms 26

3.2 K-Nearest Neighbors Accelerators 27

3.3 Hardware Platforms and Implementations 30

3.3.1 FPGA versus GPUs 30

3.3.2 Interconnect Communication Overhead 31

3.4 Summary 33

4 Overall Design Exploration Methodology **35**

4.1 Data Sets 35

4.2 Preprocessing 36

4.2.1 Data Type Conversion 36

4.2.2 Replacement of Missing Values 37

4.2.3 Scaling 37

4.3 K-NN Algorithm 37

4.3.1 The *K* Hyper-Parameter 39

4.3.2 Profiling 39

4.4 Thesis Framework 40

5 Methodology: Design Exploration **42**

5.1 Tightly Coupled Architectures 43

5.1.1 Overall ASIP Approach 43

5.1.2 The “Distance Calculation” Module 45

5.1.3 The “K-Closest Neighbors” Module 47

5.1.4 The “Most Frequent Class” Module 48

5.1.5 Alternative Bus Architectures 50
5.2 Semi-Tightly Coupled Architecture 50
5.2.1 Design Exploration of the K-NN 52
5.2.2 K-NN Custom Architectures 55
5.2.3 K-NN General Architecture 56
5.2.4 Communication Interconnect 56
5.3 Summary 62

6 Results and Analysis **63**
6.1 Experimental Setup 63
6.2 Results of Semi-Tightly Coupled Architectures 64
6.2.1 Design Exploration of Custom Architectures 64
6.2.2 Custom Architectures: Implementation 70
6.2.3 Semi-Tightly Coupled General Architecture 73
6.3 Tightly Coupled Architectures 77
6.4 Summary 79

7 Conclusions and Future Work **80**
7.1 Summary 80
7.2 Insight 81
7.3 Future Work 82

Bibliography **84**

A Glossary **93**

B Matrix Multiplication Experiment **95**

B.1 Introduction 95

B.2 Data Sets (Benchmark) 96

B.3 Tools and Platforms 96

B.4 Experimental Setup 97

B.4.1 Matrix Multiplication Architectures 97

B.4.2 Types of Interconnects 97

B.5 Matrix Multiplication results 98

B.6 Analysis and Findings 99

B.7 Summary 104

List of Tables

4.1 Benchmarks: Records, Features and Classes 36

4.2 Results of Tensilica Profiling 39

6.1 HLS: Design Exploration for BCW_9 Benchmark 64

6.2 HLS: Design Exploration for BCW_30 Benchmark 65

6.3 HLS: Design Exploration for Spectf Benchmark 66

6.4 HLS: Design Exploration for Spambase Benchmark 66

6.5 HLS: Design Exploration for Mfeat Benchmark 67

6.6 HLS: Design Exploration for RobotF Benchmark 67

6.7 Resource and Power Consumption by HLS IPs 72

6.8 Clock Cycles/Speedup for KNN HLS IPs 72

6.9 Execution time(ms)/Speedup for HLS Integrated ARM-IPs 72

6.10 HLS: Execution time (ms) / speedups using AXI_BRAMs interconnects 73

6.11 HLS:Execution time (ms) / speedups using AXI_Light interconnects . . 74

6.12 HLS: Execution time (ms) / speedups using stream interconnect with HP port 74

6.13 HLS: Execution time (ms) / speedups using stream interconnect with ACP port 75

6.14 Resource and Power Consumption by HLS IPs 75
6.15 The cores specification of the ASIP implementation 77
6.16 Clock Cycles for Various ASIP Implementation 78
6.17 Total Time/Overall Speedup of ASIP Implementations 78
6.18 Execution time in milliseconds in ARM, ASIP and ARM/IP 79

B.1 Clock cycles and speedup for 8x8 matrices 98
B.2 Clock cycles and speedup for 16x16 matrices 99
B.3 Clock cycles and speedup for 32x32 matrices 99
B.4 Clock cycles and speedup for 50x50 matrices 100
B.5 Estimated clock cycles, speedup and resources of the HLS over the baseline implementation for 8x8 matrices 100
B.6 Estimated clock cycles, speedup and resources of the HLS over the baseline implementation for 16x16 matrices 101
B.7 Estimated clock cycles, speedup and resources of the HLS over the baseline implementation for 32x32 matrices 101
B.8 Estimated clock cycles, speedup and resources of the HLS over the baseline implementation for 50x50 matrices 102

List of Figures

2.1 Organization of the records and features in the data sets 8

2.2 An example of K-NN classifier method for K=3, and K=6 10

2.3 Flowchart of processing steps in Vivado HLS tools 13

2.4 A simple block diagram of implementation of HLS design in FPGA . . 14

2.5 Implementation of Inline directive in Vivado HLS 17

2.6 Implementation of Loop Unroll directive in Vivado HLS 18

2.7 The timing diagram of instructions with and without unrolling directive 18

2.8 The timing diagram instructions with and without pipeline directive . . 19

2.9 The timing diagram of the pipeline 20

2.10 The behaviour of the FIFO and PingPong data styles in Dataflow 21

2.11 The partitioning of the array: complete, block, and cyclic 23

2.12 A simple block diagram of Tensilica design flow 24

4.1 Thesis framework 41

5.1 Distance Calculation Module 45

5.2 K-Closest Neighbors Module 48

5.3 Most Frequent Class Module 49

5.4 K-NN Design in HLS 57
5.5 AXI Light Interconnect 59
5.6 AXI BRAM Interconnect 60
5.7 Stream HP Interconnect 60
5.8 Stream ACP Interconnect 61

6.1 The accelerated architecture of the KNN Algorithm by BCDE directives 71

Chapter 1

Introduction

1.1 Problem Definition

Machine-learning is a method of data analysis that automates analytical model building. Machine-learning algorithms can determine how to perform important tasks by learning from historic data and generalizing from examples. This is often feasible and cost effective where manual programming is not. As more data becomes available, more ambitious problems can be tackled. As a result, machine-learning is widely used in engineering, computer science, and other fields. Machine-learning algorithms can be organized into several classes based on the learning style and the desired outcome of the algorithm. Examples of machine-learning algorithms include (i) supervised learning, (ii) unsupervised learning, (iii) semi-supervised learning, and (iv) reinforcement learning.

The work in this thesis is concerned with supervised learning, which is the task of inferring a function from labeled training data. The training data consists of training examples. In supervised learning, each example is a pair consisting of an input object,

from which relevant features are extracted, and a desired output value (label or class). A supervised-learning algorithm analyzes the training data and produces an inferred function, which can then be used for classifying new examples [1].

1.2 Challenges and Motivation

Most supervised-learning algorithms consume relatively large amounts of time to first train the classification model and then to use the model in classifying new data. The total time required for both training and classification is directly related to the size and dimensionality of the data (number of features). The amount of data in most practical applications is very large. Therefore, there is a need to accelerate classification algorithms, especially for embedded, real-time applications.

Many different architectures have been used for acceleration purposes such as Application-Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). In general, ASICs are sufficiently fast. However, they are expensive and lack flexibility. A GPU can achieve substantial speedup with its fixed design of relatively simple parallel processing elements. However, the main drawback of the GPU is its high power consumption. FPGAs, on the other hand, are both flexible and reconfigurable. They can often achieve excellent speedups over pure software implementations because the design can be customized by the designer, unlike with a GPU, to exploit the unique form of parallelism present in the application.

Acceleration of the classification algorithms is crucial for real-time applications, more specifically for embedded applications. FPGAs are a perfect platform for accelerating embedded applications due to their low power consumption. However, there are

many factors that affect the amount of acceleration that can be achieved, such as the type of communication links used.

1.3 Thesis Statement

The objective of this thesis can be summarized based on the following goals: First, the focus is on the K-Nearest Neighbor (K-NN) classification algorithm. This involves proposing several different hardware accelerators, which have been developed for it using Xilinx Vivado High-Level Synthesis (HLS) [2] and Cadence ASIP Tensilica tools [3]. Second, a study regarding the effects of using various types of communication links on the overall performance of the hardware accelerators is developed.

1.4 Contribution

The main contributions of this thesis are as follows:

1. The development and evaluation of several hardware accelerators for K-NN using two different approaches – one based on Xilinx Vivado HLS and the other based on Cadence Tensilica. The latter approach results in a tightly-coupled Application Specific Instruction Set (ASIP) architecture, while the former approach results in a semi-tightly coupled co-processor architecture.

2. An exhaustive design exploration of co-processors using different Xilinx Vivado HLS optimization directives with the goal of identifying/understanding which combination of optimizations results in a high-performance architecture.

3. A design exploration of ASIPs using the Tensilica Instruction Extension (TIE) language [4]. The latter is used to customize the Xtensa architecture by adding custom instructions and register files. These tightly coupled architectures are contrasted with the co-processors produced by Xilinx Vivado HLS.

1.5 Thesis Summary

The remainder of this thesis is organized as follows: Chapter 2 includes essential background on supervised machine-learning algorithms, and the most common preprocessing techniques. Moreover, a brief background is presented on hardware platforms and tools such as the Zynq platform and Xilinx Vivado HLS tools. Related research work is presented in Chapter 3. In Chapter 4, the overall methodology for the work in this thesis is presented. Different architectures for the K-NN algorithm using Tensilica tools, and design exploration based on HLS using the K-NN algorithm with a variety of interconnects study will be the main topic of Chapter 5. Chapter 6 presents the results achieved by the work presented in Chapters 4 and 5, as well as the findings and analysis accomplished by this work. Chapter 7 concludes this thesis and suggests the possible work for the future.

Chapter 2

Background

Machine-learning algorithms can be classified and organized into a taxonomy based on the desired outcome of the algorithm. Examples of machine-learning algorithms include: (i) supervised learning, (ii) unsupervised learning, (iii) semi-supervised learning, and (iv) reinforcement learning.

In this chapter, a brief introduction to supervised machine-learning classifiers and preprocessing algorithms is presented in particular, the K-Nearest Neighbors classifier. Hardware accelerator platforms in the form of FPGAs are also covered, including the CAD tools used with them.

2.1 Preprocessing Methods

The collected data is most likely not organized, and may include values that are not in appropriate numerical format. Preprocessing methods tend to convert the data set to a more appropriate representation for the classifiers. There are several preprocessing

techniques used in the classification of data [5]. In most cases, preprocessing techniques are applied on the data set prior to hardware mapping.

In this thesis, the following preprocessing techniques are used:

- **Data type conversion:** Hardware designs handle only numerical data; therefore, the nominal to numerical conversion is commonly applied on data for this type of implementation. The nominal to numerical preprocessing method converts all nominal attributes to numerical representation. The most common representation is binary vectorization. In binary vectorization, the nominal values will be replaced by a sequence of bits; each bit in the vector represents one of the nominal features. The value of each bit will return a value of "0" or "1" to reflect the existence of that feature, where "0" means the feature does not exist, and "1" means the feature does exist.

- **Replacement of missing value:** Some data sets contain attributes with missing values. There are different techniques for replacing these missing values, such as replacing missing values by zeros or a constant, or replacing missing values with values calculated statistically from the other records in the training data. For example, mode or mean of the values of the feature that include missing values can be calculated and used to replace the missing feature.

- **Scaling:** This preprocessing method returns all values of the data to a specific range. The scaling can be applied for many purposes; for example normalizing data, standardization, or scaling to a unit length. In this thesis, normalization and scaling preprocessing methods are used for converting floating point values to integer values. The latter type represents a more hardware friendly format. Normal-

ization is the process of organizing the columns (attributes) of a relational database to minimize data redundancy. The most common normalization formula is shown below:

$Z_i = \frac{X_i - min(X)}{max(X) - min(X)}$, Where ..... (1.1)

X is equal to $(X_1, X_2, ..., X_n)$

Z_i is the i^{th} normalized value

X_i is the i^{th} value in the column

2.2 Supervised Learning Algorithms

Supervised learning is the task of inferring a function based on labeled training data that consists of training examples. In supervised learning, each example is a pair consisting of an input object and a desired output value (class). A supervised learning algorithm analyzes the training data and produces an inferred function which can be used for mapping new examples. Each example is called a record or an instance. Each record consists of many features and a class label as shown in Figure 2.1, where *n* is the total number of records, and *m* is the total number of features. Most supervised learning algorithms consume relatively large amounts of CPU time in terms of training the data and classifying the testing data due to the computations needed in the classification process. The time consumed has a direct relationship with the size and dimensionality of the data, and most data in many practical applications is very large.

There are many classification algorithms that classify new, unseen data according to the trained data, such as K-Nearest Neighbor (K-NN), Naive Bayesian, Artificial Neural

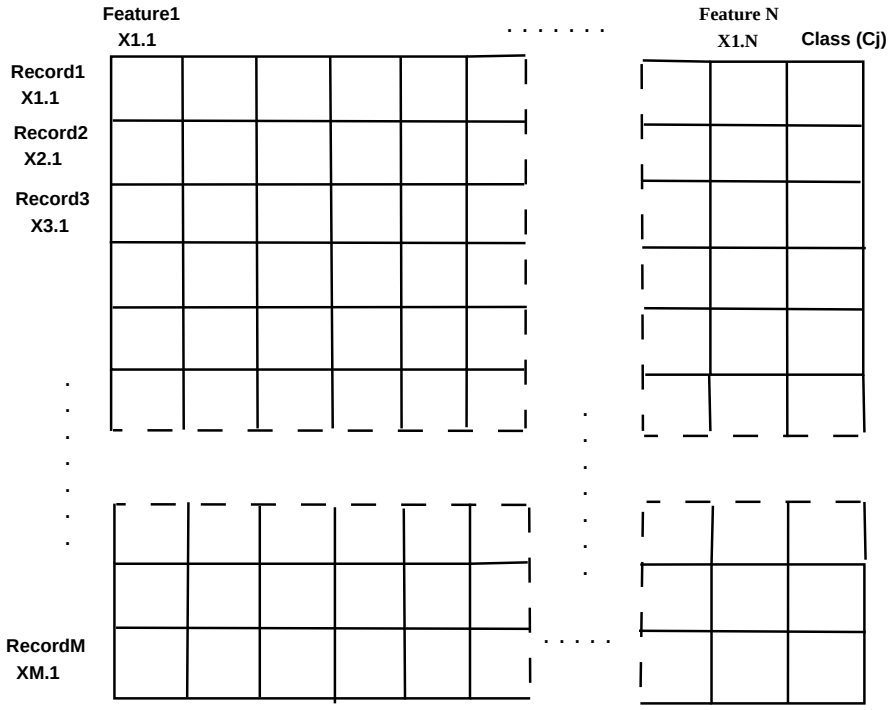

Figure 2.1: Organization of the records and features in the data sets

Networks (ANN) [6], Support Vector Machines (SVM) [7] [8] and KD-Tree, among others. In this thesis, the focus is on the K-NN classifier.

2.2.1 The K-NN Algorithm

K-NN is a non-parametric learning algorithm which means that it does not make any assumptions about the underlying data distribution. This is really useful; therefore, it is widely applied in various applications such as content-based image retrieval [9], handwriting recognition [10] and audio classification [11]. K-NN is based on simply finding the closest training examples in a feature space. The training examples are vectors in a multidimensional feature space, each with a class label. The training phase, unlike that of many other supervised learning algorithms, consists only of storing the feature vec-

tors and class labels of the training samples. K-NN is therefore considered to be a "lazy" algorithm since there is no explicit training phase. The consequence of a lack of training in the K-NN algorithm means that it keeps and uses all the training data in the testing phase. Hence, all of the training data is needed during the classification phase. This is in contrast to other supervised learning algorithms, such as ANNs and SVM, where training data can be discarded after a generalized model is developed. In the classification phase,

Algorithm 1 The K-NN Classifier algorithm

Input: X = (X_1,C_1), (X_2,C_2),....,(X_M,C_M) Classified Data
T = (T_1, ..., T_N) Unseen Data
S= ($Dist_1$,C_1),...., ($Dist_K$,C_K) Nearest Neighbors
Output:T = (T_1,C_1),(T_2,C_2), ..., (T_N,C_N)
KNN(Input,Output):
1.1 ***Foreach*** $T_i \in$ T {
1.2 ***Set*** S $\Rightarrow \infty$
1.3 ***Foreach*** $X_j \in$ X {
1.4 Dist=Calculate_Distance(T_i,X_j)
1.5 Compare_K_Values(S,Dist,X_{Cj})
1.6 }
1.7 Class(T) = Find_Majority(S)
1.8 }

Calculate_Distance(T,X):
2.1 ***Forall*** features $\in$ X and T
2.2 Dist= $\sqrt[2]{(X1.1 - T1.1)^2 + ...(X1.N - T1.N)^2}$
2.3 Return (Dist)

Compare_K_Values(S,Dis,C):
3.1 Sort({S,(Dis,C)})
3.2 Discard largest value from {S,(Dis,C)}
3.3 Return (Updated S)

the user supplies a defined constant, called *K*, and also an unlabeled vector (a test point). The K-NN algorithm attempts to classify the test point or query by assigning a label, which is most frequent among the *K* training samples nearest to that query point.

The K-NN flow is presented in Algorithm 1. The input data in Algorithm 1 is the trained points with labels C_1, ..., C_M, where *M* is the number of training instances, and

each instance has *N* features (i.e. X_1 = $X_{1.1}$, $X_{1.2}$, ... , $X_{1.N}$). The distance between each labeled instance (X_j, C_j) and the test point T_i is calculated. The calculated distances (T_i, X_j) are then sorted in ascending order to find the nearest *K* training points to the testing point. The most frequent class between the *K* minimum points is assigned to the testing point T_i. In terms of time complexity, the K-NN classifier has a high cost and depends on the search strategy used. For a linear search, the time complexity is proportional to the size of the training set for each point. If it is assumed that the points are *N*-dimensional, the straightforward implementation of finding K-NN takes O(MN) time. With approximate algorithms, this time complexity can be reduced considerably at the expense of accuracy.

Figure 2.2 offers an example of the K-NN for *K*=3 and *K*=6. The data set in this example has two dimensions ($X_{1.1}$, $X_{1.2}$), and the training set has two class labels (class *A*, designated with circles, and class *B* designated with squares). The star represents the new point (or the test data) that is to be classified. According to the vote between the majority of the three nearest neighbors, if *K*=3, the test point is assigned to class *B*; however, if *K*=6, the test point is assigned to class *A*.

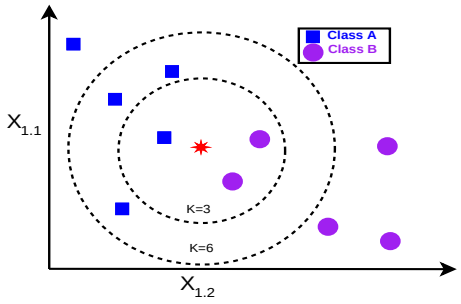

Figure 2.2: An example of K-NN classifier method for K=3, and K=6

2.3 Field Programmable Gate Arrays (FPGAs)

As mentioned in the introduction of this proposal, Field Programmable Gate Arrays (FPGAs) are good platforms for accelerating many algorithms, specifically, machine learning algorithms, because of their long execution time. Xilinx produced many versions of FPGAs, which differ in their capacity for the logic designs area, in addition to many other features. Some include soft cores, while others contain hard cores. FPGA architectures evolved from fine-grained based systems to coarse-grained architectures for implementing more complicated designs. The coarse-grained architectures include ALUs for frequently used operations, and introduce many improvements, such as producing less routing, a better cycle time, and a smaller configuration size.

Despite the existence of a variety of accelerator platforms, FPGAs are still comparable due to their customized designs that can be suitable for any application. Moreover, FPGAs are used to accelerate computations or applications by exploiting parallelism at different levels such as the bit level, the instruction level, and the architectural level. Current FPGAs rely heavily on using Electronic System Level (ESL) tools based on C, C++, and SystemC. ESL based tools tend to shorten the design time of complex algorithms.

The Xilinx Zynq FPGA is an example of a powerful platform that uses high level synthesis language and contains a hard core processor (ARM processor) running at 667MHz [2]. The Zynq board is configured by CAD tools and algorithms found in Xilinx Vivado flows. Moreover, the Zynq board is a hybrid System-on-Chip (SoC) computing platform that supports efficient communication between the ARM processor and the co-processors on the XC7000 FPGA. The most common communication interfaces are:

1. **AXI_Light interconnect:** passes data serially between the ARM processor and

co-processors in the programmable logic portion of the FPGA.

2. **AXI_BRAM interconnect:** passes data and stores it in internal BlockRAMs of the FPGA.

3. **AXIS stream interconnect with Accelerator Coherency Port (ACP):** streams data from the L2 cache memory to Co-processors in the programmable logic portion of the FPGA.

4. **AXIS stream interconnect with High Performance (HP) port:** streams data from external DDRAM to Co-processors in the programmable logic portion of the FPGA.

2.4 Tools

There are a variety of tools that are used to create designs on FPGAs. However, using high-level synthesis tools provides a great benefit, as they will shorten the implementation and the development time of the design compared to HDL implementations. This section, focuses on Vivado tools that are used with Xilinx FPGA Zynq based boards.

2.4.1 Xilinx Vivado HLS Tool

The Xilinx Vivado High Level Synthesis tool provides the opportunity to explore different architectures using high-level languages, such as C, C++, and SystemC, before transferring them into Register Transfer Level (RTL), and mapping the optimized designs onto an FPGA. Furthermore, HLS tools enable design exploration and reduce the time for final implementation. The synthesis of the design is accomplished with the help

of estimated scheduling and binding operations. The scheduling process in HLS determines in which clock cycle the operations should occur, while binding helps the HLS to define which operations are bound to which hardware resources [12]. Figure 2.3 shows the necessary steps required for the HLS tool to convert the C/C++ and SystemC code into a final bitstream to be transferred to the FPGA. First, a C/C++ or SystemC code is written in HLS tool. The functionality of the algorithm is then verified by simulating the code. Many optimization directives can be applied to the code to add parallelism and manage resources. Next, the code is synthesized into RTL. Finally, the synthesized RTL design is exported as an IP core to the Xilinx Vivado (ISE/EDK) tool where the bitstream is generated.

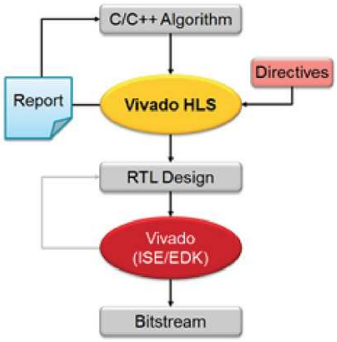

Figure 2.3: Flowchart of processing steps in Vivado HLS tools

Vivado HLS based designs consist of two main components in any design implementation: the Top Function (TF) file and the Test Bench (TB). The TF file contains all synthesized operations that are converted into the core processor on the FPGA, while the TB file contains all unsynthesized operations (such as dynamic memory allocation and I/O operations). In addition it contains the verification code that is added in order to verify the functionality of the code with C simulation and C/RTL co-simulation. The

operations in the TB file are transferred and run on any general-purpose processor (soft core, or hard core like the ARM processor on a Zync FPGA). As shown in Figure 2.4, data is transferred and processed by the FPGA in the following steps:

1. Data is usually stored in external memory since most data sizes are too large to be stored locally in the BlockRAM of the FPGA fabric.

2. The data is passed to the ARM processor via dedicated buses (I/O buses).

3. The data is then transferred from the ARM processor through dedicated ports to the hardware accelerator in the FPGA, and stored in BRAMs or registers.

4. Results are finally sent back to the ARM processor through dedicated ports to be analyzed.

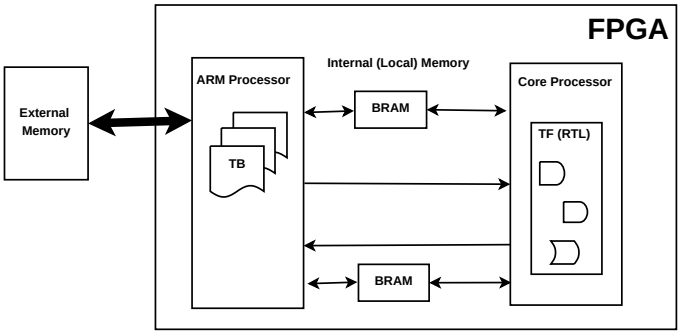

Figure 2.4: A simple block diagram of implementation of HLS design in FPGA

Vivado HLS Directives

HLS optimization directives can be applied into the code within regions. They can be implemented as Tcl commands or pragmas in HLS using a synthesized GUI. Three classes of directives are used by HLS to optimize and control the design:

- Interface-based directives: optimize the type of communication connections and protocols.
- Resources-based or Area-based directives: optimize the resources used (BRAMs, DSPs, LUTs, and FFs).
- Performance-based directives: optimize the timing and improve the speedup of the implemented designs.

Below are some important directives that are used in this thesis:

1. ***The Loop_Tripcount Directive:***

   The loop_tripcount directive defines the minimum, average, and maximum iterations performed in a loop. By applying the tripcount, the latency will be identified accurately because it represents the latency *in clock cycles* of the loop body multiplied by the total number of iterations in the loop. For example, if the (*if*) instruction in the following loop is true, the loop will terminate with a minimum of seven trip counts; otherwise, it will terminate after ten trip counts. Therefore, by specifying the loop_tripcount with min = 7 and max = 10, the HLS tool will provide a more accurate latency value. The minimum and maximum latency values are calculated by ***(the time consumed by one trip count inside the loop*** x ***min trip count / max trip count)***, respectively. If the loop_tripcount directive is missing in the code, the HLS tool will not report reasonable or accurate latency values.

```
for(i=0; i < 10; i++)
{
    #pragma HLS loop_tripcount min=3 max=10
     if(x < n)
        i = i + 3;
}
```

2. ***The Inline Directive:***

   The inline directive inlines the functions from their hierarchy level to the upper level function that is calling them. The functions will be treated as part of the function that is calling them rather than as separate entities, as shown in Figure 2.5. The inlined function cannot be shared, which affects the consumed area. Furthermore, the inlining is performed **by default** only to the next level of the hierarchy. For example, if there are three nested functions, as shown in the following code, by applying the inline directive into C ( ), the function will be inlined into the B ( ) function only.

   If it is necessary to inline the C ( ) function into both B ( ) and A ( ), the inline directive should be applied to both the C ( ) and the B ( ) functions.

```
/* The nested functions */
      Function A() {
             Function B() {
                  Function C()
            }
       }
```

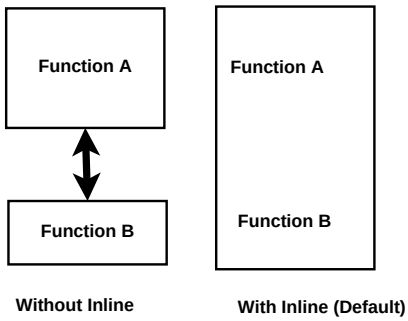

Figure 2.5: Implementation of Inline directive in Vivado HLS

3. ***The Loop Unroll Directive:***

   The general idea of loop unrolling is to replicate the code inside a loop body a number of times. The number of copies is termed the loop unrolling factor (F). The number of copies can be specified by the user, or it can be left to the compiler's default option. In the default option, the tool creates a number of copies of loop bodies equal to the total trip count of the loop (the total number of iterations), if there are enough resources; otherwise, "F" copies will be created. The loop boundaries should be known in order to apply unrolling. The implementation of the unrolling directive in Vivado HLS tool is shown in Figure 2.6, and the timing diagram for the instructions with and without unrolling is shown in Figure 2.7. Assuming the execution of the loop body takes one clock cycle, the loop unrolling will run in one clock cycle instead of *(one clock cycle* x *trip_count)*. The implementation of the loop unroll directive, shown in Figure 2.6, is represented in the following code:

```
for (i=0; i < 3 ; i++)
{
    Loop Body
}
```

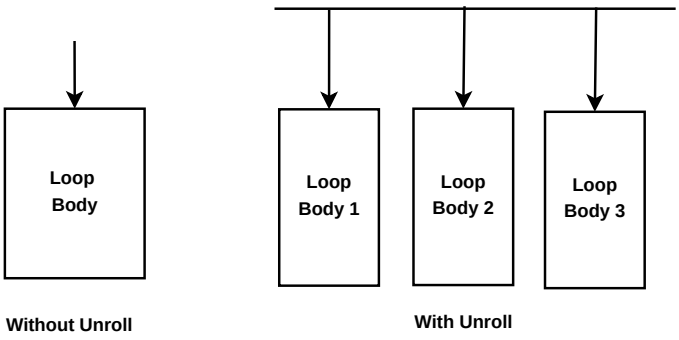

Figure 2.6: Implementation of Loop Unroll directive in Vivado HLS

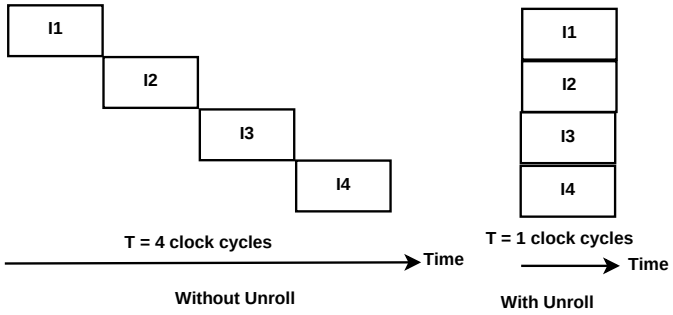

Figure 2.7: The timing diagram of instructions with and without unrolling directive

4. ***The Pipeline Directive:***

   Pipelining is an optimization that offers one of the major performance advantages of hardware over software. It allows concurrent or parallel operations to be automatically implemented in an RTL design. The pipeline directive can be applied to both functions and loops. The HLS tool creates registers between the operations in RTL to save the intermediate results between the operations, and it also tends to achieve parallelism between them. In Vivado HLS, by applying the pipeline directive, a new input will be processed in the loop or function after a number of clock cycles (the interval value). The initial interval (II) can be set by the user to a specific value, and the HLS tools will try to satisfy this value. If the tool is unable

to satisfy the II value, it will attempt to satisfy the most optimized value. Figure 2.8 represents an example of the timing diagram before and after applying the pipeline directive, where I is the instruction duration and T is the total time of processing one input in clock cycles. The following code example is presented in Figure 2.9, which shows how the data is transferred between the operations 1, 2 and 3 in each clock cycle. In addition, Figure 2.9 shows how the registers are used to save the intermediate results after each clock cycle.

```
Function A {
   int n,m;
   int x[5] = {1,2,3,4,5};
   for (i=0; i<5; i++)
        { operation1: n = x[i] * 2;
          operation2: m = x[i] * 7;
          operation3: Sum = n + m;
       }
}
```

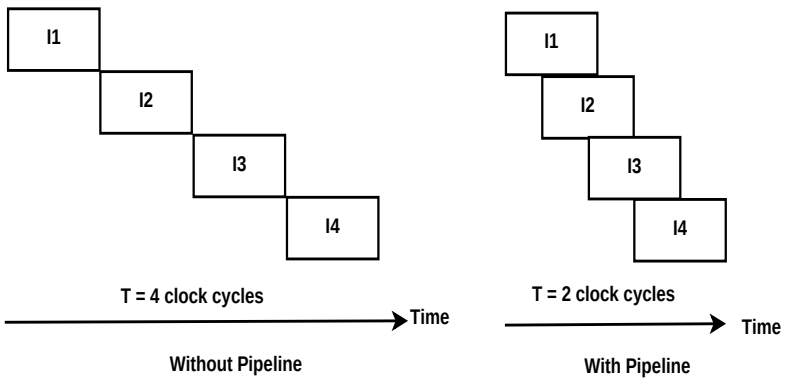

Figure 2.8: The timing diagram instructions with and without pipeline directive

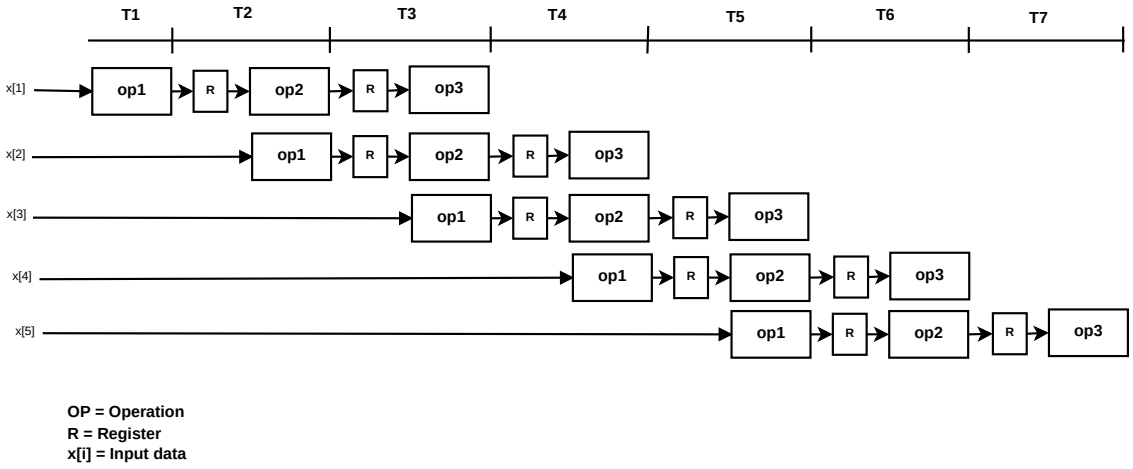

Figure 2.9: The timing diagram of the pipeline

5. ***The DataFlow Directive:***

The goal of dataflow is to express parallelism at a coarse-grain level. HLS extracts this level of parallelism by evaluating the interactions between a program's different functions, based on their inputs and outputs. In the simplest case, when a function deals with different data and when there is no communication between them, the HLS tool allocates resources for each function to run independently. However, the more common scenario in any complex system is when a function provides results to another function. In this case, the HLS supports two models. The first is the PingPong model, where the HLS tool creates a pair of BRAM memories, arranged as memory banks ping and pong. Each function can access only one memory bank, ping or pong, for the duration of a function call. Whenever a new function call begins, the HLS-generated circuit switches the memory connections for both the producer function and the consumer function. This approach guarantees functional correctness, but limits the level of achievable parallelism across

function calls.

In the FIFO model, the consumer function can work with partial results from the producer function. The same two resources will be created for each function, and the data will be passed sequentially to the producer function. The consumer function will wait for partial results from the producer for a specific time (initial interval II). Both functions will work in parallel. In this way, the data can be provided to the next function before the operations in the previous function have ended. The dataflow directive goes **by default** with PingPong, and it has better effects when it is combined with the pipeline and unroll directives. The dataflow style is pre-set using the configuration directive (config_dataflow directive). Figure 2.10 shows the difference between the FIFO and PingPong styles.

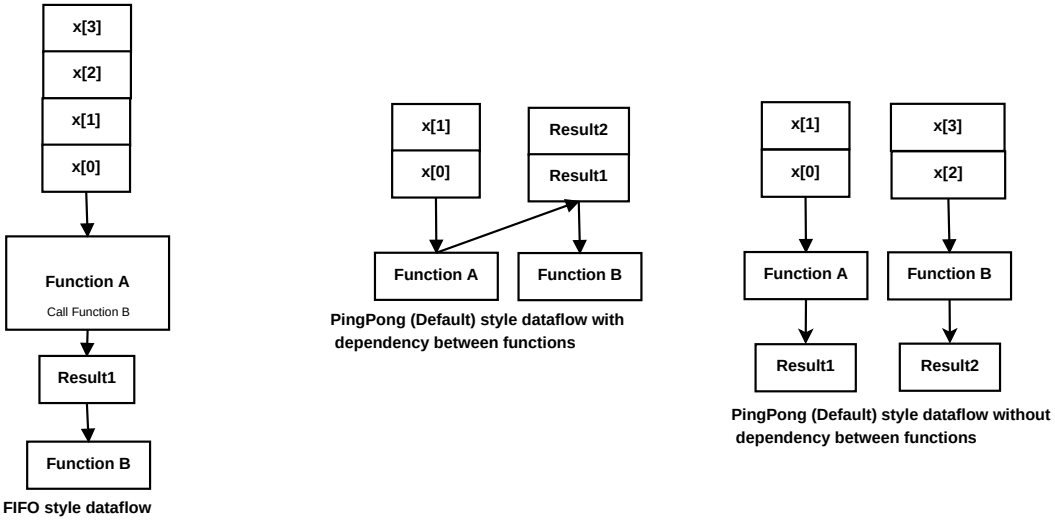

Figure 2.10: The behaviour of the FIFO and PingPong data styles in Dataflow

6. ***The Array_Partition Directive:***

In the case of a multi-dimensional array, the array_partition directive partitions

one large memory or BRAM into smaller separate memories, with separate read and write ports for each one. The number of partitioned memories is specified as an argument in the directive's options. In the case of a one-dimensional array, a complete array_partitioning will partition the memory contents into registers. This directive allows the user to partition the array either as:

- Complete: all data elements in the array will be converted into registers.
- Cyclic: the data elements will be partitioned with a specific dimension (set by parameter dim) in cyclic order.
- Block: the data elements will be partitioned with a specific dimension (set by parameter dim) in sequential order as blocks.

The array_partition directive allows data to be passed in parallel into processing elements or functions. The performance will be improved; however the area (the size of the Demultiplexer) will be increased. Figure 2.11 shows an implementation of the three types of partitioning, in which the order of the stored elements is organized sequentially in block partitioning, and cyclically in cyclic partitioning. The single arrow represents an individual bus that connects between the BRAM and the design circuit.

2.4.2 Xtensa Tensilica Tool

An application-specific instruction set processor (ASIP) is a component of the system-on-a-chip design. ASIP instructions are designed to be a general instruction set customized for a specific application. The specialization of the core provides both flexibility

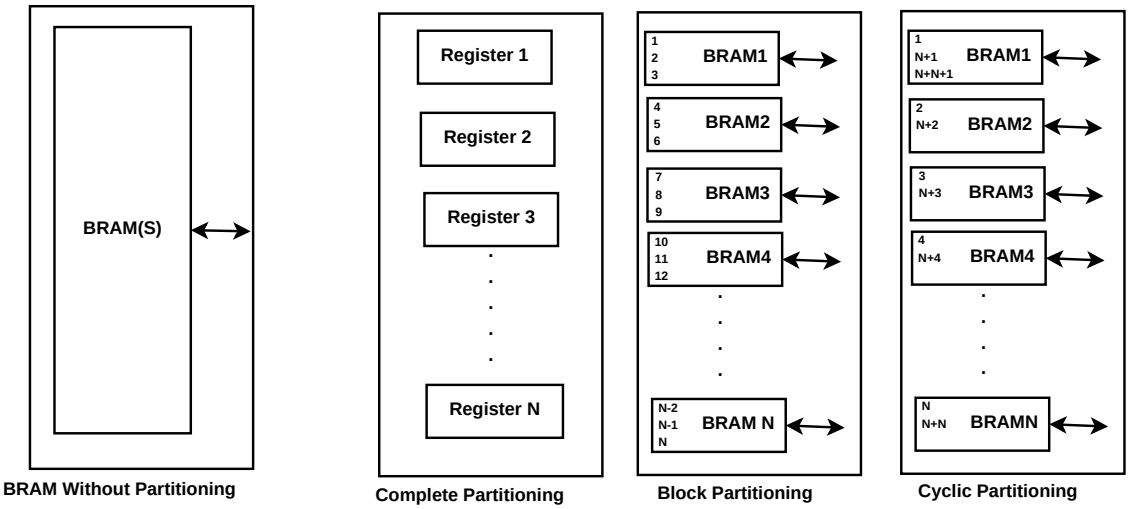

Figure 2.11: The partitioning of the array: complete, block, and cyclic

and improvement in performance, which combines the benefits of the CPU in terms of flexibility, and ASIC in terms of high performance [13].

An ASIP is a configurable processor in which the instruction set is tailored to benefit a specific application. The main reasons for investigating ASIPs are twofold: First, they combine the flexibility of a GPP with the performance of a pure RTL implementation. Second, they avoid the communication costs associated with co-processors.

The work presented in this thesis targets the Tensilica Xtensa family of processors [3]. Xtensa processors are configurable cores that can be initialized and described at the micro-architectural level. Moreover, they can be extended at the instruction-set level. Modifying the instruction set and the datapath with appropriate instructions and hardware extensions can not only speed up the application, but can also reduce design time. The designer is only required to implement the specialized instructions and hardware extensions, while the rest of the processor design steps are automatically performed by the Tensilica tool.

ASIPs can be used for accelerating applications with a minimum development time by modifying the software instructions. Figure 2.12 presents the design flow of the Xtensa Tensilica tool [14].

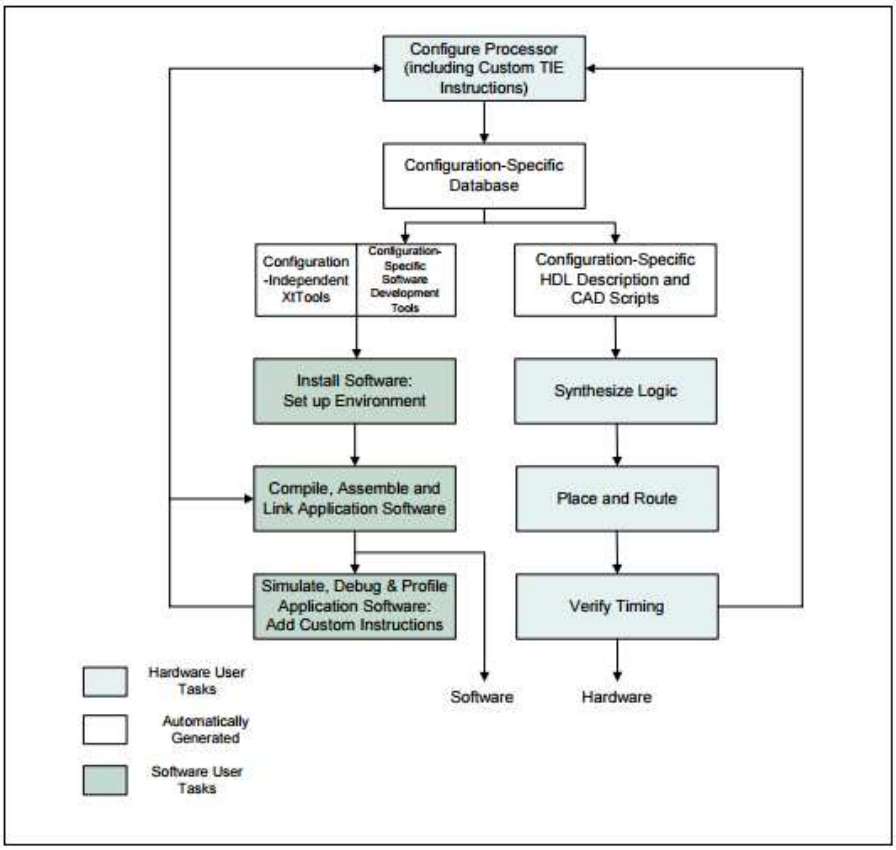

Figure 2.12: A simple block diagram of Tensilica design flow

2.5 Summary

This chapter presents a brief background of the most common pre-processing techniques and supervised machine-learning algorithms. Supervised learning algorithms use a training set (the labeled data) to create a model capable of predicting the class of the unlabeled data (testing set). The use of classification algorithms, such as the K-NN algorithm, is increasing for embedded systems applications such as medical and health care applications [15], auto industry [16], voice recognition [11], and image applications [17]. The acceleration of machine-learning algorithms is becoming an essential task due to the need for real-time performance. FPGAs are considered excellent candidates for accelerating machine learning algorithms due to their flexibility, low power consumption, and reconfigurability features. Since complex algorithms can be spatially mapped onto the FPGA fabric, orders of magnitude of speedup can be achieved, which enable real-time performance. In Chapter 3, a literature review of the hardware implementations for the K-NN algorithm is conducted to identify the advantages and disadvantages of the current published works.

Chapter 3

Literature Review

This chapter presents the most significant previous works in reconfigurable computing related to the acceleration of different supervised learning algorithms. In general, many supervised learning algorithms can be excellent candidates for acceleration in hardware. The main objective of this chapter is to highlight the advantages and the disadvantages of the recent published works in accelerating supervised learning algorithms, more specifically, the K-NN algorithm. This chapter also provide guidance as to possible future work in this field.

3.1 Accelerating Supervised Learning Algorithms

One of the main characteristics that is used to describe the efficiency of machine-learning algorithms is run time. Therefore, several works can be identified for accelerating machine-learning algorithms in literature based on their run time. In this section, the focus will be on current published works on hardware acceleration for various classifiers using FPGAs.

Different classifiers are successfully accelerated using FPGAs, such as naïve Bayes [18], [19], [20], [21] and [22], Artificial Neural Networks [23], [24], [25] and [26], Support Vector Machines [27] [28] [29] [30] and [31] and Decision Trees [32], [33], [34], [35] and [36]. The next section focuses on work related to the K-NN Classifier.

3.2 K-Nearest Neighbors Accelerators

Several architectures have been proposed in the literature for the K-NN algorithm. In [37], a SIMD-style architecture for pattern recognition is presented. This is one of the first proposed architectures and implementations of the K-NN classifier. The work in [37] offers a conceptual design that is considered to be impractical. This is due to the reduction in execution time, which determines the operations of the K-NN algorithm in two paths: the forward path to calculate the distance, and the backward path to find the winner class. This work is presented in schematic form and the actual implementation is not covered. The work in [38] presents two different architectures in linear array style that are described as soft parameterized IP cores in VHDL. The cores are used to synthesize and evaluate different array architectures for various K-NN problem instances and Xilinx FPGA platforms. The design in [38] has been validated using the Breast Cancer Wisconsin (BCW-Diagnostic) dataset from the UCI Machine-Learning Data Repository [39]. In [40], a systolic algorithm for the k-nearest neighbors problem is presented. The proposed algorithm can solve the K-NN efficiently with $\sqrt{N \times K}$ processing elements. They defined N as the problem size and K as the sum of all K-values. In [41], the authors present an accelerator for the k-th nearest neighbor thinning algorithm for reducing vectors in a multi-dimensional feature space to smaller representative vectors. The authors

report speedups of almost 385x over existing software implementations for raw data in embedded computing implementations in FPGAs, and a 6.6x speedup over a software reference used for multi-objective evolutionary optimization. The authors in [42] implemented the K-NN classifier as a systolic array in two different designs on an FPGA. They reconfigured the value of K (number of neighbors) dynamically during execution time. The authors reported a speedup of 76x in the first architecture and 68x in the second architecture over a general-purpose processor (GPP). In [43], the authors used Handel-C to accelerate the K-NN algorithm by employing multiple processing elements for the multiplication and addition operations, in addition to pipelining. Two data sets were used to evaluate their work: the breast cancer set, and the prostate cancer set. The authors achieved a speedup up to 33x over the software running in CPU with a reduction in accuracy of 5.66% in the breast cancer dataset and 7.95% in the prostate cancer dataset.

In other works, such as [44] and [45], the K-NN algorithm was accelerated by using the Graphical Processing Unit (GPU). In [44], the GPU with its multi-leveled memory is used to parallelize the K-NN calculations without the square root from the Euclidean distance calculation. The radix sort was used for the sorting stage of the algorithm. The authors exploited the hierarchy of the existing memory. Two data sets were used for evaluation and comparison purposes. The work in [44] achieved an average speedup of 33.76x over the software running in CPU, and an average speedup of 15.17x over the ANN-brute algorithm. However, this work does not scale well for other data sets.

According to the literature, there are a few examples of mapping multiple classifiers on the hardware, especially on FPGAs. Some of the efforts found are in [46] and [47]. In [46], the drawback of the brute-force K-NN classifier was observed. This includes

its unsuitability for some real-time applications that need to be updated on the fly, such as multimedia traffic on the Internet. Therefore, they improved the brute-force K-NN by proposing Locality Sensitive Hashing (LSH) for the K-NN classifier. The authors in this work focused on four goals: the accuracy of classification, the throughput of packets, the ability to update data, and the area used. The hamming distance method was used in LSH for the K-NN classifier to calculate the distance in order to prevent the multiplication operation that exists in the Euclidean distance method. The proposed parallel architecture was mapped into an FPGA to enhance the throughput, and achieved 80Gbps. The achieved accuracy of the classifier was 99% for a large size of data, and the area used was relatively small. Achieving more than one goal in their work shows the critical study, analysis, and effort of the authors. Using the FPGA's dual-port RAM represents a good usage of the available resources. Moreover, this work promotes the importance of using FPGAs for real-time applications.

With respect to hybrid machine learning-methods and techniques, [48] shows the extensive value of some preprocessing techniques that can be used for speeding up the classification techniques. PCA was used in this work to reduce the distance calculations in the K-NN algorithm by projecting the training points into feature space, and to sort each projected point. An index was associated with each point. K points were found in each feature dimension. Applying PCA to training data can result in a bottleneck; however, a great speedup can be achieved in the testing stage. This work is suitable for real-time applications, in which the testing data are continuous. The developed algorithm in [48] was not implemented in hardware, which limited the amount of speedup that can be achieved.

3.3 Hardware Platforms and Implementations

3.3.1 FPGA versus GPUs

As GPUs are comparable to FPGAs in performance for many applications, there have been many studies comparing FPGAs and GPUs, such as [49], [50], [51], [52], [53], and [54]. In [49] the authors compare the performance of the NVidia GeForce 8500GT GPU and the Altera Stratix III EP3SE110-F780C2 with respect to Gilbert's SVM algorithm. They observed that the performance of both the FPGA and the GPU are increased with an increasing dimensionality of the dataset. The FPGA outperformed the GPU when the FPGA has enough BlockRAMs to store data. Otherwise, other techniques need to be implemented to split the data into smaller subsets in order to retain this performance. FPGAs outperformed GPUs in a comparison made in [50] for different applications: Gaussian Elimination, Data Encryption Standard, and Needleman-Wunsch sequence alignment. A comparison was also made in terms of code complexity, where GPUs have less code complexity than FPGAs. The complexity of code in FPGAs was reduced after introducing a high level synthesis in Xilinx Vivado HLS in 2013. In [50], the authors recommended a better accelerator according to the type of the application, where GPUs can perform better when the application needs an external memory and contains independent parallel computations, such as a sparse matrix vector multiply [54].

Pauwels et al. in [52] built their work upon that of [51]. In [52], a comparison between the FPGA and the GPU was made for real-time vision architectures. In [51], the authors realized that a customized FPGA exceeds the performance achieved by GPUs in the presence of data dependencies. However, Pauwels et al. argued about the simplicity of the model that the authors in [51] used. Pauwels et al. defended their argument by

applying larger models to both accelerators, and expanding the range of comparison. They found that the GPU exceeds the FPGA in the following areas: accuracy, absolute speed, design time, high-performance computing, and cost. In comparison, the FPGA outperforms in the following areas: power consumption, normalized speed, flexibility, and embedded platforms. However, this comparison is still lacking other points, such as the amount of resources used.

For Sliding-Window applications, the authors in [53] achieved results based on comparisons between FPGAs, GPUs, and OpenCL Multicore in terms of energy and performance. The implemented operations were a sum of the absolute difference for content-based image retrieval application and 2D convolution, which are commonly used in digital signal processing for embedded computing, and cross entropy for image comparison applications. According to their results, FPGAs can achieve a speedup of up to 11x and 57x over GPU and Multicore, respectively.

The authors in [55] compare the performance and power consumption of the K-NN design among three platforms: the CPU, the GPU and the FPGA. In general, most of the works that compare hardware accelerators in FPGAs over GPUs, conclude that GPUs are most appropriate in non-embedded based applications that need to be accelerated regardless of high power consumption. FPGAs are the most suitable devices for embedded real-time based applications due to their low power consumption.

3.3.2 Interconnect Communication Overhead

Even though FPGAs are considered an excellent platform for the purpose of hardware acceleration for machine-learning applications, selecting the most appropriate interconnect media for transferring data between the host processor and the accelerated IP can

affect the overall performance of the hardware system.

In the literature few works published results on this issue in spite of its important role in achieving good acceleration. The first work that focuses on the communication overhead is [56]. In this work, the authors assessed the capabilities and performance achieved by comparing the AXI ACP port and AXI HP port in the Xilinx Zynq 70C20 FPGA board. Their study revealed that the type of hardware implementation and the size of the application data have a major impact on the overall speedup and energy consumption. For applications that need to be implemented as a hardware/software co-design, they suggest to use an AXI ACP port with CPU On-Chip Memory (OCM) for a smaller data size, and an AXI HP with CPU DDRAM with a largeer data size. However, in the case of pure IP accelerator implementations, they advise using AXI ACP, or AXI ACP and CPU OCM when the size of the data is smaller than the size of the cache and the OCM memory. The use of AXI HP is suggested for data with a larger size. This work provides a good introduction to communication overhead bottleneck issue between the Processing System (PS) and the Programmable Logic (PL) portions in the Zynq FPGAs; however, several available interconnection methods are not covered.

The authors in [57] expand on the previous work by measuring the throughput of six types of data transfer between the ARM processor and the accelerator. The authors built their analysis depending only on the size of the transferred data and the size of each burst transferred. Their advice was slightly different than that of the authors of [56]. The authors in [56] built their selection of the interconnect port according to the data size ranges as follows: they select 1) the AXI GP for data range (16B - 64B); 2) AXI ACP with OCM and cache enabled mode for data range (128B - 2048B); 3) AXI ACP with COM and cache disabled mode for data range (4K - 64K); and 4) external DDRAM and

AXI HP for larger sizes.

In [58], the authors published their analytical study about the performance of all possible data paths between the programmable logic and processing system in the Xilinx Zynq 7000 board. They created different analytical models based on three parameters: PL frequency, burst length and port width ratio. These models were applied to small, medium and large data sizes. After extensive experiments, they claimed to produce an accurate performance measure for data path parameters considered by user designs. Their results include a maximum and average error rate of 5% and 1% respectively, representing the differences between the theoretical and experimental measurements. The latter work emphasizes the huge impact that the selected data path method has on the overall performance measure.

3.4 Summary

Based on the literature covered in this chapter, there are several implementations of hardware classifiers for the K-NN algorithm. This thesis proposes the following improvements to advance the state of the art:

1. Design and implement a tightly coupled application specific instruction processor (ASIP) for the K-NN algorithm.

2. Perform an extensive design exploration for the K-NN algorithm with the minimum development time using Xilinx HLS tools.

3. Compare the accelerated K-NN algorithm in two different hardware implementations: tightly coupled architectures using the Cadence Tensilica tool (ASIP), and

semi-tightly coupled architectures using the Xilinx Vivado HLS tool.

4. Verify how different methods of passing data from the CPU to the hardware accelerator can either improve or degrade the overall performance of the proposed design.

Chapter 4

Overall Design Exploration Methodology

4.1 Data Sets

Table 4.1 shows the benchmarks used to evaluate the performance of the hardware accelerators used in this work. Twelve different types and sizes of data sets, all from the UCI Machine Learning Repository [59], are used in these experiments. Several preprocessing techniques were applied to all of the datasets, including nominal to numerical conversion, replacement of missing values, and normalizing and scaling of all data.

The data was split into a training set and a testing set using a 10-fold cross validation technique. The ratio of 90% of the data was allocated for training and 10% for testing for each of the 10 runs. The final accuracy represents the average accuracy obtained from the 10 folds. All classification results obtained via the hardware accelerators were verified in terms of accuracy to be equivalent to those obtained via the pure software

implementation running on a desktop computer.

Benchmark	Records (Instances)	Features	Classes
BCW_9	699	9	2
Wine_R	1599	11	10
Wine_W	4898	11	10
HD_Long	200	13	5
HD_Hung	294	13	5
HD_Clev	303	13	5
Credit_g	1000	20	2
BCW_30	569	30	2
Spectf	267	44	2
Spambase	4601	57	2
Mfeat	2000	76	10
RobotF	164	90	5

Table 4.1: Benchmarks: Records, Features and Classes

4.2 Preprocessing

The benchmarks employed in this work contain missing data, floating point values and feature values of different ranges and types. Therefore, several preprocessing techniques were applied on these data sets using the Waikato Environment for Knowledge Analysis (WEKA) 3.7 application tool [60].

4.2.1 Data Type Conversion

There is more than one method for data type conversion in the WEKA tool. For the data used in this thesis, a nominal to numerical conversion was required. All nominal data used in the benchmarks were to rank class labels. Therefore, the unsupervised filter *"RenameNominalValues"* was used to replace each nominal class value with a representative number, I (I=1, 2, ... , number of classes). This simple method was sufficient for the data sets used in this thesis as the nominal values were all found to be within the class values.

4.2.2 Replacement of Missing Values

Many records within the benchmarks used for evaluation included missing values. Hardware synthesis designs, unlike some software compilers, cannot handle data with missing values. The WEKA filter *"ReplaceMissingValues"* was therefore used to replace missing values by the average of the attribute values that the missing items belong to.

4.2.3 Scaling

By applying the K-NN algorithm to any data set, it is necessary to have the values of the features of the data set within a specific range; otherwise, the feature with large values will dominate the results of the distance calculation. In this case, normalized data values serve the purpose of providing better accuracies. The unsupervised filter *"Normalize"* in WEKA was applied to all twelve data sets.

The problem with the normalized data is that the data values ranges will be between 0-1 or (-1) - (+1) (i.e. a floating point). This adds more complexity to the hardware design and leads to more consumption of resources and power. All data values have therefore been scaled to a range between (0-50000) to avoid the floating point, and at the same time to return as many mantissa values as possible.

4.3 K-NN Algorithm

The K-NN algorithm task of classifying records builds on three main functions: 1) Calculating the distance between the training points and the testing point. 2) Finding the nearest K points to the testing point. 3) Finding the winner class among the closest K

points.

1. ***Calculating the distance:*** There are a variety of distance functions, such as Euclidean distance, Manhattan distance, and Hamming distance. Each method is suitable for a specific data type. In this work, all data types are converted to numerical values to be suitable for hardware implementation. Also, all values in the data set are scaled to positive integers; therefore the Euclidean distance function was used to achieve better accuracies. In the K-NN implementation in this thesis, the square root operation is removed from the algorithm, similar to what was done in [41] because its complex hardware design that will consume a large amount of resources and power. This change does not have an impact on the algorithm's accuracy because the K-NN algorithm's accuracy is calculated by comparing distances between training points and the testing point. The changes which result from removing the square root operation will therefore affect all distance calculations equally.

2. ***Finding the nearest K points:*** To find the nearest K points, different searching algorithms can be used. In the K-NN algorithms used in this work, the K values were initially set to a maximum value. Each new distance that is calculated is compared with the highest distance value, which is supposed to be Neighbor #1. If it is less, all distance neighbors are pushed down, and the new distance is saved in Neighbor Location #1. Otherwise, the comparison operation is propagated to the next location (#2 and so on) until all of the distances between the testing point and all of the training points are checked.

3. ***Finding the Winner:*** A number of counters reflecting the number of classes are

initiated to zero. For the K nearest neighbors, whenever a specific label appears, its reflected counter increases by 1. The label of the counter with highest count is the winner.

4.3.1 The *K* Hyper-Parameter

One of the main parameters that determines the performance of the K-NN algorithm is '*K*', the number of neighbors for the test point. An extensive evaluation on the 12 data sets for the K-NN algorithm is performed. Each run is repeated five times and the average accuracy is recorded. Based on the results of this extensive experimentation, a value of *K*=10 was found to give an overall adequate accuracy for all of the data sets used in this thesis.

4.3.2 Profiling

A C-version of the K-NN algorithm (with profiling enabled) was first executed on the Tensilica Xtensa processor to determine the bottlenecks and the parts of the algorithm that can be accelerated. This version of the K-NN was reffered to as the *baseline*. The mapping of the baseline to the Xtensa processor was performed automatically us-

Benchmark	Calc_Dist()	Comp_K_Val()	Find_Maj()
BCW_9	56%	40%	3%
BCW_30	80%	18%	1%
Spectf	77%	21%	1%
Spambase	81%	18%	1%
Mfeat	91%	8%	1%
RobotF	92%	7%	0%
Average	80%	19%	1%

Table 4.2: Results of Tensilica Profiling

ing Xtensa Xplorer Workbench 6.0 and its various compiling, debugging, profiling and

hardware-generation tools. Profiling results were collected for six of the benchmarks identified in Table 4.1. Each benchmark produced similar profiling trends when executed by the baseline code. Table 4.2 shows the profiling results for six of the benchmarks introduced earlier. Note that Table 4.2 only shows the main functions of Algorithm 1 introduced in Chapter 2, while neglecting minor and less computational functions. It can be seen from Table 4.2 that the Calculate_Distance( ) function is the major bottleneck in the code, accounting for approximately 80% percent of the runtime on average.

4.4 Thesis Framework

In this section, the overall methodology of the thesis frame work will be summarized. Figure 4.1 presents the phases of the frame work that is accomplished. The thesis framework, which will be explained in more detail in Chapter 5, includes the following main stages:

1. An extensive design exploration of semi-tightly coupled architectures using Xilinx Vivado HLS optimization directives is explained for custom and general architectures in Chapter 5. Execution time, resources used and power consumed are recorded for pre-synthesis and post-synthesis in Chapter 6.

2. Evaluation of the interconnects communication between the ARM processor and the K-NN IP core in the Zynq FPGA board is studied in Chapter 5. Execution time, resources used and power consumed are recorded for pre-synthesis and post-synthesis in Chapter 6.

3. Design exploration of tightly coupled architectures is accomplished using the Xtensa

Tensilica Application Specific Instructions Processor (ASIP) tool in Chapter 5. Execution time, resources used and power consumed are recorded for multiple ASIP architectures in Chapter 6.

4. A pure software implementation for the K-NN algorithm is run on the ARM processor. The execution time is recorded for 12 UCI Benchmarks.

5. The results obtained in stages 1, 2, 3 and 4 are analyzed and compared in Chapter 6 in terms of speedup, resources used, and power consumed.

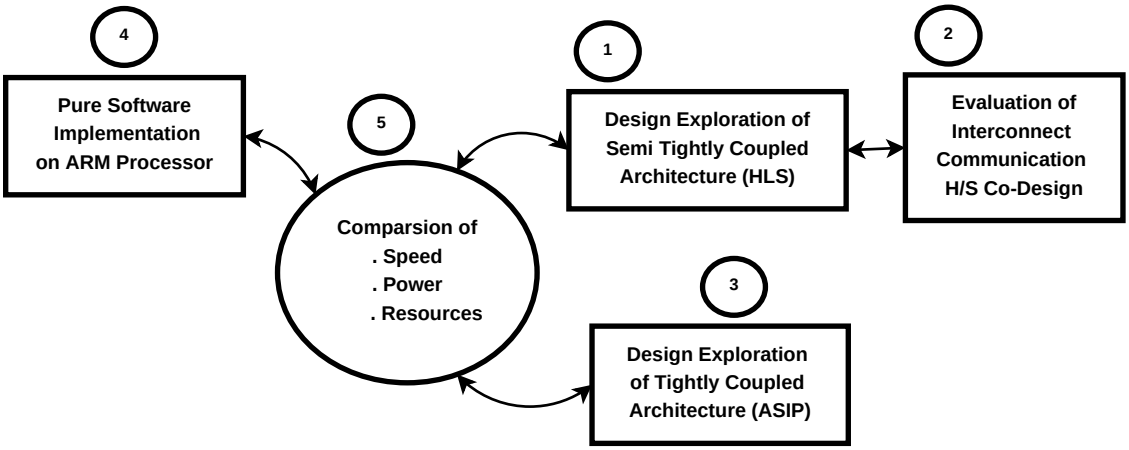

Figure 4.1: Thesis framework

Chapter 5

Methodology: Design Exploration

This chapter presents different types of architectures that are designed to accelerate the K-NN machine-learning algorithm. The main difference among the proposed implementations is based on the placement of the hardware accelerator. Hardware accelerators can be placed at different locations with respect to the processing element ranging from tightly coupled to loosely coupled. In this thesis, the hardware accelerators for K-NN were implemented using both the functional unit (tightly coupled) and the co-processor (semi-tightly coupled) approaches. Design exploration of several architectures were performed using Xilinx Vivado HLS and Cadence Tensilica tools [3]. The advantages and disadvantages of each approach are highlighted and compared between these two approaches in terms of the resources used, the power consumed and the speedup achieved.

5.1 Tightly Coupled Architectures

In this section, the Tensilica Software Development Toolkit [3] is used to explore (tightly-coupled) hardware accelerators. With this approach, Application Specific Instruction Processors (ASIPs) will be presented for accelerating the K-NN Algorithm 1 presented in Chapter 2. Overall, the ASIP approach described here relies on running the software version of the K-NN algorithm on the Xtensa processor. The software is tested and then profiled to determine the main bottlenecks in the code. The TIE language is then used to convert the bottlenecks into specific merged instructions and custom hardware to enhance performance. These steps are explained in detail below.

5.1.1 Overall ASIP Approach

The implementation of the new ASIP tightly coupled processor involves creating special instructions and custom micro-architectures in which these instructions are executed. Based on the profiling results introduced earlier in Chapter 4, the following important points were observed:

i The three main sub-functions were rewritten using the TIE language [4], and produced the following three architectures:

- The architecture of the first sub-function "Calculate_Distance()", which is responsible for calculating the distance between the new test point (T_i) and its neighbors(s) (X_j), as shown in Figure 5.1.
- The architecture of the second sub-function "Compare_K_Values()" which is responsible for determining the "K" nearest neighbors to the test point (T_i),

as shown in Figure 5.2.

- The architecture of the third sub-function “Find_Majority()”, which identifies the majority class among the “K” nearest neighbors, is presented in Figure 5.3.

ii The number of features (attributes used in classification) of the different datasets is an important parameter that should be accessed and handled with ease by all of the instructions. Therefore, a dedicated register of size 16-bits (65,536 features) capturing the number of attributes of each dataset was allocated within the architecture, as shown in Figure 5.1.

iii Every test point (T_i) is accessed frequently, and is therefore stored in a special register file. However, different datasets often have different feature types and sizes. In order to enhance the generality of the developed ASIP architecture, a “16x16-bit” register file is implemented to accommodate 16 features, each of a size of 16-bits, as shown in Figure 5.1.

iv Fast storage for the computed current distance is required; therefore a dedicated 64-bit register is used for this task.

v The *K*=10 neighbors are accessed extensively; therefore a register file of size 10 x 72 bits is created (64-bits are reserved for the distance and 8-bits for the Class), as shown in Figure 5.2.

5.1.2 The "Distance Calculation" Module

Figure 5.1 shows a block diagram for the Calculate_Distance() module (steps 2.1 - 2.3 of Algorithm 1 introduced in Section 2.1.1). This figure is comprised of four distinct sub-

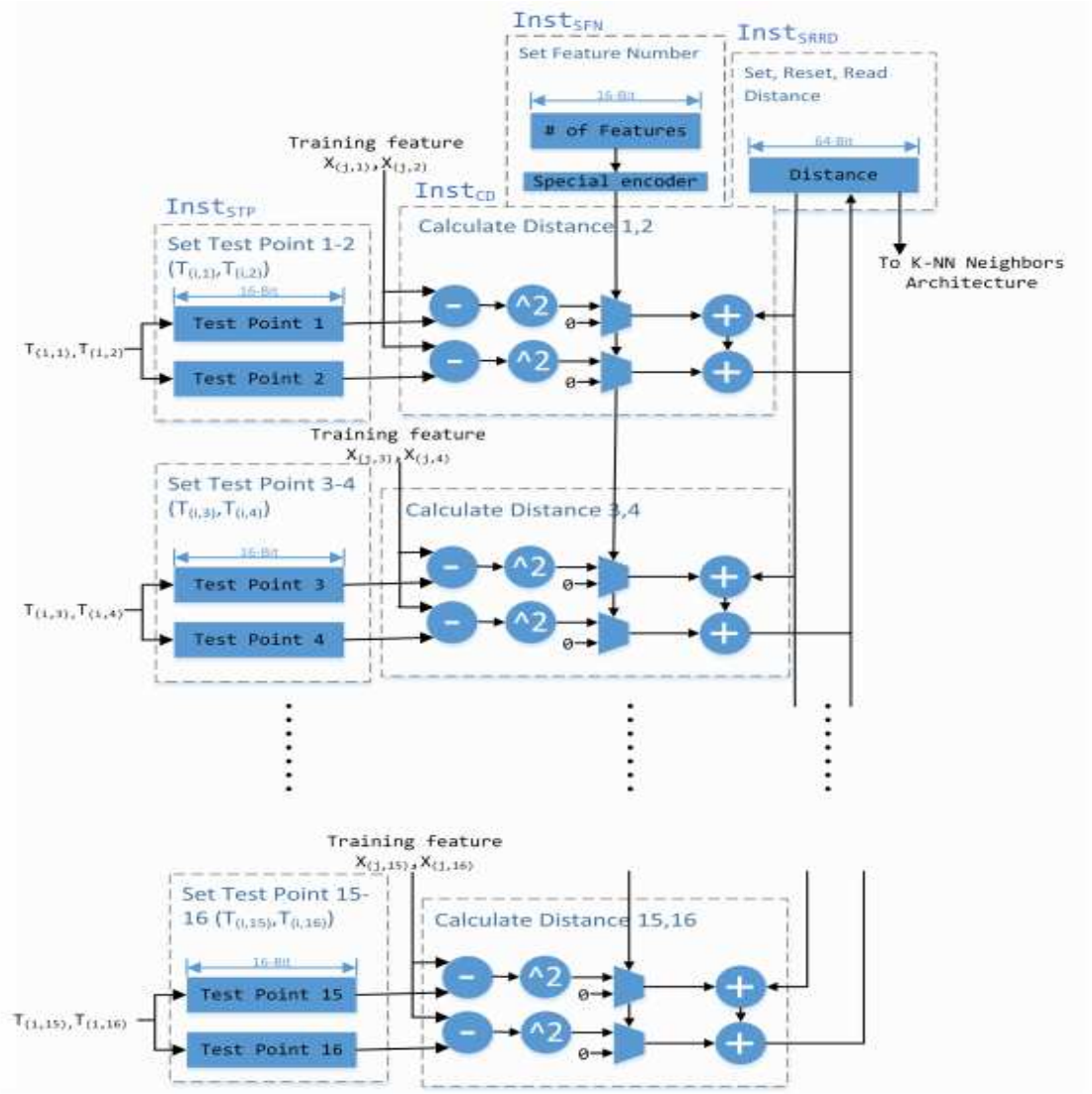

Figure 5.1: Distance Calculation Module

modules/instructions ($Inst_{STP}$, $Inst_{CD}$, $Inst_{SFN}$ and $Inst_{SRRD}$). Each sub-module

is responsible for performing a specific task(s) within the Calculate_Distance() module. The following steps take place in the execution of the distance calculation:

i Instruction $Inst_{SFN}$ is used to set the number of features in the current dataset under investigation. This is stored in a 16-bit register which is passed to a special encoder that enables the necessary "Calculate Distance" sub-modules required for the computation. For example, if the number of features = 3, the top two "Calculate Distance" sub-modules 1,2 and 3,4 will be enabled while all others are disabled. However, if the number of features within a dataset exceeds 32, which is the upper limit of the registers accommodating the features, then uploading the features will take multiple cycles. This instruction is executed only once at the start of the entire K-NN algorithm.

ii Each feature of the test point, T_i, is stored in a 16-bit register in the $Inst_{STP}$ module. Based on a bus of size 32-bits within the architecture, only two features (16-bits each) can be stored at a time. For example, if a test point has only three features, they will be stored in registers "Test Point 1" up to "Test Point 3".

iii For each record in the training set, the following steps are applied:

- The $Inst_{SRRD}$ module resets the distance register to zero.
- The features of the training record X_j are applied to the $Inst_{CD}$ module. This module is responsible for carrying out the basic Euclidean distance calculation.
- When the final distance has been calculated, the system forwards this value to the K-NN Neighbors Architecture, which is explained next.

5.1.3 The "K-Closest Neighbors" Module

Figure 5.2 shows the overall architecture of the module responsible for calculating the K-closest neighbors of the testing point. The value of the distance calculated by the "Distance Calculation" module (i.e., the distance between the test point, T_i and the training point X_j) is used to identify the *K*=10 closest neighbors. The following are the steps taken by the "K-Closest Neighbors" module:

i The value stored in the *Neighbor Registers* 1-10 are set to a maximum (high) value for every new test point, T_i.

ii For each training point, X_j:

- The distance from the testing point, T_i is propagated towards the K-Closest Neighbors Module.
- The comparator in the first sub-module determines if the new distance should occupy Register Neighbor #1 or propagate further down the register file. This is based on the value of the distance stored in Register Neighbor #1.
- The comparison operation is repeated for all values stored in the Neighbor Registers (#2 to #10).

iii After all distances from the training set to the testing point have been calculated, the shortest distances will occupy the Neighbor Registers. In other words, the shortest distance (along with its associated class) will be stored in Neighbor Register #1 while the 10th shortest distance will be stored in Neighbor Register #10.

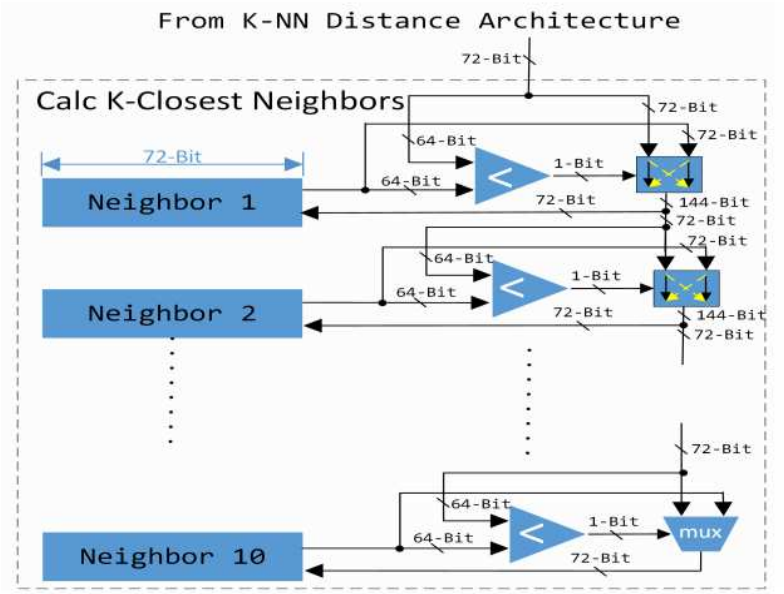

Figure 5.2: K-Closest Neighbors Module

5.1.4 The "Most Frequent Class" Module

Following the identification of the *K*=10 closest neighbors by the K-Closest Neighbors module, these values are available and stored in the registers 'Neighbor#1-#10'. It is important to notice that the width of these registers are 72-bits (64-bits reserved for the distance and 8-bits for the Class); only the Class (8-bits of this register) will be used for the majority calculation. The main role of the Most Frequent Class Module (shown in Figure 5.3) is to identify the winning class; i.e., the class with the highest majority among the 10 candidates. The main steps taken by this module are as follows:

i The 8-bit (class) contents of register 'Neighbor#1', which contains the closest distance to the test point T_i, are compared to the remaining classes stored in the other registers.

ii Based on this comparison, the number of matching classes in the other registers

will increment the count of this class.

iii This operation is repeated for all other registers.

iv The comparators connected to the accumulators then identify the winning class.

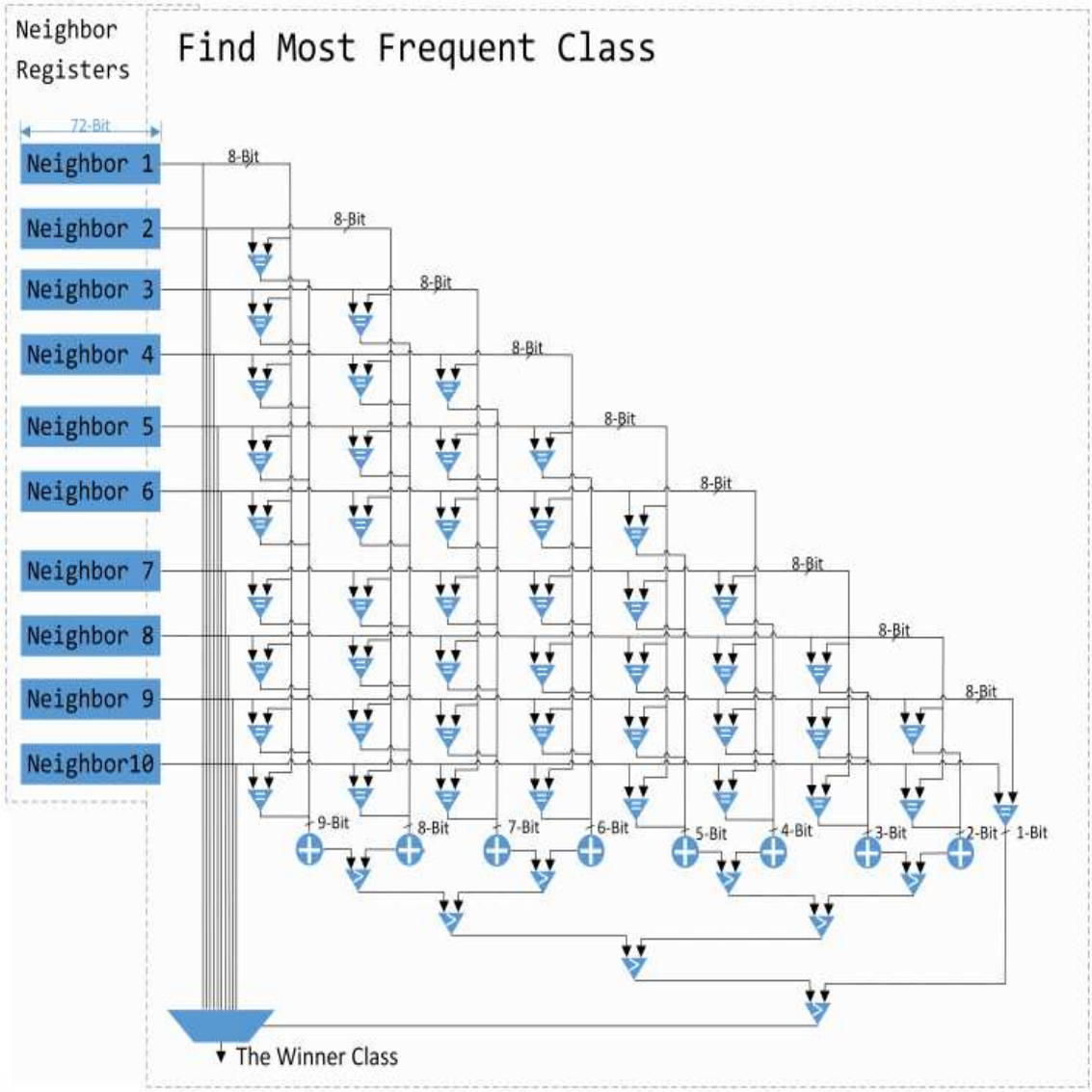

Figure 5.3: Most Frequent Class Module

5.1.5 Alternative Bus Architectures

The term “TIE32” is assigned to the implementation of all the previously described architectures. In addition to the “TIE 32” implementation, two more architectures were introduced (i.e., TIE128 and TIE512). The main difference between the TIE32, TIE128 and TIE512 architectures is in the width of the data bus that is used to transport features, distances, and results. By increasing the data bus width, the number of the distance modules involved in the computation can be increased, thus enhancing the overall performance. TIE32 can issue one distance instruction (two features at a time), while TIE128 can issue four distance instructions in parallel (eight features at a time). The final enhanced architecture, TIE512, can issue sixteen distance instructions in parallel (thirty two features at a time). Several other improvements have been performed on the TIE512 architecture to further enhance execution time and accommodate various data set feature sizes.

5.2 Semi-Tightly Coupled Architecture

In addition to the previous (tightly coupled) hardware accelerators developed using the Tensilica Software Development Toolkit, Xilinx Vivado HLS was used to explore (semi-tightly coupled) hardware accelerators. The Xilinx HLS tool transforms a C specification into an RTL co-processor, which can be exported as an IP block, and later imported and used in different applications.

The creation of specific high performance hardware implementations is achieved by controlling the C synthesis process through optimization directives (as explained in Chapter 2). The latter allows design exploration by quickly creating many different im-

plementations (solutions) from the C source code of the application, thus improving the chance of finding a suitable implementation that balances the performance with the resources used.

The main goal of this part of the thesis is defined based on the following objectives:

1. To identify the interaction between Vivado HLS directives, and to explore different hardware implementations of the K-NN algorithm. Each implementation will result in a pure RTL implementation and seek to decrease the latency and run- time of the K-NN algorithm.

2. To map six custom architectures of the K-NN algorithm for six different benchmarks on the FPGA. Each architecture represents the best implementation from the design exploration in Step 1. Moreover, the running times, resources used and power consumption are reported for each mapped architecture.

3. To design a general architecture that is capable of handling any benchmark. This general architecture will be mapped on the FPGA using different interconnect interface that connect the developed hardware accelerator with the ARM processor in a H/S-Co-Design approach. The benefit of the latter framework is to study the efficiency and performance of the interfaces on the run-time, resources and power consumption.

5.2.1 Design Exploration of the K-NN

HLS Design Steps

In the Vivado HLS design flow, the user can specify any sub-function below "main( )" as the top level function for synthesis. Since "main( )" cannot be synthesized by the HLS tool, it is used as a test bench as described below. The inputs to Vivado HLS are:

- A "C" function, usually called a Top function, which may contain a hierarchy of sub-functions that is written in C or C++. This function is synthesized into an IP block, which is used as a hardware accelerator.
- A "C" test bench "main( )" and the associated files, which are used by Vivado HLS to simulate the C function prior to synthesis and to verify the RTL output using C/RTL Co-simulation.

The various hardware accelerators for K-NN in this work are implemented on the Zynq-7000 platform. The latter is a hybrid System-on-Chip (SoC) computing platform that supports efficient communication between an ARM processor and a co-processor implemented on the XC7Z020 FPGA. The original K-NN code is split into a top function and a test bench. The code in the test bench runs on the ARM processor of the Zync FPGA (ZedBoard), while the code of the top function is converted into RTL (i.e., an IP) and the bitstream is transferred as a hardware implementation onto the reconfigurable logic of the FPGA.

The datasets (training and testing sets) are defined as 2D global arrays, while the class label is added as an additional column to the array. In addition, the datasets are passed to the top function as memory ports from the external memory. The main outputs from Vi-

vado HLS include the RTL implementation files in hardware description language (HDL) standard formats (VHDL (IEEE 1076-2000) or Verilog (IEEE 1364-2001)). Vivado HLS packages the implementation files as an IP block for use with other tools in the Xilinx design flow which can be transformed into an FPGA bitstream.

Architecture Synthesis

When performing high-level synthesis, Vivado HLS performs the following (high-level) steps:

- Synthesizes Top-Level Function arguments into RTL input and output ports.
- Synthesizes C functions into blocks in the RTL hierarchy. The final RTL design will have a hierarchy of modules which have a one-to-one correspondence with the original C function hierarchy.
- Arrays in the C code are synthesized to block-RAM in the final FPGA design.

HLS Metrics

The three most important metrics when evaluating the quality and performance of the hardware implementation produced by Vivado HLS are: (i) Area (BRAMs, DSP48s, FFs, LUTs), (ii) Latency in clock cycles, and (iii) Throughput.

The three metrics seek to define the efficiency and the performance of the created IP cores. The resources, latency and throughput of the interfaces (communication) between the IP core and the ARM processor is not part of the HLS results. Accordingly, the metrics provided by HLS are considered to be an estimation.

More accurate timing and resource consumption, and estimated power consumption, can be obtained after placement and routing using the Xilinx Vivado tool. In addition, more accurate measurements can be obtained after downloading the bitstream and mapping the design onto the FPGA. In this thesis, all mentioned simulation/pre-synthesis results of the HLS tools will be presented for the K-NN algorithm. These results will be verified by mapping the K-NN architectures into FPGA and producing the post-synthesis results.

Applying HLS Directives

Design exploration for the K-NN algorithm in Vivado HLS's is applied in two different steps. Initially, the effects of applying individual directives are considered by utilizing and applying the following directives to all functions in the code:

- **Loop Unroll** directive (**A**). This directive is applied to all functions and loops of the 1 in Chapter 2.
- **Loop/Function Pipeline** directive (**B**). This directive is applied to all functions and loops of the 1 in Chapter 2.
- **DataFlow** directive (**C**). This directive is applied to the main K-NN Classifier function that includes steps 1.1-1.8 of 1 in Chapter 2.
- **Function Inline** directive (**D**). This directive is applied to all functions and loops of the 1 in Chapter 2.
- **Array_Partitioning** directive (**E**). This directive is applied to the input arrays X and T presented in 1, as shown in Chapter 2.

The initial objective is to apply each directive individually and then report the effects on the performance and resources used. Next, the effect of combining a subset of these directives is investigated.

The array partitioning directive (E) restructures an array into multiple arrays in order to allow parallel access to data when an interface does not normally allow it. The directive will take an array and split it into (*sf*) smaller arrays, where (*sf*) is the splitting factor provided to the directive. In the case of the K-NN algorithm, (*sf*) is selected such that given *M* training instances and *N* features, a total of *N* BlockRam interfaces are created, each containing an M x 1 array of training points for a particular (single) feature. In other words, each feature is assigned to its own BRAM along with its training points.

5.2.2 K-NN Custom Architectures

The baseline and best implementations resulting from the design exploration of the six data sets explained in the previous section are selected to be exported to the Xilinx Vivado tool as IP blocks. The AXI_BRAM interface is used for transferring data to each custom IP core. In total, 12 custom designs (6 baselines and 6 best implementations) are mapped to the Xilinx Zynq FPGA. The best implementation was produced when the directives (partitioning, pipelining, unrolling and inline) are applied to the K-NN design. By applying a partitioning directive with a factor equal to the number of features for each data set, a number of block RAMs with separate interfaces for each is created. Each BlockRAM saves the data values of one feature (one column of the array). By applying pipeline and unroll directives, a number of processing elements equal to the number of features will be created. Each processing element calculates the distance between the individual features of the training and testing points.

Figure 5.4 shows the block diagram of the K-NN design hardware accelerator, the ARM processor, and the Block RAMs a long with the required interconnects created in Xilinx Vivado tools.

5.2.3 K-NN General Architecture

A more general architecture was designed to handle any data set. The hardware is designed to have up to 90 processing elements. In the case of any data set with a different number of features, dummy features are added to the data in the general architecture. A design exploration of the general architecture was carried out similar to the custom architecture approach. The baseline and the best implementation are selected and mapped onto the FPGA. The best implementation is obtained when applying the directives (partitioning, pipelining, unrolling and inlining) on the K-NN design, as will be shown in the next chapter. All twelve data sets of Table 4.1 in Chapter 4 are used to verify the implementation of the general architecture design.

5.2.4 Communication Interconnect

The Xilinx HLS tool estimates the running time and the resources consumed without adding the effects of the communication overhead between the ARM processor and the co-processors while providing the hardware design with data. In general, the size of most data is too big to fit in the existing FPGA; therefore, data needs to be stored in external storage devices, and the communication overhead has to be accounted for. As expected, some interconnects are faster than others. Selecting the most suitable interconnect is an important task to obtain the overall best speedup. Therefore, this part of the thesis

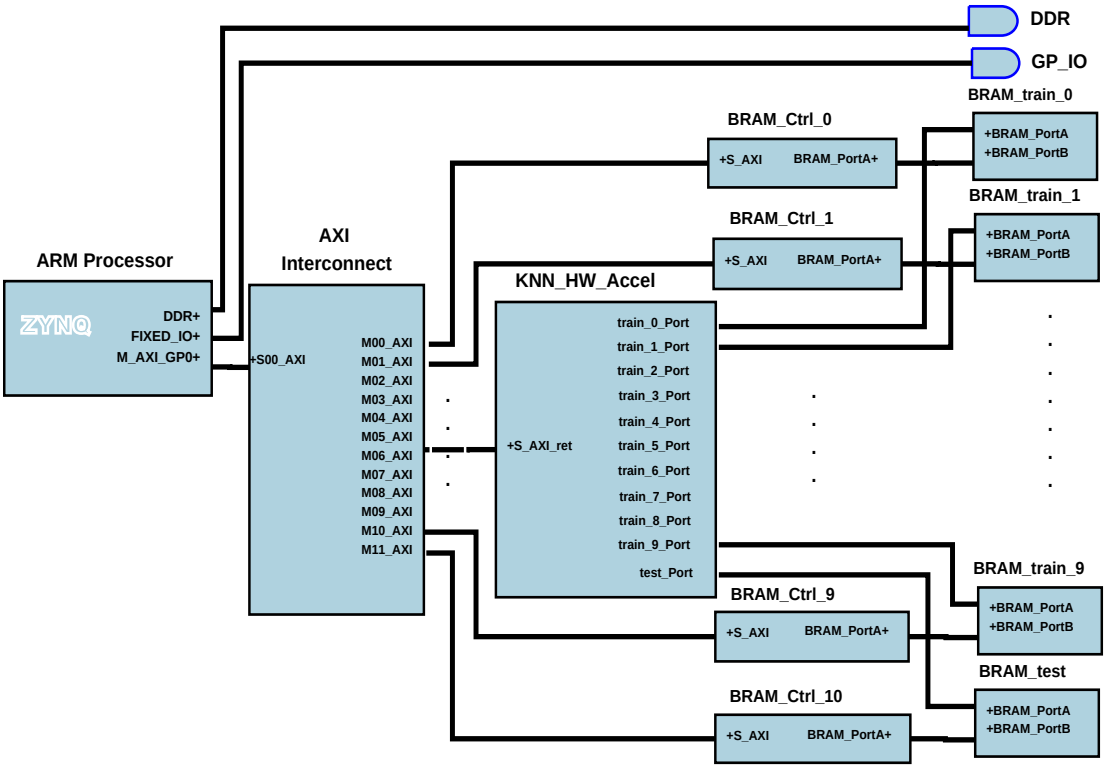

Figure 5.4: K-NN Design in HLS

explores the effects of four different communication schemes for transferring data from the ARM processor to the dedicated hardware accelerator. The four interconnection methods are:

1. **AXI_Light interconnect** is shown in Figure 5.5. The data is transferred serially between the DDR3 DRAM external memory controlled by the DRAM controller through the master AXI General purpose (AXI_GP) port and the slave port of the general AXI interconnect logic core. The latter passes data between its AXI master port and the K-NN AXI_Light slave port. The width of this data port is 32-64 bits. Each record (91 x 32-bit) of testing data is sent through this port once for each run, and all training records are sent sequentially to the IP core, one record for each run. The K-NN algorithm is applied, then the misclassification error is passed back to the ARM processor for each training point through AXI_Light port. When this type of interconnect is used, the IP core has to wait for data to be passed sequentially. The recorded execution time includes both the time consumed while passing data and the time consumed by the IP core.

2. **AXI_BRAM interconnect**, shown in Figure 5.6. The data is transferred between the DDR3 DRAM external memory controlled by the DRAM controller through the master AXI General purpose (AXI_GP) port and the slave port of the general AXI interconnect logic core. The latter passes data between its AXI master port and the AXI_BRAM slave port. Data is stored in BRAMs before passing it to/from the K-NN AXI_BRAM port. Single port BRAMs are used in this work. The width of the AXI_BRAM port ranges from 32-1024 bits. In general architecture, the data are stored into internal BRAMs before being utilized by the K-NN IP core. The

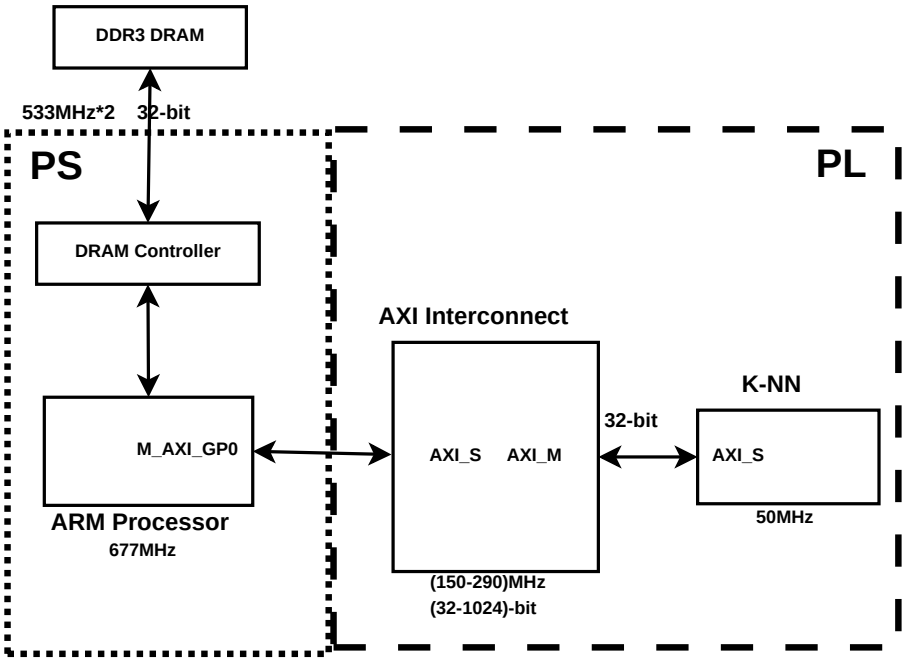

Figure 5.5: AXI Light Interconnect

recorded execution time includes both the time consumed while passing data and the time consumed by the IP core.

3. **Stream interconnect with HP port**, shown in Figure 5.7. The data is streamed between the DDR3 DRAM external memory and the master port of the AXI memory interconnect logic core through the AXI High Performance (AXI_HP) port. The AXI memory interconnect streams data between its AXI slave port and the K-NN slave port through the Direct Memory Accesses (DMA) logic core. The DMA can pass data of width ranges from 32-1024 bits. The recorded execution time includes both the time consumed while passing data and the time consumed by the IP core.

4. **Stream interconnect with ACP port**, shown in Figure 5.8. The data is streamed between the L2 cache memory of the ARM dual-core Cortex-9 processor and the

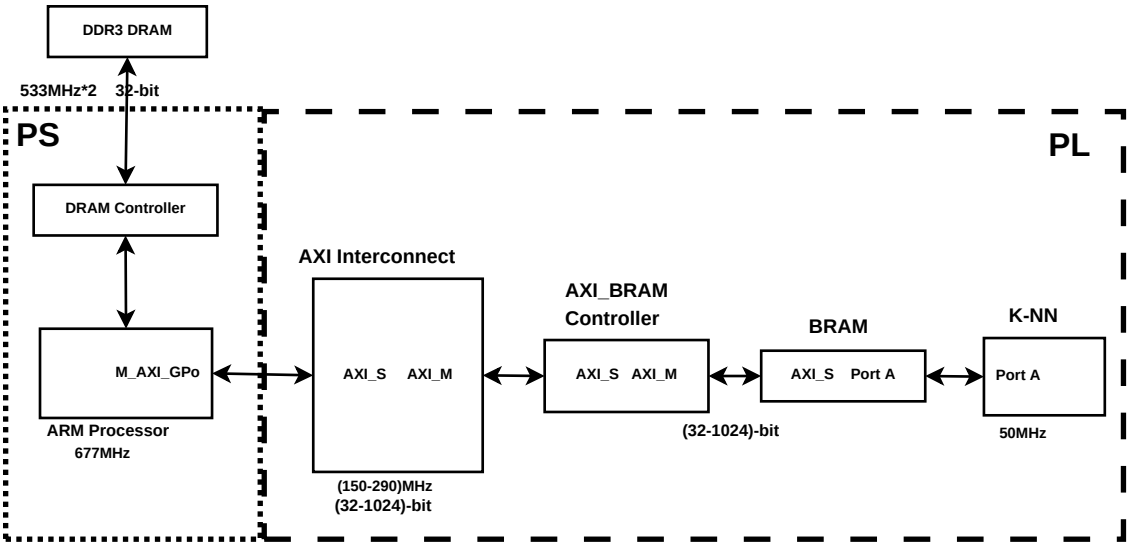

Figure 5.6: AXI BRAM Interconnect

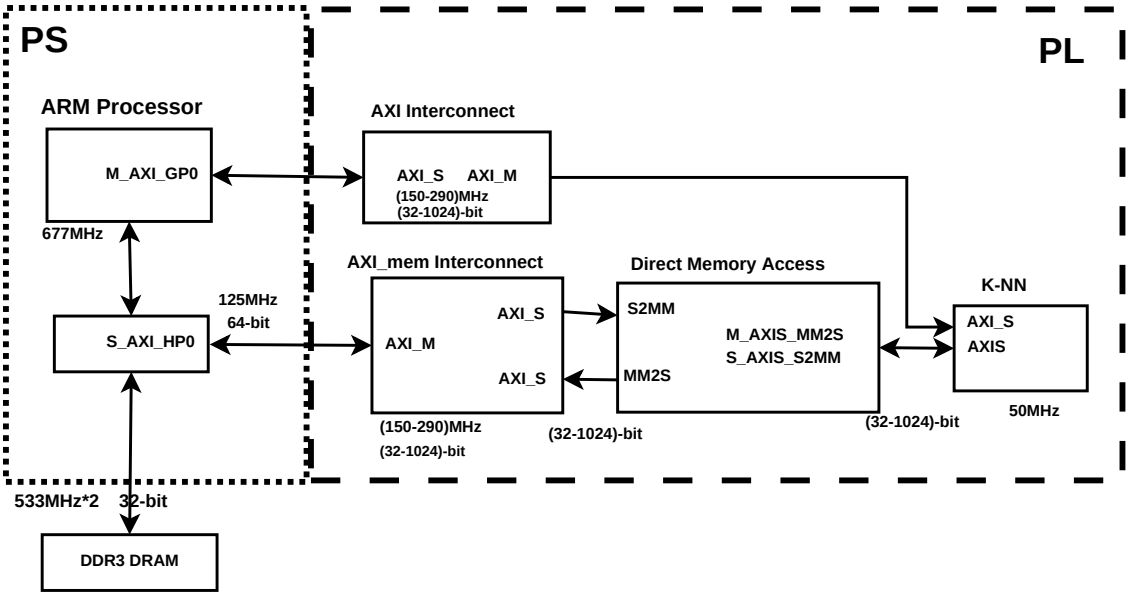

Figure 5.7: Stream HP Interconnect

master port of the AXI memory interconnect logic core through the AXI Accelerator Coherency Port (AXI_ACP) port. The AXI memory interconnect streams data between its AXI slave port and the K-NN slave port through the Direct Mem-

ory Accesses (DMA) logic core. The DMA can pass data of width ranges from 32-1024 bits. The recorded execution time includes both the time consumed while passing data and the time consumed by the IP core.

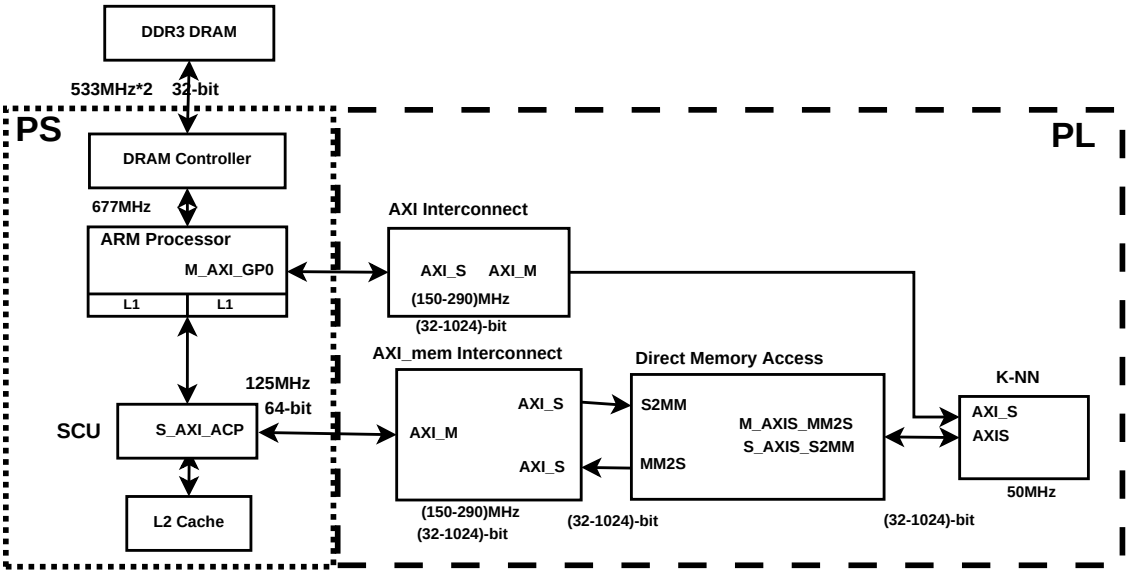

Figure 5.8: Stream ACP Interconnect

.

5.3 Summary

This chapter presented the design and implementations of tightly coupled based hardware accelerators and semi-tightly coupled based designs for the K-NN algorithm using Tensilica and Xilinx Vivado tools. The next chapter compares and evaluate designs in terms of performance, resources used and power consumption.

Chapter 6

Results and Analysis

In this chapter, results based on the tightly coupled and semi-tightly coupled architectures described in Chapter 5 are presented. These results are analyzed and discussed in detail.

6.1 Experimental Setup

The design exploration of the K-NN algorithm is carried out using Xilinx Vivado HLS 2015.1 tool running on an Intel®Core™i7 CPU 920 @ 2.67GHz. The AVNET Zed-Board (accommodating a Zync FPGA XC7Z020) is used to implement and map all hardware accelerators created by the Vivado HLS tool. The ARM dual-core Cortex-9 MPCore processor is used in the Zynq FPGA board with a 512 MB DDR3 on-chip memory. To determine the improvements in runtime (which is the main objective), a baseline of the original C code without any optimizations was implemented. The tools apply some optimization directives by default; therefore, these directives are turned off.

The design exploration for the ASIP approach is based on Xtensa Xplorer, Xtensa

Development Environment Version 6.0.2, running under Windows 7, 64-bit on an Intel Xeon 3.07 GHz processor with 6 GB of RAM.

6.2 Results of Semi-Tightly Coupled Architectures

6.2.1 Design Exploration of Custom Architectures

The Vivado HLS provides optimization directives or pragmas that can be applied to the synthesizable part of the code in the top function to implement hardware exploration of the design. Basically, three main types of optimizations that were applied; performance driven, resource driven and interface driven directives that are available in the Vivado HLS tool. To determine the improvements in runtime (which is the main objective), a baseline of the original C code without any optimizations was implemented. Tables 6.1,

Directive	CLK CYC	DSPs	FFs	LUTs	Speedup
Baseline	65,482	1	1,354	1,652	1.0
A(Unroll)	17,162	9	3,327	5,628	3.8
B(Pipeline)	16,252	9	3,254	5,642	4.0
C(Dataflow)	65,942	1	1,312	1,475	1.0
D(Inline)	63,712	1	1,407	1,786	1.0
E(Part)	55,482	1	4,170	6,868	1.2
AE	17,162	9	3,391	5,208	3.8
BE	15,352	9	3,265	5,219	4.3
CE	55,512	1	4,211	6,972	1.2
DE	55,512	1	5,480	13,632	1.2
ABE	15,352	9	3,265	5,219	4.3
ACE	16,262	9	3,331	5,206	4.0
ADE	9,142	9	3,497	5,174	7.2
BCE	12,682	9	3,243	5,248	5.2
BDE	9,142	9	3,497	5,174	7.2
CDE	55,512	1	5,454	13,666	1.2
ABCE	12,682	9	3,243	5,248	5.2
ABDE	9,142	9	3,497	5,174	7.2
ACDE	9,142	9	3,377	5,175	7.2
BCDE	1,132	9	3,372	5,717	57.8
ABCDE	1,132	9	3,372	5,717	57.8

Table 6.1: HLS: Design Exploration for BCW_9 Benchmark

6.2, 6.3, 6.4, 6.5 and 6.6 present the detailed results of the design exploration applied to the BCW_9 benchmark, the BCW_30 benchmark, the Spectf benchmark, the Spambase benchmark, the Mfeat benchmark and the RobotFailure benchmark, respectively.

Directives	CLK CYC	DSPs	FFs	LUTs	Speedup
Baseline	141,082	1	1,424	1,657	1.0
A(Unroll)	36,062	30	4,757	8,455	3.9
B(Pipeline)	36,052	30	4,832	8,492	3.9
C(Dataflow)	141,542	1	1,382	1,480	1.0
D(Inline)	139,312	1	1,443	1,790	1.0
E(Part)	149,982	1	4,195	6,923	0.9
AE	36,062	30	4,893	4,893	3.9
BE	35,152	30	4,746	7,241	4.0
CE	150,012	1	4,233	7,027	0.9
DE	131,112	1	7,372	13,776	1.1
ABE	35,152	30	4,746	7,241	4.0
ACE	36,062	30	4,997	7,127	3.9
ADE	9,062	30	4,737	5,841	15.6
BCE	32,482	30	4,756	7,189	4.3
BDE	9,062	30	4,737	5,841	15.6
CDE	131,112	1	7,346	13,810	1.1
ABCE	32,482	30	4,756	7,189	4.3
ABDE	9,062	30	4,737	5,841	15.6
ACDE	10,862	30	4,976	5,779	13.0
BCDE	1,362	30	5,411	7,755	103.6
ABCDE	1,362	30	5,411	7,755	103.6

Table 6.2: HLS: Design Exploration for BCW_30 Benchmark

Directives	CLK CYC	DSPs	FFs	LUTs	Speedup
Baseline	193,282	5	1,496	1,728	1.0
A(Unroll)	51,362	48	5,641	10,498	3.8
B(Pipeline)	52,252	48	6,403	10,601	3.8
C(Dataflow)	193,742	5	1,454	1,551	1.0
D(Inline)	193,312	5	1,625	1,901	1.0
E(Part)	214,782	5	4,232	7,086	0.9
AE	51,362	46	5,970	8,621	3.8
BE	50,452	46	5,642	8,706	3.8
CE	214,812	3	4,273	7,190	0.9
DE	181,512	1	6,718	13,907	1.1
ABE	50,452	46	5,642	8,706	3.8
ACE	51,362	46	6,039	8,488	3.8
ADE	9,492	44	6,520	8,549	20.4
BCE	46,882	46	5,745	8,598	4.1
BDE	9,492	44	6,520	8,549	20.4
CDE	181,512	1	6,692	13,941	1.1
ABCE	46,882	46	5,745	8,598	4.1
ABDE	9,492	44	6,520	8,549	20.4
ACDE	12,192	44	6,829	8,416	15.9
BCDE	1,512	44	6,688	9,048	127.8
ABCDE	1,512	44	6,688	9,048	127.8

Table 6.3: HLS: Design Exploration for Spectf Benchmark

Directives	CLK CYC	DSPs	FFs	LUTs	Speedup
Baseline	240082	4	1,495	1,534	1.0
A(Unroll)	63062	61	6,592	12,315	3.8
B(Pipeline)	63952	61	7,575	1,2433	3.8
C(Dataflow)	240542	5	1,454	1,551	1.0
D(Inline)	240112	5	1,625	1,901	1.0
E(Part)	273282	3	4,232	7,240	0.9
AE	63062	59	6,901	9,895	3.8
BE	62152	59	6,606	10,006	3.9
CE	273312	3	4,273	7,344	0.9
DE	228312	1	7,134	14,035	1.1
ABE	62152	59	6,606	10,006	3.9
ACE	63062	59	6,953	9,736	3.8
ADE	9,622	57	7,586	9,812	25.0
BCE	59,482	59	6,548	9,878	4.0
BDE	9,622	57	7,586	9,812	25.0
CDE	228,312	1	7,108	14,069	1.1
ABCE	59482	59	6,548	9,878	4.0
ABDE	9622	57	7,586	9,812	25.0
ACDE	12,322	57	8,001	9,660	19.5
BCDE	1,622	57	6,911	10,231	148.0
ABCDE	1,622	57	6,911	10,231	148.0

Table 6.4: HLS: Design Exploration for Spambase Benchmark

Directives	CLK CYC	DSPs	FFs	LUTs	Speedup
Baseline	311,582	5	1,500	1,544	1.0
A(Unroll)	81,062	80	8,054	14,986	3.8
B(Pipeline)	81,052	80	9,491	15,177	3.8
C(Dataflow)	311,582	5	1,474	1,578	1.0
D(Inline)	311,582	5	1,466	1,519	1.0
E(Part)	359,772	3	4,854	8,446	0.9
AE	81,062	78	8,470	11,765	3.8
BE	79,252	78	8,217	11,942	3.9
CE	359,772	3	4,546	8,660	0.9
DE	299,672	1	9,205	17,649	1.0
ABE	79,252	78	8,217	11,942	3.9
ACE	80,162	78	8,308	11,566	3.9
ADE	10,712	76	9,588	11,663	29.1
BCE	75,682	78	7,962	11,767	4.1
BDE	10,712	76	9,588	11,663	29.1
CDE	299,672	1	9,179	17,683	1.0
ABCE	75,682	78	7,962	11,767	4.1
ABDE	10,712	76	9,588	11,663	29.1
ACDE	13,412	76	9,645	11,478	23.2
BCDE	1842	76	9,467	12,029	169.2
ABCDE	1842	76	9,467	12,029	169.2

Table 6.5: HLS: Design Exploration for Mfeat Benchmark

Directives	CLK CYC	DSPs	FFs	LUTs	Speedup
Baseline	360,332	5	1,495	1,534	1.0
A(Unroll)	93,682	94	8,822	17,744	3.8
B(Pipeline)	93,662	94	11,046	17,956	3.8
C(Dataflow)	360,332	5	1,469	1,568	1.0
D(Inline)	356,732	1	1,441	1,594	1.0
E(Part)	422,672	3	4,572	8,626	1.0
AE	93,682	92	9,597	13,961	3.8
BE	91,862	92	9,484	14,173	3.9
CE	422,672	3	4,546	8,660	1.0
DE	347,972	1	8,685	15,344	1.0
ABE	91,862	92	9,484	14,173	4.0
ACE	92,782	92	9,342	13,722	4.0
ADE	10,862	90	10,788	13,884	33.2
BCE	89,192	92	8,926	13,958	4.0
BDE	10,862	90	10,788	13,884	33.2
CDE	347,972	1	8,659	15,378	1.0
ABCE	89,192	92	8,926	13,958	4.0
ABDE	10,862	90	10,788	1,3884	33.2
ACDE	13,562	90	10,947	13,645	26.6
BCDE	1,992	90	10,719	14,203	180.9
ABCDE	1,992	90	10,719	14,203	180.9

Table 6.6: HLS: Design Exploration for RobotF Benchmark

Individual directives (Unrolling, Pipelining, Dataflow, Inline and Partitioning) are

first applied, followed by combinations of two, three and four directives. It is clear from these six tables that Vivado HLS allows for extensive design exploration by integrating one or more directives to produce a hardware accelerator. The best speedup is achieved when directive (”B” Pipelining) is applied along with directives (”C” Dataflow, ”D” Inline and ”E” Partitioning). Based on the results achieved in the tables above for the design exploration implementation in HLS, the following can be deduced:

- When a multidimensional loop with higher hierarchy is pipelined (B), all nested loops, except for the outside loop, are unrolled (A) automatically no matter what their directives are. In the other words, when applying pipeline (B) to all loops in the K-NN code, only the outer loop is pipelined and the others are unrolled.

- When applying the unroll (A) and/or pipeline (B) directives, any local arrays are partitioned automatically by the HLS tool. This behaviour decreases the runtime and increases the resources used. It is deduced that the unroll (A) and pipeline (B) directives cannot be applied individually in the code due to the automatic optimization behavior by the tools.

- By applying the unroll (A) directive, there is an enhancement in performance over the baseline, and an increase in consumed resources as an effect of the multiple computing elements that are created by the unroll directive. However, this improvement is not producing the anticipated results since the maximum communication capacity between the BRAMs and the multiple processing elements in Vivado HLS tools is limited by the dual port connection.

- It is noticed that whenever the pipeline (B) directive is applied, it produces the best performance with an increase in the resources used. Based on previous analysis,

the applied pipeline (B) directive to the K-NN code is converted by the automatic optimization behavior of the Vivado HLS tool into applying multiple directives, which are the pipeline (B), the unroll (A) and the partitioning of all local arrays (E). In other words, each time the pipeline directive is applied, the three aforementioned directives are inherently applied.

- The dataflow (C) directive does not show any improvement in performance when it is applied individually, and it results in a noticeable increase in the resources consumed. This increase in resources is the result of synthesizing the required circuit for applying the pipelining behavior between functions (recall that the Dataflow directive goal is to pass partial results between two functions).

- By applying the inline directive (D), almost no effect can be noticed in the performance and the resources consumed. By applying the inline function, the resources of the inlined function are duplicated whenever the function is called.

- By applying the array_partitioning directive (E), the amount of resources used increases with a slight improvement in performance. As arrays are partitioned, multiple ports are used to pass data from the memory to the processing elements. When data arrays are partitioned without applying the unroll or the pipeline directive, the data passes in parallel from memory to individual execution units, which neglects the advantages of using the parallel ports.

- Better results are achieved by combining the partitioning directive (E) with the unroll and pipeline directives since the bottleneck communication is resolved, by passing the data through parallel ports. Accordingly, the array partitioning direc-

tive is included in the design exploration.

- The maximum speedup was achieved by applying the BCDE directives, which practically includes a combination of the unroll (A) (because of the automatic optimization behavior mentioned before), pipelining (B), dataflow (C), inline (D) and array partitioning (E) directives. Applying this combination of optimization directives produces an IP that can benefit from extensive parallelism features with less communication overhead, as shown in Figure 6.1.

6.2.2 Custom Architectures: Implementation

In this section, the baseline and best custom architectures for each benchmark are mapped onto a Zynq FPGA board. A sample of each benchmark with 100 records is stored in the FPGA BRAMs due to the limitation in the available resources on the FPGA board. The running time, resources used and power consumed are recorded and reported.

The Xilinx SDK tool provides the function **XTime_GetTime()**, which can be used to calculate the clock cycles consumed by the hardware accelerators as follows:

*XTime_GetTime((XTime *)&Xtime_start);)*

program running in hardware . . .

*XTime_GetTime((XTime *)&Xtime_stop);*

Total_Time = Xtime_start - Xtime_stop;

Table 6.7 shows the resources used and estimated power consumed by the hardware accelerators produced by Vivado HLS (pre-synthesis, and post-synthesis) for the six UCI Benchmarks. The power consumption was reported only with post-synthesis results estimated by the Xilinx Vivado EDK tool. The Xilinx Vivado HLS tool does not provide

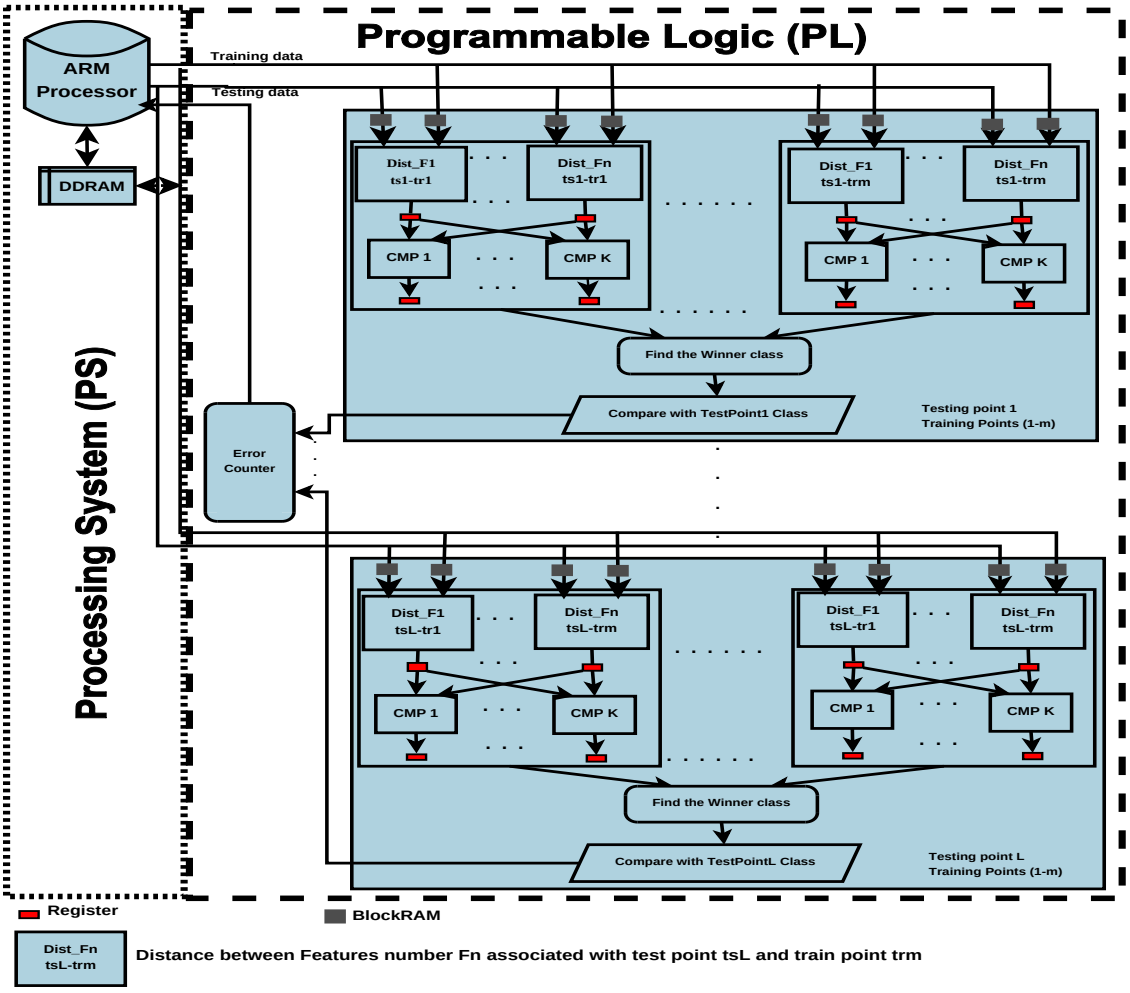

Figure 6.1: The accelerated architecture of the KNN Algorithm by BCDE directives

any estimation for the power consumption. The number of DSP units, along with FFs and LUTs, tends to increase as the number of features and classes increase.

The power consumed by the IPs tends to increase when the number of features increase since the resources used, including DSP units to perform distance calculations, also increase.

Table 6.8 shows the Clock Cycles and speedups achieved by both baseline and best

Method	HLS (Pre Synthesis)			HLS (Post Synthesis)			
Dataset	**DSP**	**FF**	**LUT**	**DSP**	**FF**	**LUT**	**P(mW)**
BCW_9	9	3,372	5,717	9	9,484	8,845	262
BCW_30	30	5,411	7,755	30	19,952	17,998	286
Spectf	44	6,688	9,048	44	26,511	23,880	341
Spambase	57	6,911	10,231	57	32,941	29,677	375
Mfeat	76	9,467	12,029	76	41,681	37,513	447
RobotF	90	10,719	14,203	90	49,997	44,550	510

Table 6.7: Resource and Power Consumption by HLS IPs

selected hardware accelerators. The speedups tend to increase as the number of fea-

Benchmark	HLS-Baseline	HLS-Best	Speedup
BCW_9	65,482	1,132	57.9
BCW_30	141,082	1,362	103.9
Spectf	193,282	1,512	127.8
Spambase	240,082	1,622	148.0
Mfeatr	311,582	1,872	166.4
RobotF	360,332	1,992	180.9

Table 6.8: Clock Cycles/Speedup for KNN HLS IPs

tures increase for the HLS-Best architecture with respect to the HLS-Baseline due to the increase in concurrency (parallel operations performed per clock cycle). Table 6.9 shows the execution time in (ms) and speedups achieved after integrating the IP with the ARM processor on the FPGA. The speedups achieved by the ARM/IP-Best architec-

Benchmark	ARM/IP-BL	ARM/IP-Best	Speedup
BCW_9	12.33	0.27	46.0
BCW_30	29.03	0.29	101.1
Spectf	40.27	0.35	116.6
Spambase	50.42	0.37	134.8
Mfeatr	65.99	0.43	155.3
RobotF	76.52	0.45	169.4

Table 6.9: Execution time(ms)/Speedup for HLS Integrated ARM-IPs

ture in Table 6.9 are less than those achieved by HLS-Best (prior to integration) due to the communication and synchronization overhead between the ARM processor and the hardware accelerator.

6.2.3 Semi-Tightly Coupled General Architecture

In this section, a general architecture that is capable of running all benchmarks with any number of records is designed and implemented for the K-NN algorithm. This design was tested using the 12 popular benchmarks from the UCI Machine learning repository website, shown in Table 4.1. Moreover, the general K-NN architecture is demonstrated and analyzed with different interconnection methods (AXI_BRAM, AXI_Light, Stream using AXI_HP port, and Stream using AXI_ACP port). Preliminary investigation of the efficiency of the same communication interconnects (AXI_BRAM, AXI_Light, Stream using AXI_HP port, and Stream using AXI_ACP port) on the matrix multiplication problem can be found in Appendix B.

Table 6.10 presents results based on the AXI_BRAM interconnect, Table 6.11 presents similar results using the AXI_Light interconnect, Table 6.12 presents similar results using the Stream_HP interconnect, and Table 6.13 presents similar results using the Stream_ACP interconnect. Table 6.14 presents the resources used and the estimated power consumed by a hardware accelerator of general design produced by Vivado HLS (pre-synthesis and post-synthesis) for the different interconnects methods.

Benchmark	SW(ARM)	Baseline	Best(BCDE)	X_SW	X_Base
Robotf	21.57	28.27	5.39	4.0	5.2
HD_Long	32.67	43.74	6.64	4.9	6.6
Spectf	58.44	69.82	8.84	6.6	7.9
HD_Hung	71.80	82.93	9.74	7.4	8.5
HD_Clev	74.44	89.44	10.07	7.4	8.9
BCW30	234.62	299.39	19.25	12.2	15.6
BCW9	341.73	409.75	23.61	14.5	17.4
Credit_g	665.70	790.51	33.60	19.8	23.5
Wine_R	1,745.91	1,884.68	53.84	32.4	35.0
Mfeat	2,685.38	2,979.96	68.25	39.3	43.7
Spambase	6,724.85	15,404.85	175.72	38.3	87.7
Wine_W	8,021.08	17,393.38	189.47	42.3	91.8

Table 6.10: HLS: Execution time (ms) / speedups using AXI_BRAMs interconnects

Benchmark	SW(ARM)	Baseline	Best(BCDE)	X_SW	X_Base
Robotf	21.57	102.91	65.89	0.3	1.6
HD_Long	32.67	166.89	106.71	0.3	1.6
Spectf	58.44	275.81	176.27	0.3	1.6
HD_Hung	71.80	332.18	229.95	0.3	1.4
HD_Clev	74.44	360.19	229.95	0.3	1.6
BCW30	234.62	1,275.82	814.84	0.3	1.6
BCW9	341.73	1,767.09	1,127.22	0.3	1.6
Credit_g	665.70	3,458.51	2,208.25	0.3	1.6
Wine_R	1,745.91	9,603.56	5,346.13	0.3	1.8
Mfeat	2,685.38	13,315.54	8,502.46	0.3	1.6
Spambase	6,724.85	69,861.17	44,580.72	0.2	1.6
Wine_W	8,021.08	78,901.48	50,365.91	0.2	1.6

Table 6.11: HLS:Execution time (ms) / speedups using AXI_Light interconnects

Benchmark	SW(ARM)	Baseline	Best(BDE)	X_SW	X_Base
Robotf	21.57	28.24	4.92	4.4	5.7
HD_Long	32.67	41.28	5.31	6.2	7.8
Spectf	58.44	68.91	7.13	8.2	9.7
HD_Hung	71.80	82.57	7.95	9.0	10.4
HD_Clev	74.44	83.42	7.98	9.3	10.5
BCW30	234.62	296.30	13.15	17.8	22.5
BCW9	341.73	409.75	15.62	21.9	26.2
Credit_g	665.70	729.69	21.38	31.1	34.1
Wine_R	1,745.91	1,842.67	34.99	49.9	52.7
Mfeat	2,685.38	2,919.96	42.45	63.3	68.8
Spambase	6,724.85	15,099.00	62.28	108.0	242.4
Wine_W	8,021.08	15,909.13	63.67	126.0	249.9

Table 6.12: HLS: Execution time (ms) / speedups using stream interconnect with HP port

Based on results reported in Tables 6.10, 6.11 and 6.14 we conclude the following:

- It is noticed from the results of Tables 6.7 and 6.14 that there are differences between the estimated times and resources reported by the Vivado HLS tool (pre-synthesis results) and the time and resources calculated after placement and routing (post-synthesis results). The reason for an inaccurate estimation by the Xilinx HLS tools is not including any estimation of the interconnect logic cores that are required for communication between the Processing system (PS) and Programmable logic (PL) portions of the FPGA board.

- If a fair comparison is made between the custom architectures and the general ar-

Benchmark	SW(ARM)	Baseline	Best(BDE)	X_SW	X_Base
Robotf	21.57	28.24	4.55	4.7	6.2
HD_Long	32.67	41.28	5.00	6.5	8.3
Spectf	58.44	68.88	6,82	8.6	10.1
HD_Hung	71.80	82.46	7.62	9.4	10.8
HD_Clev	74.44	83.38	7.82	9.5	10.7
BCW30	234.62	296.26	13.14	17.9	22.5
BCW9	341.73	409.66	15.62	21.9	26.2
Credit_g	665.70	729.57	21.38	31.1	34.1
Wine_R	1,745.91	1,830.45	34.99	49.9	52.3
Mfeat	2,685.38	2,919.95	42.45	63.3	68.8
Spambase	6,724.85	15,098.69	65.24	103.1	231.4
Wine_W	8,021.08	15,905.31	66.07	121.4	240.7

Table 6.13: HLS: Execution time (ms) / speedups using stream interconnect with ACP port

Method	HLS (Pre Synthesis)			HLS (Post Synthesis)			
Interconnect Type	**DSP**	**FF**	**LUT**	**DSP**	**FF**	**LUT**	**P(mW)**
AXI_Light	90	17,188	25,325	90	9,716	7,457	182
AXI_BRAM	90	10,719	14,203	90	48,145	45,198	548
StramHP	90	10,749	14,927	90	15,284	17,946	253
StreamACP	90	10,749	14,927	90	15,284	17,946	253

Table 6.14: Resource and Power Consumption by HLS IPs

chitecture using AXI_BRAMs interconnects, the general architecture achieves less of an overall speedup compared to the custom architectures. This is expected since the custom design passes data from the internal BRAMs directly to the IP core, while the general architecture passes data from external storage devices. Accordingly, communication overhead between the fast IP core and the external storage devices degraded the speedup by a factor of 4x.

- From the results of Table 6.10, 6.12, and 6.13 it is deduced that when the number of records increases, the amount of achieved speedup increases. This is due to the better utilization of the parallelism feature found in hardware compared to the sequential run in pure software.

- From the results obtained in Tables 6.11, it was noticed that there is no speedup

achieved when an AXI_Light interconnect is used for transferring data between the ARM processor and the K-NN IP. Contrary to the HLS estimated results, the accelerated K-NN is slower than the pure software run. This behavior is due to the slow AXI_Light interface that degrades the overall performance dramatically. Accordingly, the AXI_Light interconnect is not a good selection for transferring data with large size; instead it can be used for transferring signals, such as control signals, between the ARM and the IP core.

- From the results of Tables 6.10, 6.11, 6.12, and 6.13, it is deduced that the streaming interconnects interfaces are the best options for use in the case of transfer data with large size. The best speedup is achieved with the stream interfaces due to their good frequencies and their work protocol comparing with the other interconnect interfaces.

- From the results of Tables 6.12, and 6.13, it is deduced that streaming data with the ACP port produces better speedups than streaming data with the HP port with smaller data size and vice versa. Since the ACP port streams data from cache memory, it's performance is the best with cache hit cases when data size is smaller. The performance of the HP port beats the performance of the ACP port in the case of cache miss with larger data sizes.

- General architecture consumes slightly more resources than those consumed in custom architectures. Accordingly, it can be deduced that by adding dummy features to the data, extra resources are consumed, even though they are not necessarily utilized by the application. This is one of the drawbacks of the general architecture designs.

- General architecture consumes more power than custom architecture. This is expected because when more circuits are used in general architecture, more power is consumed. Moreover, the AXI_BRAM interconnect consumes more power than the AXI_Light because the latter has a lower complex design with only a simple communication interface, while the former has a complicated interconnect circuit.

- The designer has the option to select between a faster architecture that utilizes less resources and power, in the case of a custom design, or a reasonably slower architecture that utilizes more resources and power for more general purposes.

6.3 Tightly Coupled Architectures

Table 6.15 presents the alternative tightly coupled architectures developed for the K-NN algorithm using the Tensilica ASIP tool [3]. The table demonstrates the different fre-

Core Data Bus Size	Max Frequency	Core Size		Core Power
	(MHz)	**(mm^2)**	**(gates)**	**(mW)**
32-Bit	875	0.134	136,217	41.258
128-Bit	768	0.158	158,031	38.747
512-Bit	723	0.222	215,063	42.422

Table 6.15: The cores specification of the ASIP implementation

quencies of operation in MHz, the core sizes in mm^2, the total number of gates required to implement each architecture, along with the estimated power consumption in milliwatts for the three core data bus sizes. Table 6.16 presents the total number of clock cycles used by all benchmarks based on the baseline architectures, TIE32, TIE128 and TIE512 respectively. It is clear that the number of cycles decreases dramatically as the width of the core data bus size increases (i.e., more concurrency), which is expected.

Benchmark	Baseline CLK CYC	TIE 32 CLK CYC	TIE 128 CLK CYC	TIE 512 CLK CYC
BCW_9	3,974,468	127,405	63,997	14,831
BCW_30	9,235,407	745,049	401,775	53,647
Spectf	12,714,641	1,094,468	580,132	332,148
Spambase	15,993,699	1,454,430	770,094	332,148
Mfeat	20,719,655	2,103,662	959,486	446,034
RobotF	24,287,600	2,525,646	1,200,314	446,204

Table 6.16: Clock Cycles for Various ASIP Implementation

Table 6.17 demonstrates the time in milliseconds to run each of the benchmarks based on the various architectures implemented. The overall speedup of the TIE 512 over the baseline architecture is presented in the last column. It is interesting to note that the overall speedup achieved tends to decrease as the size of the datasets increase in terms of features and classes. This is expected since the current architecture used can handle 16 features simultaneously. As the number of features increases beyond 16, the "Distance Calculation" module performs this task in multiple cycles rather than in a single cycle. Table 6.18 demonstrates the time in milliseconds to run each of the benchmarks

Benchmark	Baseline Time(ms)	TIE 32 Time(ms)	TIE 128 Time(ms)	TIE 512 Time(ms)	Overall SpeedUp(x)
BCW_9	4.54	0.15	0.08	0.02	221.4
BCW_30	10.55	0.85	0.52	0.07	142.3
Spectf	14.53	1.25	0.76	0.46	31.6
Spambase	18.28	1.66	1.00	0.46	39.8
Mfeat	23.68	2.40	1.25	0.62	38.4
RobotF	27.76	2.89	1.56	0.62	44.9

Table 6.17: Total Time/Overall Speedup of ASIP Implementations

on the ARM processor, ASIP, and ARM/IP on the Zync FPGA respectively. The last two columns of Table 6.18 show the speedup achieved by the tightly coupled architecture (ASIP) and ARM/IP co-processors over the pure software implementation on the ARM processor. It is interesting to see that the speedup of the Tensilica based ASIP implementations decreases while the speedup achieved by the ARM/IP tends to increase as

the benchmarks increase by feature size. This is attributed to the massive concurrency that can be achieved in Vivado HLS due to loop unrolling/pipelining and coarse function pipelining via dataflow, which exploits the parallelism between loop iterations.

Benchmark	ARM Time(ms)	ASIP Time(ms)	ARM/IP Time(ms)	ASIP Speedup	ARM/IP Speedup
BCW_9	13.01	0.02	0.27	650.5	47.8
BCW_30	29.31	0.07	0.29	418.7	101.1
Spectf	39.68	0.46	0.35	86.3	113.4
Spambase	49.67	0.46	0.37	108.0	134.2
Mfeatr	64.80	0.62	0.43	104.5	150.7
RobotF	75.42	0.62	0.45	121.6	167.6

Table 6.18: Execution time in milliseconds in ARM, ASIP and ARM/IP

6.4 Summary

In this chapter, the results of the semi-tightly coupled architectures implemented in the Tensilica ASIP tool, and the semi-tightly coupled custom and general architectures implemented in the Xilinx HLS tools for the K-NN algorithm are introduced, discussed, analyzed and compared. The achieved speedups are discussed along with the resources used and the power consumed. The next chapter will conclude this thesis, and propose possible future directions of investigation.

Chapter 7

Conclusions and Future Work

This chapter concludes the thesis with a brief summary of the achievements, the highlights, the lessons, and the insights from this thesis, and proposes possible future works to investigate.

7.1 Summary

The K-NN algorithm is one of the most robust, yet simple, machine-learning algorithms available for classification. It can be accelerated using reconfigurable computing platforms, such as FPGAs, due to the structure of the algorithm, which permits the exploitation of parallelism. In this thesis, two different approaches were used to create hardware accelerators for the K-NN machine-learning algorithm based on Xilinx Vivado HLS and Cadence Tensilica. Detailed implementations of the Tensilica ASIP approach were introduced along with alternative bus widths that further enhanced performance. Within the Xilinx Vivado HLS flow, the effect of interaction was explored among the most impor-

tant directives, such as loop unrolling, dataflow, pipelining, inline, and array_partitioning directives. Results obtained indicate that the combination of a subset of these directives was quite efficient. Custom and general architectures of the K-NN classifier algorithm were designed and implemented using AXI_BRAMs interconnects. The general architecture with alternative interface interconnects was further studied.

Results obtained show that there are hardware speedups in an HLS that range from 48x-168x for custom architecture with data sets stored in internal BRAMs. A speedup of 4x-42x was achieved for general architecture with data stored in an external DDRAM and passed to internal BlockRAMs via AXI_BRAMs interconnects. A speedup of 0.1x-0.2x was achieved for general architecture with data stored in an external DDRAM and passed to internal BRAMs via AXI_Light interconnects. Results for tightly coupled architectures obtained using Cadence Tensilica tools were more optimistic with a speedup ranging from 86x-650x.

7.2 Insight

Before selecting the acceleration approach, the designer needs to ask the following questions: Is the algorithm suitable to be mapped into hardware? Does the structure of the algorithm permit it to exploit parallelism? What tools are available to accomplish this task? What are the cost limitations? Where will the accelerated algorithm be utilized (in a custom application or in multiple applications)? Is the main goal acceleration, or reducing the cost, or reducing the resources consumed, or reducing the power consumed? By answering all of the above questions, enables the selection of a better acceleration approach with the most suitable details indicating interconnects.

7.3 Future Work

In this section, extensions to the work in this thesis are suggested, and future directions are proposed. The proposed directions are:

- In this thesis, only the baseline and the best architectures are implemented and mapped to the FPGA board. Other architectures created in the design exploration of the K-NN algorithm can be implemented and mapped to the FPGA board to study the relationship between the pre-synthesis and post-synthesis results.
- The study of the communication interconnects can be expanded to include other interconnects available on the FPGA board.
- The work of the K-NN algorithm can be expanded to other machine-learning algorithms such as artificial neural networks, support vector machines, and naive Bayes.
- The work of the K-NN algorithm can be replicated using the GPU board, and a comparison made of the running time and power consumption obtained from the FPGA and GPU platforms.
- The work in this thesis can be implemented using different tools, such as Xilinx SDAccel (OpenCL). Many comparisons can be made between Vivado HLS and Xilinx SDAccel as design entry tools at different levels of abstraction.
- A smart machine-learning algorithm selector can be designed for the best classification algorithm in hardware. This selector can base its selection factor on

different characteristics such as accuracy, running time, resources and/or power consumption.

Bibliography

[1] M. Kiang, “A Comparative Assessment of Classification Methods,” *Journal of Decision Support Systems*, vol. 35, pp. 441–454, 2003.

[2] X. Documentations. (2012) Zynq-7000 all programmable soc: Concepts, tools, and techniques (ctt).

[3] Tensilica, “Xtensa Instruction Set Architecture (ISA),” Cadence Design Systems, Inc., 2655 Seely Ave. San Jose, CA 95134, Technical Publications, Mar-2015.

[4] T. Documentations. (2011) Tensilica instruction extension (tie) language.

[5] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. San Fracisco, CA: Morgan Kaufmann, 2005.

[6] M. Kiang, “A comparative assessment of classification methods,” *Decis. Support Syst.*, vol. 35, no. 4, pp. 441–454, July 2003.

[7] R. Caruana and A. Niculescu-Mizil, “An empirical comparison of supervised learning algorithms,” in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML ’06. New York, NY, USA: ACM, 2006, pp. 161–168.

[8] S. Kotsiantis, "Supervised machine learning: A review of classification techniques," in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2007, pp. 3–24.

[9] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest nighbor search using gpu," *08. IEEE Computer Society: Computer Vision and Pattern Recognition Workshops*, pp. 1–6, 2008.

[10] J. AlKhateeb, F. Khelifi, J. Jiang, and S. Ipson, "A new approach for off-line handwritten arabic word recognition using knn classifier," *IEE International conference on Signal and Image Processing Applications*, pp. 191–194, 2009.

[11] L. Lu, H. Zhang, and H. Jiang, "Content analysis for audio classification and segmentation," *IEE Transaction on Apeech and Audio Processing*, no. 7, pp. 504–516, 2002.

[12] X. Documentations. (2013) ug902-vivado-high-level-synthesis.

[13] D. Liu, *Embedded DSP processor design: application specific instruction set processors*. Morgan Kaufmann, 2008, vol. 2.

[14] "Xtensa Instruction Set Architecture (ISA)," Tensilica Inc., Santa Clara, CA 95054, Technical Publications, Mar-2012.

[15] B. Longstaff, S. Reddy, and D. Estrin, "Improving activity classification for health applications on mobile devices using active and semi-supervised learning," in *Per-*

vasive Computing Technologies for Healthcare (PervasiveHealth), 2010 4th International Conference on-NO PERMISSIONS, Munich, March 2010, pp. 1–7.

[16] Kaggle. (2011) Stay alert! the ford challenge.

[17] G. Lacey, G. Taylor, and S. Areibi, "Deep learning on FPGAs: Past, present, and future," *arXiv preprint arXiv:1602.04283*, 2016.

[18] D. Zhijie, W. Yong, and T. Xiaoling, "Method of network traffic classification using naïve bayes based on fpga," in *Conference Anthology, IEEE*, China, Jan 2013, pp. 1–3.

[19] Y. Alkabani, M. El-Kharashi, and H. Bedor, "Hardware/software partitioning of a bayesian spam filter via hardware profiling," in *Industrial Electronics, 2006 IEEE International Symposium on*, vol. 4, Montreal, Que, July 2006, pp. 3264–3269.

[20] C. Sriramakrishnan and A. Shanmugam, "Implementation of featureset reduced symmetric transform in image retrieval optimized for fpga," *IJCSE*, vol. 2, pp. 2993–2995, 2010.

[21] M. Marsono, M. El-Kharashi, and F. Gebali, "Binary lns-based naive bayes inference engine for spam control: noise analysis and fpga implementation," *Computers Digital Techniques, IET*, vol. 2, no. 1, pp. 56–62, January 2008.

[22] W. Vanderbauwhede, A. Frolov, L. Azzopardi, S. Chalamalasettil, and M. Margala, "High throughput filtering using fpga-acceleration," in *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, ser. CIKM '13. New York, NY, USA: ACM, 2013, pp. 1245–1248.

[23] A. Savich, M. Moussat, and S. Areibi, "A scalable pipelined architecture for real-time computation of mlp-bp neural networks," *Microprocess. Microsyst.*, vol. 36, no. 2, pp. 138–150, Mar. 2012.

[24] A. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing mlp-bp on fpgas: A study," *Neural Networks, IEEE Transactions on*, vol. 18, no. 1, pp. 240–252, Jan 2007.

[25] A. Yilmaz, B. Erkmen, and O. Yavuz, "Fpga implementation of differential evaluation algorithm for mlp training," in *Innovations in Intelligent Systems and Applications (INISTA) Proceedings, 2014 IEEE International Symposium on*, Alberobello, June 2014, pp. 425–430.

[26] J. Nunez-Perez, J. Cardenas-Valdez, J. G. Aguilar, C. Gontrand, B. Goral, and J. Verdier, "Measure-based modeling and fpga implementation of rf power amplifier using a multi-layer perceptron neural network," in *Electronics, Communications and Computers (CONIELECOMP), 2014 International Conference on*, Cholula, Feb 2014, pp. 237–242.

[27] S. Wang, Y. Peng, G. Zhao, and X. Peng, "Accelerating on-line training of ls-svm with run-time reconfiguration," in *Field-Programmable Technology (FPT), 2011 International Conference on*, New Delhi, Dec 2011, pp. 1–6.

[28] M. Papadonikolakis and C. Bouganis, "Novel cascade fpga accelerator for support vector machines classification," *IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS*, vol. 23, no. 7, pp. 1040–1052, 2012.

[29] C. Liu, F. Qiao, X. Yang, and H. Yang, "Hardware acceleration with pipelined adder for support vector machine classifier," in *Digital Information and Communication Technology and it's Applications (DICTAP), 2014 Fourth International Conference on*. Bangkok: IEEE, 2014, pp. 13–16.

[30] B. Mandal and M. andK. Sarma, "Design of a systolic array based multiplierless support vector machine classifier," in *Signal Processing and Integrated Networks (SPIN), 2014 International Conference on*. Noida: IEEE, 2014, pp. 35–39.

[31] S. Kim, S. Lee, and K. Cho, "Design of high-performance unified circuit for linear and non-linear svm classifications," *Journal of semiconductor technology and science in Korea*, vol. 2, pp. 162– 167, 2012.

[32] M. Chiarandini, "Learning decision trees for the analysis of optimization heuristics," in *Learning and Intelligent Optimization*, ser. Lecture Notes in Computer Science, C. Blum and R. Battiti, Eds. Springer Berlin Heidelberg, 2010, vol. 6073, pp. 208–211.

[33] R. Narayanan, D. Honbo, G. Memik, A. Choudhary, and J. Zambreno, "An fpga implementation of decision tree classification," in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, Nice, April 2007, pp. 1–6.

[34] A. Monemi, R. Zarei, and M. Marsono, "Online netfpga decision tree statistical traffic classifier," *Comput. Commun.*, vol. 36, no. 12, pp. 1329–1340, July 2013.

[35] A. Monemi, R. Zarei, M. Marsono, and M. Khalil-Hani, "Parameterizable decision tree classifier on netfpga," in *Intelligent Informatics*, ser. Advances in Intelligent

Systems and Computing, A. Abraham and S. M. Thampi, Eds. Springer Berlin Heidelberg, 2013, vol. 182, pp. 119–128.

[36] P. Skoda, V. Sruk, and B. Rogina, "Frequency table computation on dataflow architecture," in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on*, Opatija, May 2014, pp. 342–346.

[37] S. Lucas, "A Fast Exact Parallel Implementation of the KNN Pattern Classifier," *IEEE World Congress on Computational Intelligence*, vol. 3, pp. 1867–1872, 1998.

[38] E. Manolakos and I. Stamoulias, "Flexible IP cores for the k-NN classification problem and their FPGA implementation," in *IEEE Int'l Symposium on*, Atlanta, GA, 2010, pp. 1–4.

[39] W. Wolberg. (1991) The breast cancer wisconsin (bcw) diagnostic dataset from uci machine learning data repository.

[40] Y. Chen, Y. Lin, and L. Chang, "A Systolic Algorithm for the k-Nearest Neighbors Problem," *IEEE Transactions on Computers*, vol. 41, no. 1, pp. 013–108, 1992.

[41] T. Schumacher, R. Meiche, P. Kaufmann, E. Lubbers, C. Plessl, and M. Platzner, "A hardware accelerator for k-th nearest neighbor thinning," in *Proceeding of Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, USA, 2008, pp. 245–251.

[42] H. Hussain, K. Benkrid, and H. Seker, "An adaptive implementation of a dynami-

cally reconfigurable k-nearest nighbor classifier on fpga," *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference*, pp. 205– 212, 2012.

[43] M. Tahir and A. Bouridane, "An fpga based coprocessor for cancer classification using nearest neighbour classifier," in *IEEE International Conference on Acoustics, Speech and Signal Processing*. Toulouse: IEEE, 2006, pp. III–III.

[44] Q. Kuang and L. Zhao, "A practical gpu based knn algorithm," in *Second Symposium International Computer Science and Computational Technology(ISCSCT 09)*. Hsngshan, China: Academic Puplisher, 2009, pp. 151–155.

[45] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest nighbor search using gpu," *08. IEEE Computer Society: Computer Vision and Pattern Recognition Workshops*, pp. 1–6, 2008.

[46] M. Gokhale and W. Jiang, "Real-time classification of multimedia traffic using fpga," *proceeding of:International Conference on Field Programmable Logic and Applications, FPL*, pp. 56 – 63, August 31 20102010-September 2, 2010.

[47] S. Qamar and S. Adil, "Comparative analysis of data mining techniques for financial data using parallel processing," in *Proceedings of the 7th International Conference on Frontiers of Information Technology*. New York, NY, USA: ACM, 2009, pp. 25:1–25:6.

[48] M. Du and X. Chen, "Accelerated k-nearest neighbors algorithm based on principal component analysis for text categorization," *Journal of Zhejiang University SCIENCE C*, vol. 14, no. 6, pp. 407–416, 2012.

[49] M. Papadonikolakis, C. Bouganis, and G. Constantinides, "Performance comparison of gpu and fpga architectures for the svm training problem," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, Sydney, NSW, Dec 2009, pp. 388–391.

[50] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with gpus and fpgas," in *Application Specific Processors, 2008. SASP 2008. Symposium on*, Anaheim, CA, June 2008, pp. 101–107.

[51] B. Cope, P. Cheung, W. Luk, and L.Howes, "Performance comparison of graphics processors to reconfigurable logic: A case study," *Computers, IEEE Transactions on*, vol. 59, no. 4, pp. 433–448, April 2010.

[52] K. Pauwels, M. Tomasi, J. D. Alonso, E. Ros, and M. V. Hulle, "A comparison of fpga and gpu for real-time phase-based optical flow, stereo, and local image features," *Computers, IEEE Transactions on*, vol. 61, no. 7, pp. 999–1012, July 2012.

[53] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 2012, pp. 47–56.

[54] Y. Zhang, Y. Shalabi, R. Jain, K. Nagar, and J. Bakos, "Fpga vs. gpu for sparse matrix vector multiply," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, Sydney, NSW, Dec 2009, pp. 255–262.

[55] Y. Pu, J. Peng, L. Huang, and J. Chen, "An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl," in *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 167–170.

[56] M. Sadri, C. Weis, N. Wehn, and L. Benini, "Energy and performance exploration of accelerator coherency port using xilinx zynq," in *Proceedings of the 10th FPGA-world Conference*, ser. FPGAworld '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:8.

[57] J. Silva, V. Sklyarov, and I. Skliarova, "Comparison of on-chip communications in zynq-7000 all programmable systems-on-chip," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 31–34, March 2015.

[58] M. Tahghighi, S. Sinha, and W. Zhang, *Analytical Delay Model for CPU-FPGA Data Paths in Programmable System-on-Chip FPGA*. Cham: Springer International Publishing, 2016, pp. 159–170.

[59] M. Lichman, "UCI machine learning repository," 2013.

[60] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.

Appendix A

Glossary

ACP	: Accelerator Coherency Port
ANN	: Artificial Neural Network
ASIC	: Application-Specific Integrated Circuit
ASIP	: Application Specific Instruction Processor
AXI	: Advanced Extensible Interface
BRAM	: Block Random Access Memory
CPU	: Central Processing Unit
DDR3	: Double Data Rate 3
DFG	: Data Flow Graph
DRAM	: Dynamic Random Access Memory
ESL	: Electronic System Level
FPGA	: Field Programmable Gate Array
GMACs	:Giga Multiply-Accumulate Operations per Second
GP	: General-purpose Port
GPP	: General Purpose Processor

GPU	: Graphics Processing Unit
HLS	: High-Level Synthesis
HP	: High-performance Port
K-NN	: K-Nearest Neighbour
MLP	: Multi-Layer Perceptron
NB	: Naïve Bayesian
NoC	: Network-on-Chip
OCM	: On-Chip-Memory
RTL	: Register Transfer Language
SIMD	: Single Instruction Multiple Data
SVM	: Support Vector Machines
TB	: Test Bench
TF	: Top Function
TIE	: Tensilica Instruction Extension
WDBC	: Wisconsin Diagnosis Breast Cancer
WEKA	:Waikato Environment for Knowledge Analysis

Appendix B

Matrix Multiplication Experiment

In this Appendix, a study about the effects of variety of interconnects is accomplished for a matrix multiplication application.

B.1 Introduction

Matrix multiplication is the base operation for a variety of algorithms, most specifically for machine learning algorithms. The operation of multiplying two matrices is a time consuming operation. This time increases with an increase in the size of the matrices. Accelerating matrix multiplication is crucial for many real time applications. FPGAs are appropriate platforms for the acceleration purpose.

Even thoug FPGAs provide a good improvement in performance, combined with a low power consumption, this performance can be highly affected by the type of interconnect used for providing data to the hardware accelerators (IPs).

To select the appropriate interconnect for the data transformation method between the

IP and the data provider, an experiment is initiated to study the behavior of the available communication interconnect between the IP and the data provider. In this experiment, the matrix multiplication application design is transferred into the hardware architecture. Multiple ARM-IP designs are implemented in Zynq FPGA.

B.2 Data Sets (Benchmark)

Two 2-dimensional arrays, A and B, are multiplied. The values of A and B are artificially created as:

$A[i][j] = i + j$

$B[i][j] = i * j$, where i and j are random integer numbers

Four different sizes of integers are implemented for the A and B matrices: the (8x8), (16x16), (32x32) and (50x50) integers.

B.3 Tools and Platforms

The design exploration of matrix multiplication is carried out using the Xilinx Vivado HLS 2015.1 tool running on Intel®Core™i7 CPU 920 @ 2.67GHz. The ARM dual-core Cortex-A9 MPCore processor is used in the Zynq FPGA board with a 512 MB DDR3 on-chip memory.

B.4 Experimental Setup

In this experiment, two main steps are implemented: optimizing the baseline design and applying four available communication interfaces on the baseline designs and the optimized designs.

B.4.1 Matrix Multiplication Architectures

Four custom architectures are designed for different sizes of matrices. Each architecture is designed to create a number of processing elements equal to the number of rows or columns (all matrices are square). The processing elements multiply and add one row with one column in parallel. Two solutions are implemented for each size of the matrix: the baseline and the optimized architectures. Partition, Pipeline, Inline and Unroll directives are applied to create the optimized architecture.

B.4.2 Types of Interconnects

The following interconnects between the ARM processor and the HLS IP core are tested:

1. AXI_Light interconnect.
2. AXI_BRAM interconnect.
3. Streaming data through ACP interconnect.
4. Streaming data through High performance (HP) interconnect.

B.5 Matrix Multiplication results

This section includes results obtained from the matrix multiplication experiment. Tables B.1, B.2, B.3, and B.4 present the execution time in clock cycles, and speedups over software for the baseline and best architectures after implementing mapping to the FPGA platform for matrices of sizes 8x8, 16x16, 32x32 and 50x50 respectively. Tables B.5, B.6, B.7, and B.8 represent the estimated clock cycles, resources, and the speedup over the baseline produced by Xilinx Vivado HLS simulation runs for matrices of sizes 8x8, 16x16, 32x32 and 50x50 respectively.

Baseline(FPGA run), SW = 7,753			
Interface	**HW**	**Speedup_SW**	
AXIL	24,453	0.32	
AXI+BRAMs	49,254	0.16	
BRAMs	28,687	0.27	
Stream(ACP)	34,246	0.23	
Stream(HP)	34,422	0.23	
Optimized (FPGA run)			
Interface	**HW**	**Speedup_SW**	**Speedup_Base**
AXIL	23,749	0.33	1.03
AXI+BRAMs	22,580	0.34	2.18
BRAMs	816	9.50	35.16
Stream(ACP)	6,342	1.22	5.40
Stream(HP)	6,309	1.23	5.46

Table B.1: Clock cycles and speedup for 8x8 matrices

Baseline(FPGA run), SW = 65,198			
Interface	**HW**	**Speedup_SW**	
AXIL	93,002	0.70	
AXI+BRAMs	299,831	0.22	
BRAMs	222,391	0.29	
Stream(ACP)	234,748	0.28	
Stream(HP)	234,074	0.28	
Optimized (FPGA run)			
Interface	**HW**	**Speedup_SW**	**Speedup_Base**
AXIL	93,000	0.70	1.00
AXI+BRAMs	87,930	0.74	3.41
BRAMs	2132	30.58	104.311
Stream(ACP)	11,599	5.62	20.24
Stream(HP)	11,754	5.55	19.91

Table B.2: Clock cycles and speedup for 16x16 matrices

Baseline(FPGA run), SW = 449,546			
Interface	**HW**	**Speedup_SW**	
AXIL	369,012	1.22	
AXI+BRAMs	1,089,545	0.41	
BRAMs	881,105	0.51	
Stream(ACP)	899,891	0.50	
Stream(HP)	900,210	0.50	
Optimized (FPGA run)			
Interface	**HW**	**Speedup_SW**	**Speedup_Base**
AXIL	222,121	2.02	1.66
AXI+BRAMs	217,979	2.06	5.00
BRAMs	3,749	119.91	235.03
Stream(ACP)	18,773	23.95	47.94
Stream(HP)	20,762	21.65	43.36

Table B.3: Clock cycles and speedup for 32x32 matrices

B.6 Analysis and Findings

By analyzing the results of this experiment, the following points were deduced:

Baseline(FPGA run), SW = 2,492,213		
Interface	**HW**	**Speedup_SW**
AXIL	907,410	2.75
AXI+BRAMs	7,506,474	0.33
BRAMs	6,701,094	0.37
Stream(ACP)	6,778,375	0.37
Stream(HP)	6,780,650	0.37

Optimized (FPGA run)			
Interface	**HW**	**Speedup_SW**	**Speedup_Base**
AXIL	549,617	4.53	1.65
AXI+BRAMs	474,999	5.25	15.80
BRAMs	8,646	288.25	775.05
Stream(ACP)	77,690	32.08	87.25
Stream(HP)	79,401	31.39	85.40

Table B.4: Clock cycles and speedup for 50x50 matrices

Baseline Simulation(HLS)									
Method	**Interval**	**HLS (Pre Synthesis)**				**HLS (Post Synthesis)**			
Interface	**Interval**	**BRAMs**	**DSP**	**FF**	**LUT**	**BRAMs**	**DSP**	**FF**	**LUT**
AXIL	4,532	9	4	511	509	5	3	1,176	932
BRAMs	4,242	0	4	175	168	6	3	2,421	1,996
ACP	4,551	3	4	245	287	4	3	4,081	3,189
HP	4,551	3	4	245	287	4	3	4,077	3,165
Optimized Simulation (HLS)									
Method		**HLS (Pre Synthesis)**				**HLS (Post Synthesis)**			
Interface	**Interval**	**BRAMs**	**DSP**	**FF**	**LUT**	**BRAMs**	**DSP**	**FF**	**LUT**	**X_Base**
AXIL	203	7	32	1,587	748	4	24	2,216	2,227	22.33
BRAMs	73	0	32	231	270	34	24	9,924	8,358	58.11
ACP	268	1	32	1,312	537	3	24	5,116	4,261	16.98
HP	268	1	32	1,312	537	3	24	5,112	4,261	16.98

Table B.5: Estimated clock cycles, speedup and resources of the HLS over the baseline implementation for 8x8 matrices

1. Matrix multiplication application is built on two main operations: multiplication and addition. These operations can be applied on the elements of the matrices inde-

Baseline Simulation(HLS)									
Method	**Interval**	**HLS (Pre Synthesis)**				**HLS (Post Synthesis)**			
Interface	**Interval**	**BRAMs**	**DSP**	**FF**	**LUT**	**BRAMs**	**DSP**	**FF**	**LUT**
AXIL	34,404	9	4	535	550	5	3	1,204	971
BRAMs	33,314	0	4	183	183	6	3	2,429	2,004
ACP	34,439	3	4	206	334	4	3	8,642	6,564
HP	34,439	3	4	206	334	4	3	4,099	3,171
Optimized Simulation (HLS)									
Method		**HLS (Pre Synthesis)**				**HLS (Post Synthesis)**			
Interface	**Interval**	**BRAMs**	**DSP**	**FF**	**LUT**	**BRAMs**	**DSP**	**FF**	**LUT**	**X_Base**
AXIL	780	7	64	2,857	1,231	4	48	3,410	3,596	44.11
BRAMs	266	0	64	462	447	66	48	17,683	14,957	125.24
ACP	1038	33	64	1,557	753	19	48	5,282	3,676	33.18
HP	1038	33	64	1,557	753	19	48	5,278	3,653	33.18

Table B.6: Estimated clock cycles, speedup and resources of the HLS over the baseline implementation for 16x16 matrices

Baseline Simulation(HLS)									
Method	**Interval**	**HLS (Pre Synthesis)**				**HLS (Post Synthesis)**			
Interface	**Interval**	**BRAMs**	**DSP**	**FF**	**LUT**	**BRAMs**	**DSP**	**FF**	**LUT**
AXIL	268,484	12	4	559	584	6	3	1,240	997
BRAMs	264,258	0	4	191	195	6	3	2,437	1,994
ACP	268,551	6	4	289	373	6	3	4,159	3,279
HP	268,551	6	4	289	373	5	3	4,121	3,183
Optimized Simulation (HLS)									
Method		**HLS (Pre Synthesis)**				**HLS (Post Synthesis)**			
Interface	**Interval**	**BRAMs**	**DSP**	**FF**	**LUT**	**BRAMs**	**DSP**	**FF**	**LUT**	**X_Base**
AXIL	3,085	72	128	3,263	1,567	36	96	3,731	2,052	87.03
BRAMs	1,035	0	128	2,901	815	65	96	34,839	27,603	255.32
ACP	4,110	66	128	2,984	1,374	35	96	6,613	4,261	65.34
HP	4,110	66	128	2,984	1,374	35	96	6,609	4,261	65.34

Table B.7: Estimated clock cycles, speedup and resources of the HLS over the baseline implementation for 32x32 matrices

Baseline Simulation(HLS)										
Method	**Interval**	**HLS (Pre Synthesis)**				**HLS (Post Synthesis)**				
Interface	**Interval**	**BRAMs**	**DSP**	**FF**	**LUT**	**BRAMs**	**DSP**	**FF**	**LUT**	
AXIL	1,015,304	48	4	701	762	24	3	1,399	1,048	
BRAMs	1,005,102	0	4	257	267	96	3	2,547	2,105	
ACP	1,015,407	24	4	428	550	15	3	4,302	3,324	
HP	1,015,407	24	4	428	550	14	3	4,264	3,251	
Optimized Simulation (HLS)										
Method		**HLS (Pre Synthesis)**				**HLS (Post Synthesis)**				
Interface	**Interval**	**BRAMs**	**DSP**	**FF**	**LUT**	**BRAMs**	**DSP**	**FF**	**LUT**	**X_Base**
AXIL	7,515	132	204	5,199	2,037	66	150	5,215	2,573	135.10
BRAMs	2,511	0	201	4,822	1,029	103	150	53,625	42,094	400.28
ACP	10,015	108	203	4,916	1,856	56	150	8,075	4,714	101.39
HP	10,015	108	203	4,916	1,856	56	150	8,071	4,688	101.39

Table B.8: Estimated clock cycles, speedup and resources of the HLS over the baseline implementation for 50x50 matrices

pendently (no dependency). Therefore, matrix multiplication is a good candidate to be transformed into hardware.

2. To get an accelerated matrix multiplication IP, it is crucial to select the type of interconnect for transferring data to avoid the communication overhead bottleneck.

3. AXI_BRAMs interconnect is the most time consuming interconnect for the baseline implementations because it requires the time for transferring data from the DDR to the BRAMs using AXI interconnect, in addition to the time of reading data from the BRAMs to the IP.

4. When the data size is increased, the amount of speedup over pure software implementation is increased.

5. When data size is increased, the improvement on the performance of hardware over

baseline implementation can be easily noticed for the StreamACP and StreamHP interconnects; however, there are very small improvement to the speedup when the AXI_Light interconnect is used. Accordingly, the AXI_Light interconnect is considered not sufficient for transferring data with large sizes.

6. AXI_Light interconnect can only be used for transferring small amounts of data, such as control signals, without affecting the overall speedup because it sends data serially from the PS to the IP.

7. The used of the AXI_BRAMs interconnect will require the use of the AXI interface for transferring data from the DDR into the BRAMs. Therefore, when the size of data is small, the obtained speedup is small. By increasing the size of the data, a better speedup can be obtained over the software; however, this speedup is still less than the speedup obtained by using streaming interconnects.

8. If the required data will be pre-stored in BRAMs, a huge speedup can be obtained (better than any other interconnect used) over pure software implementation as the effect of the communication overhead will be reduced. This case can be true if the sizes of the BRAMs are enough for the required data.

9. With huge data, the most sufficient option is streaming data via the stream HP or stream ACP interconnect.

10. The best speedup over the software implementation obtained in this experiment was achieved by using these interconnects in this order: 1) ACP (Max speedup = 87x); 2) HP (Max speedup = 85x); 3) AXI-BRAMs (Max speedup = 15x); 4) AXIL (Max speedup = 1.65x); and 5) BRAMs (Max speedup = 288.3x).

11. Fewer resources are used and less power is consumed with architectures that use AXI_Light interconnect due to the simplicity of the design. On the other hand, architectures that use AXI_BRAMs interconnect require the most resource and power consumption due to the heavy use of BRAM resources.

B.7 Summary

The type of interconnect used for transferring data to the hardware plays a crucial role in the acceleration factor. In this experiment, it was found that the acceleration factor is highly dependent on the size of and the type of interconnect used. Streaming data from the cache memory to the hardware IP is the best interconnect selection for a large data size. AXIL interconnect is the simplest but the slowest method for transferring data.

Pre-stored data in BRAMs produces a huge amount of speedup over the software implementation. This type of implementation can only be applied when the available size for the BRAMs is enough for saving the data. If a simpler interconnect design is created, fewer resources are used and less power is consumed.