FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Dynamically Reconfigurable Multi-Classifier Architecture on FPGA

Joana Lima Macedo

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Paulo de Castro Canas Ferreira

Second Supervisor: Nuno Miguel Cardanha Paulino

October 26, 2020

Resumo

Nos últimos anos, a aprendizagem automática tornou-se predominante em diversas áreas de investigação, incluindo a área do reconhecimento de atividades humanas. Contudo, com o aumento da complexidade e a necessidade de maior precisão para aplicações práticas, o desempenho e a energia necessária tornam a implementação impraticável. Para melhorar o desempenho, bem como manter o baixo consumo de energia, o uso de *hardware* reconfigurável despertou interesse.

Para superar o problema da complexidade, melhorar o desempenho e manter o baixo consumo de energia, é proposto a implementação e validação de uma arquitetura de *hardware*. O objetivo deste trabalho consiste em conceber e avaliar uma arquitetura reconfigurável em tempo de execução baseada em FPGA que suporte a implementação dinâmica de múltiplos núcleos de aceleradores.

A ideia principal é desenvolver uma arquitetura que acelere classificadores de aprendizagem automática, executados num processador de uso geral, mapeando de forma transparente partes computacionalmente intensivas da sua execução para *hardware* reconfigurável. Com esta estratégia, o desempenho da aplicação irá ser melhorada uma vez que as unidades irão agir como co-processadores de aceleração do processador de uso geral.

Os resultados obtidos durante esta pesquisa mostram que os aceleradores gerados podem ser usados para executar algoritmos complexos de classificação de aprendizagem automática com uma quantidade grande de dados. Também foi explorada a capacidade de escalar o sistema para diversos processadores, cada um com o correspondente acelerador. No entanto, os resultados experimentais mostram que o desempenho do acelerador para o algoritmo *K-nearest neighbours* é afetado significativamente pelos atrasos introduzidos pelo mecanismo de acesso à memória externa.

Abstract

In the past few years, machine learning has become predominant in various research fields, including human activity recognition. However, with the increasing accuracy requirements and complexity for practical applications, the speed and the power cost make the implementation impracticable. To improve the performance as well as to maintain low power consumption, the use of reconfigurable hardware has aroused interest.

Implementation and validation of a hardware architecture are proposed to overcome the complexity problem while improving the speed-performance of classification as well as maintaining the low power consumption. This project aims to devise and evaluate a complete FPGA-based, run-time reconfigurable architecture that supports the dynamic deployment of multiple accelerator cores.

The main idea is to develop an architecture which accelerates machine learning classifiers, running on a general-purpose processor, by transparently mapping computationally intensive parts of their execution to reconfigurable hardware. With this strategy, the performance of the embedded application will be enhanced since the units will act as acceleration co-processors of the general-purpose processor.

The results obtained during this research, show that the generated custom loop accelerator could be used to compute complex machine learning classifiers with an increasingly large amount of data. It also devised the ability to scale the system architecture to several processors, each one with the correspondent accelerator as a co-processor. However, the experimental results show that the accelerator performance for the K-nearest neighbours algorithm is significantly affected by the overhead introduced by the external memory accesses mechanism.

Agradecimentos

Desejo exprimir os meus agradecimentos a todos aqueles que de alguma forma permitiram que esta tese se concretizasse. Em primeiro lugar desejo agradecer ao meu orientador, professor João Canas Ferreira, pela orientação e disponibilidade durante o desenvolvimento desta dissertação. De igual modo agradeço ao meu coorientador, professor Nuno Miguel Cardanha Paulino, pelo disponibilidade e apoio na resolução de problemas que foram surgindo. Desejo também agradecer ao professor José Carlos dos Santos Alves pela preocupação que demonstrou sobre o estado da presente dissertação e pela oportunidade que me proporcionou. Este trabalho foi realizado no âmbito do projeto PTDC/EEI-HAC/30848/2017 financiado pela FCT (Fundação para a Ciência e Tecnologia).

Desejo agradecer ao meu namorado, Daniel Granhão, por todo o apoio, não só durante esta dissertação mas também ao logo de todo o meu percurso académico, e por sempre acreditar nas minhas capacidades. Agradeço também aos meus pais e à minha irmã por me terem proporcionado esta oportunidade e por estarem sempre presentes. De igual modo agradeço aos meus amigos, Bruno Baptista, Marco Silva e Pedro Neto, pela amizade e companheirismo desde o início do meu percurso académico.

Joana Lima Macedo

Contents

List of Figures

Listings

Abbreviations and Acronyms

AXI	Advanced eXtensible Interface
BNN	Binarized Neural Network
BRAM	Block RAM
CDFG	Control-Data Flow Graphs
CLA	Customized Loop Accelerator
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CR	Communication Routine
DDR	Double Data Rate
DMA	Direct Memory Access
DOT	Graph Description Language
DPR	Double Data Rate
DTC	Decision Tree Classifier
ELF	Executable and Linkable Format
EMIO	Extended Multiplexed Input/Output
FC	Fully Connected
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
FSBL	First Stage Boot Loader
FSL	Fast Simplex Link
FSM	Finite-State Machine
GPIO	General-Purpose Input/Output
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
HAR	Human Activity Recognition
HDL	Hardware Description Language
HLS	High-Level Synthesis
ICAP	Internal Configuration Access Port
ILA	Integrated Logic Analyzer
ILP	Instruction-Level Parallelism
IOP	Input/Output Peripheral
IP	Intellectual Property
IPC	Instructions Per Cycle
KNN	K-Nearest Neighbors
LMB	Local Memory Bus
LSTM	Long Short-Term Memory

MFS	Memory File System
MLP	Multilayer Perceptron
MPSoC	MultiProcessor System-on-a-Chip
PL	Processing Logic
PS	Programmable System
RM	Reconfigurable Module
RNN	Recurrent Neural Network
RPU	Reconfigurable Processing Unit
RTL	Register Transfer Level
SVM	Support Vector Machines

Chapter 1

Introduction

1.1 Context

In the past few years, machine learning has become predominant in various research fields, including human activity recognition. The increasingly large amount of data in human activity datasets requires machine learning methods [1]. The significant progress in the automation of human behaviour recognition, using machine learning classifiers to detect human actions, provides the needed services automatically and instantly for maximizing user safety and comfort [2]. However, with the increasing accuracy requirements and complexity for practical applications, the speed and the power cost become impracticable. To improve the performance as well as to maintain low power consumption, the use of reconfigurable hardware has aroused interest.

The use of reconfigurable hardware has been shown to enable considerable speedups when compared with implementations running on general-purpose central processing units (CPUs) and even general-purpose graphics processing units (GPUs). The speedups are achieved by extensively exploiting the ability to customize the hardware architecture to the specific parameters of the actual problem instances. Reconfigurable hardware establishes a bridge between hardware and software, achieving potentially much higher performance than software while maintaining a higher level of flexibility than hardware [3]. The most common type of reconfigurable hardware device is a field-programmable gate array (FPGA) due to its flexibility, hardware-timed speed, reliability, and parallelism.

FPGAs are reconfigurable integrated circuits that contain an array of programmable logic blocks. In addition to all the advantages already mentioned, some FPGA families can reconfigure a part of the chip while other areas remain working. This feature is widely used in accelerated computing. Due to their programmable nature, FPGAs are an ideal fit for many different markets and are especially interesting in the human behaviour recognition context. Another profit is the ability to perform the computational tasks on-spot, which improves response time and saves bandwidth.

In the context of human activity recognition, identifying the specific movement or action of a person based on sensor data is a challenging time series classification task. It is a challenging

problem as there are no direct ways to relate sensor data to specific human activities since each subject may perform an activity with significant variation. The intent is to predict the activity given a database with recorded sensor data corresponding to several subject activities. A hardware architecture for an ensemble of learning algorithms is considered in this work to overcome the classification challenge.

Ensemble methods are learning algorithms that construct a set of classifiers and then classify new data by taking a weighted vote of their predictions [4]. According to the authors, the use of this type of approach can often perform better than any single classifier. Additionally, ensembles are well-established as a method for obtaining highly accurate classifiers by combining less accurate ones.

1.2 Motivation

The field of human activity recognition provides the ability to detect current activity based on sensor data which can be collected from wearable sensors or accelerometer. This subject has become one of the trendiest research topics due to the availability of sensors and accelerometer, the low cost and less power consumption, and the advancement in machine learning [5]. Several activities are recognized though human activity recognition such as walking, standing, laying down, walking upstairs, walking downstairs, and sitting.

It can be extensively used for eldercare and healthcare as an assistive technology when combined with other technologies, for recognizing driving activities and lead to safe travel, for recognizing military action, along with others. Due to the significance of the application areas, there is a high necessity of accuracy when classifying human behaviours, as well as reduced execution time and power consumption. Hence, feature extraction and learning methods should be carefully chosen to guarantee a reasonable response time. All these constraints, combined with the need for running complex algorithms to achieve satisfactory classification rates, give rise to a challenging design problem in the field of embedded systems [6].

1.3 Objectives

Implementation and validation of a hardware architecture are proposed to overcome the complexity problem while improving the speed-performance of classification as well as maintaining the low power consumption. This project aims to devise and evaluate a complete FPGA-based, run-time reconfigurable architecture that supports the dynamic deployment of multiple accelerator cores.

The main objective can be divided into three particular objectives as over the next lines to provide a better understanding. Devise an FPGA-based architecture that supports dynamically reconfigurable accelerator cores. Exploit and evaluate the scalability of the FPGA-based architecture to multiple support of dynamically reconfigurable accelerator cores. Improve the speed-performance

of classification when compared with a classifier running on general-purpose CPUs by making use of the hardware reconfiguration and parallelism features of an FPGA.

1.4 Dissertation Structure

The remaining of this document is structured as follows. Chapter 2 provides a revision of machine learning classifiers in the human activity recognition context, some background on ensemble methods, and implementation of accelerator hardware. Chapter 3 presents a problem description and the proposed mechanism and architectures to overcome this problem. Chapter 4 describes the implementation of the proposed mechanism and presents the obtained results. Lastly, in chapter 5 conclusions about the results are taken, and recommendation to provide continuity to this work are given.

Chapter 2

Background and Literature Review

The following sections start by a revision of machine learning classifiers used to classify human activities. The classifiers were compared to each other and evaluated in terms of accuracy. Then, some background is provided on ensemble methods, following by the reasons way ensemble may work better than single classifiers. Next, a few techniques are exploited to develop hardware accelerators for machine learning classifiers. Finally, an FPGA-based approach that automatically generates pipelined customized loop accelerators from run-time instruction traces is explored.

2.1 Human Activity Recognition

Human activity recognition (HAR) is the task of correctly identifying human actions and activities given sensory inputs [7]. The sensor data can be remotely recorded, such as video, or be recorded directly on the subject, by carrying custom hardware or smartphones with accelerometers and gyroscopes. Accurate activity recognition remains a challenge and an active area of research due to its complexity and diversity [8]. Various classifiers have been proposed in the literature to overcome the problem of correctly identify human actions and activities, given sensory inputs. Machine learning classifiers are an increasing trend in human activity recognition, in addition to statistical and probabilities models.

2.1.1 Machine Learning Classifiers

In [7], an evaluation of the state-of-the-art of classifiers for HAR is performed by using the same set of identified activities. Implemented classifiers were trained to recognize the most common set of activities when a smartphone is used. Six classifiers were chosen based on their broad usage on activity recognition. In the next paragraphs, a brief description of the classifiers evaluated in [7] is provided.

K-Nearest Neighbours The K-nearest neighbours (KNN) classification is one of the most fundamental and straightforward classification methods. It should be one of the first choices for

classification when there is no prior knowledge about the distribution of the data [9]. The algorithm has been extensively used on human activity recognition domain, due to its ability to handle multi-class problems.

The algorithm essentially forms a majority vote between the K most similar instances to a new given observation. The similarity is defined according to a distance metric between two data points. The commonly used distance metric is the Euclidean distance [1].

Decision Trees A decision tree classifier (DTC) is a flow-chart tree structure consisting of nodes which test for the value of a specific attribute, branches that correspond to the outcome of a test and connect to the next node or leaf, and leaves that are terminal nodes that predict the outcome representing class labels or class distribution.

One of the drawbacks of DTC is that the performance strongly depends on how well the tree is designed. The design of a DTC can be decomposed into the following tasks: 1. appropriate choice of the tree structure, 2. the choice of feature subsets to be used at each internal node and, 3. the choice of the decision rule or strategy to be used at each internal node [10]. There may be some difficulties involved in designing an optimal DTC.

Multilayer Perceptron A multilayer perceptron (MLP) is a class of feedforward artificial neural network. It consists of multiple layers of neurons that interact using weighted connection. After the input layer there are usually any number of intermediate (or hidden) layers followed by an output layer. Weights measure the degree of correlation between the activity levels of neurons that they connect [11].

In the training phase, each pattern of the training set is used in succession to clamp the input and output layers of the network. After several sweeps through the training data, the network is supposed to have learned the relationship between the input and output vectors in the training sample. In the testing phase, the algorithm is expected to be able to utilize the information encoded in its connection weights to assign the correct output labels for the test vectors.

Convolutional Neural Network Convolutional neural network (CNN) is a deep learning module for processing data that has a grid pattern. It is designed to automatically and adaptively learn spatial hierarchies of features through backpropagation by using multiple building blocks, such as convolution layers, pooling layers, and fully connected layers. The convolution and the polling layer perform feature extraction, whereas the fully connected layer maps the extracted features into final output [12].

The convolution layer performs feature extraction which typically consists of a combination of linear and nonlinear operations, such as convolution operation and activation function. The pooling layer provides a downsampling operation which reduces the dimensionality of the feature maps in order to introduce a translation invariance to shifts and distortions. The commonly used pooling operation is maximum polling which extracts patches from the input feature maps, output

[1] Straight-line distance between two points in Euclidean space.

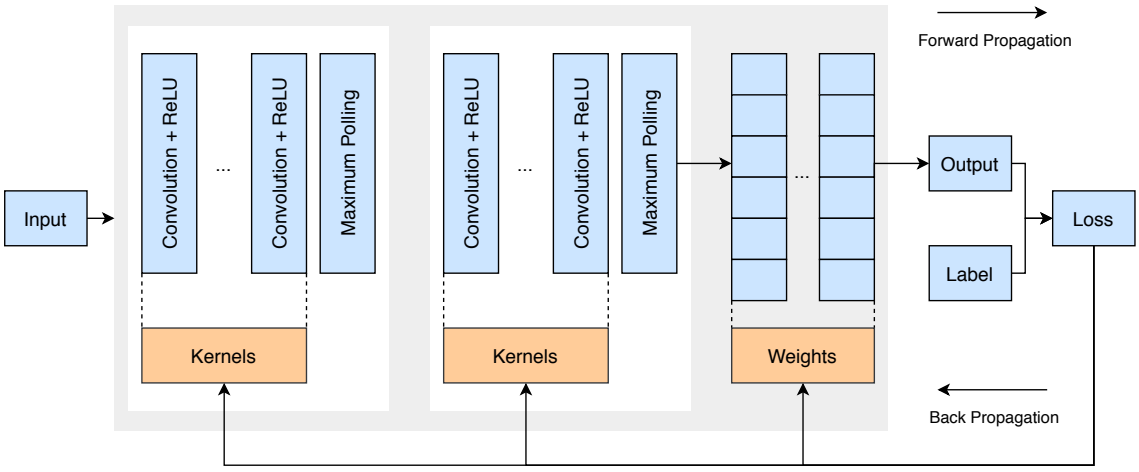

Figure 2.1: An overview of a CNN architecture and the training process from [12]

the maximum value in each patch, and discards the other values. The fully connected layer transforms the output feature maps into a one-dimensional array and connects them into one or more fully connected layers.

Figure 2.1 presents an overview of a CNN architecture and the training process. The model is composed of three building blocks: convolution layers, maximum pooling layers and fully connected layers. The performance of the model is executed with a loss function through forwarding propagation on a training dataset. Learnable parameters, such as kernels and weights, are updated according to the loss function value through backpropagation.

Recurrent Neural Network Recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence, allowing previous outputs to be used as inputs while having hidden states. It is a derivation from the feedforward neural network due to use internal memory to process variable-length sequences of inputs.

Long short-term memory (LSTM) is a particular RNN architecture design to model temporal sequences and their long-range dependencies more accurately. The LSTM contains memory blocks in the recurrent hidden layer and multiplicative units called gates to control the flow of information. In addition to the gates, the memory blocks contain memory cells with self-connections storing the temporal state of the network. Each memory block contains an input gate to control the flow of input activations into the memory cell, and an output gate to control the output flow of cell activations into the rest of the network [13].

Support Vector Machine Support vector machine (SVM) aims to find a hyperplane [2] in an N-dimensional space, where N represents the number of features, that distinctly classifies the data points. Hyperplanes are decision boundaries through which data points positioned on either side

[2]Subspace whose dimension is one less than that of its ambient space.

can be attributed to different classes. The data points closer to the hyperplane are called support vectors. These vectors control the position and orientation of the hyperplane. The objective is to find a plane that has the maximum margin, that is the maximum distance between data points of different classes. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more accuracy.

The classifiers evaluated in [7] were implemented based on already published papers. The KNN algorithm was tested with a varying number of neighbours. The best result was observed when the number of neighbours was set to 18. All neighbours were equally weighted. Other than the DTC, an ensemble method constituted by 70 DTCs designated random forest was implemented. Each tree in the random forest predicts a class and the class with the most votes becomes the model's prediction. The MLP was implemented using Keras and consisted of 3 hidden layers. The structure of the implemented CNN was a convolution layer followed by a maximum pooling layer, repeated three times and a fully connected layer before the output layer. The LSTM consisted of 3 hidden layers. Considering SVMs have initially been designed for binary classification [14], the algorithm was implemented by constructing and combining several binary classifiers.

In the dataset used, there are six different classes, each representing a single activity: walking, standing, laying down, walking upstairs, walking downstairs and sitting. All classifiers performed the same set of activity drills and after training each classifier was evaluated against the test set. The evaluation results were presented thought F1-score metric for each method and activity. F1-score is the harmonic mean of the precision [3] and recall [4]. The deep learning techniques achieved a significantly higher score, especially the CNN classifier with an F1-score of 0.94. The RNN reached an F1-score of 0.84. In the next baseline are random forest, KNN and SVM with an F1-score of 0.72, 0.69 and 0.68 respectively. Lastly, DT and MLP achieved an F1-score of 0.62.

2.2 Ensemble Methods

An ensemble of classifiers is a set of classifiers whose individual decisions are combined, typically by weighted or unweighted voting, to classify new data [4]. A relevant discovery is that ensembles are often much more accurate than the individual classifiers that make them up. The three fundamental aspects that enable an ensemble to perform better than a single classifier are statistical, computational and representational.

1. The statistical advantage arises when the amount of training data available is too small compared to the size of the hypothesis space. Through the construction of an ensemble out of classifiers, the algorithm can "average" their votes and reduce the risk of choosing the wrong classifier.

[3] The number of true positives divided by the number of false positives plus true positives.
[4] The number of true positives divided by the number of true positives plus false negatives.

2. The computational advantage appears when learning algorithms perform a local search that may get stuck in local optima. An easy solution would be an ensemble constructed by running the local search from many different starting points. This solution may provide a better approximation to the real unknown function than any of the individual classifiers.

3. The representational advantage arises when the real function cannot be represented by any of the hypotheses in the hypotheses space. By forming weighted sums of hypotheses drawn from the hypotheses space, it may be possible to expand the space of representable functions.

2.3 Hardware Accelerators for Machine Learning Classifiers

Previous research had shown that Moore's Law [5] has mostly ended [15], and although this might be arguable, there is no denying that this is inevitable with any technology, it eventually saturates out. Moore's Law conditions the scalability of processors performance and efficiency. Therefore, to continue scaling the performance and efficiency of computing hardware, alternative architectures, such as domain-specific hardware accelerators, are one of the few ways [16].

According to [16], domain-specific accelerators exploit four main techniques for performance and efficiency gains: 1. data specialization, 2. parallelism, 3. local and optimized memory, and 4. reduced overhead. Specialized logic on domain-specific data can optimize both performance and efficiency. High levels of parallelism provide gains in performance. Through the storage of data structures in many local memories, very high bandwidth can be achieved with low cost and energy. The overhead of program interpretation can be reduced or eliminated by specializing hardware.

In the following subsections, a few techniques to develop hardware accelerators to perform machine learning classifiers are exploited. Firstly, in subsection 2.3.1, the high-level synthesis (HLS) technique to develop hardware accelerates for machine learning classifiers is exploited. Then, subsection 2.3.2 describes and evaluates the dynamic partial reconfiguration (DPR) technique. In both subsections, a speedup comparison between the generated accelerators and a general-purpose processor (GPP) for the given machine learning classifier is evaluated.

2.3.1 High-Level Synthesis

HLS is an automated design process that creates the digital hardware that implements the desired function taking as input the algorithmic description [17]. The algorithmic description is written in a high-level programming language such as C/C++. The automated process provides the register transfer level (RTL) hardware description.

[5] Observation that the number of transistors on a microchip doubles every two years, though the cost of computers is halved.

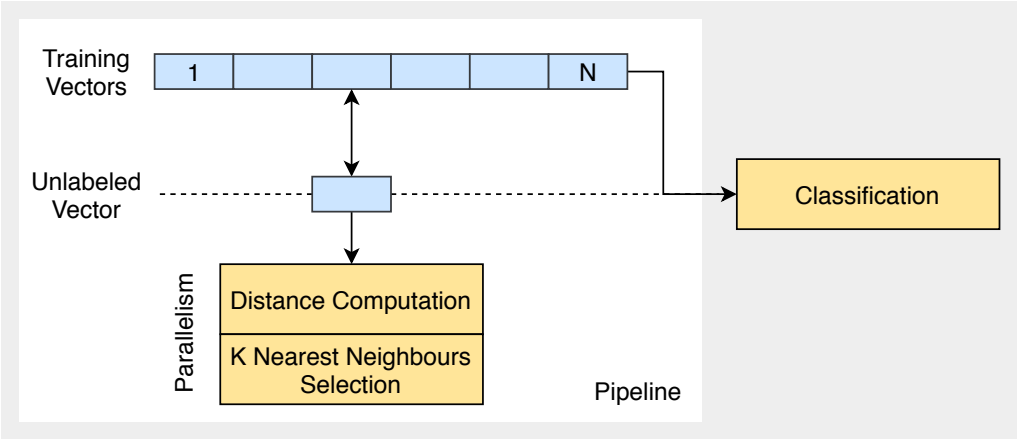

Figure 2.2: The KNN hardware architecture from [18]

In [18], an intellectual property (IP) core for the hardware acceleration of the KNN classifier is design. The design implementation consisted of the hardware architecture part for the KNN algorithm execution and the data transmission part for data transfer between the hardware architecture and the memory.

The hardware architecture part for the KNN algorithm execution was divided into three blocks whose tasks were: 1. to calculate the Euclidean distance between the unlabeled vector and all the training vectors, 2. to select the K nearest training vectors to the unlabeled vector, and 3. to classify the unlabeled vector by determining the class label that occurs most in the K-nearest training vectors. The first block consisted of several calculation units to calculate the distance between the unlabeled vector and all the training vectors. The second block consisted of K compactors used to find the K nearest neighbours and store their correspondent distances and class labels. The third block consisted of C counters and a comparator to perform the majority voting. Each counter is associated with one of the class labels and is used to record the frequencies of each class label in the K nearest neighbours. The comparator is used to find the class label with the highest frequency.

Figure 2.2 presents KNN hardware architecture. The first and the second block are arranged to perform in a parallel way, while each of them performs in a pipelined manner individually. Pipelining allows the output of one distance every clock cycle after the first distance result is obtained. The second block starts executing once the first block outputs the first distance. Hence, the execution time of the first two tasks of the algorithm can be reduced from (K+1)N to approximately N clock cycles, where N is the size of the training vector. The third block starts executing when the pipelining is completed, and the K nearest neighbours are found.

Throughout the evaluation of algorithm execution, the authors prove that the performance of the HLS-based implementation is 35.1$\times$ faster than the GPP-based implementation and close to the hardware description language (HDL) based implementations. Furthermore, compared with the HDL-based implementations, HLS technique reduces the development complexity and cost of hardware design.

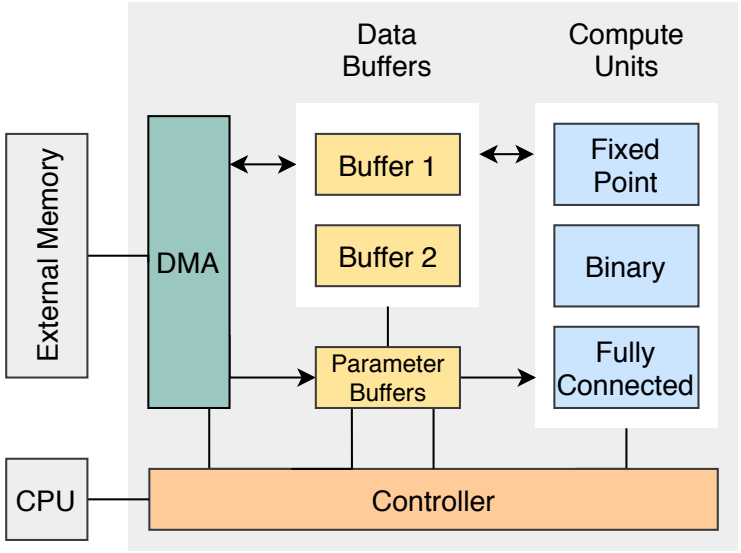

Figure 2.3: The system-level block diagram of the BNN accelerator from [19]

In [19], a binarized convolutional neural network (BNN) accelerator is designed and synthesized from C++ to FPGA-targeted Verilog. A BNN is a feedforward network where weights and activations are predominantly binary. During the feedforward, BNNs reduce significantly memory size and accesses and replace the majority of arithmetic operations with bit-wise operations, which is expected to improve power efficiency substantially [20].

The architecture of the BNN model that the accelerator of [19] targets consists of six convolution layers followed by three fully connected layers. All the layers apply batch norm before binarization, and the convolution layers use 3×3 filters and edge padding. After the second, fourth and sixth convolution layers is a 2×2 maximum polling layer. The input of the first convolution layer is floating-point and not binary as the other duo to be an image, although the weights are still binary.

Figure 2.3 presents the system-level block diagram of the BNN accelerator. The system architecture consisted of three compute units, data and weight buffers, direct memory access (DMA) system for off-chip memory transfer, and a finite-state machine (FSM) controller. The three compute units work on different types of layers: 1. the fixed-point convolution unit for the non-binary first convolution layer, 2. the binary convolution unit for the five binary convolution layers, and 3. the binary fully connected unit for the three fully connected layers. The design uses two in-out data buffers of equal size in order to avoid the extensive use of external memory. One layer reads from buffer 1 and writes its outputs to buffer 2. Then, without any external memory data transfers, the next layer reads from buffer 2 and writes to buffer 1. Therefore, external memory data transfers are only mandatory to the input image, the output prediction, and for loading each layer's weights.

The authors produce a performance comparison between the designed BNN accelerator, a multicore processor, a GPU and an embedded GPU. The designed BNN accelerator obtained $15.1\times$ better performance and $11.6\times$ better throughput per Watt over the embedded GPU. Versus the

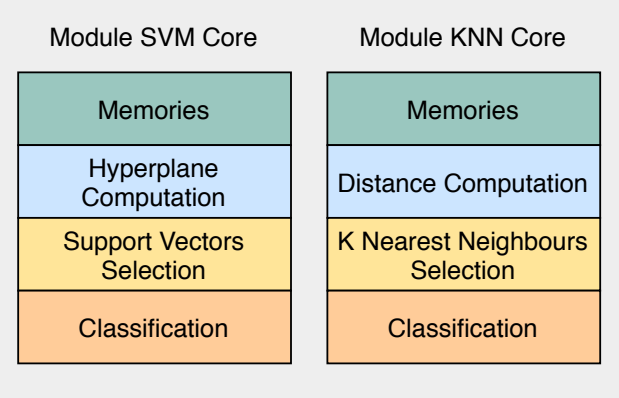

Figure 2.4: Block diagram of the modules for the SVM and KNN cores from [22]

multicore processor, it accomplishes a 2.5× speedup. Against the GPU it performs 8.1× worse, although it has significantly lower power consumption and better throughput per Watt.

2.3.2 Dynamic Partial Reconfiguration

Partial reconfiguration is the ability to reconfigure selected areas of the FPGA after its initial configuration. DPR is a particular instance of partial reconfiguration which permits to change a part of the device while the rest of the FPGA is running [21]. According to the authors, some of the advantages acquired when using this method are: 1. allowing the FPGA to adapt to changing hardware algorithms, 2. improving fault tolerance and resource utilization, 3. enhancing performance or reducing power consumption. DPR is incredibly valuable for devices operating in a mission with a critical environment that cannot be interrupted while some subsystems are being redefined.

In [22], implementation of two standard classifiers, namely, SVM and KNN, is proposed to form a multi-classifier FPGA architecture which can utilize particular regions of the FPGA to perform as either. The architecture allows for interchanging various copies of the KNN and SVM cores.

Figure 2.4 summarizes the main building blocks which constitute the SVM and KNN cores. The authors proceeded to construct the multi-classifier starting by creating a design wrapper which instantiates a black-box. The black-box is an empty core compatible in the number of I/O ports with both cores. It has the same I/O ports than the SVM core, as the SVM I/O ports include all the KNN I/O ports plus an additional port exclusive to the classifier. The black-box needs to be loaded with the logic contents of either SVM or KNN classifier. Thus, reconfigurable modules (RMs) need to be created with a size based on the most extensive logic resource requirement. RMs are several images of the KNN and SVM cores with different parameters and data sizes. After the implementation of different configurations, the bitstreams were generated to create a library of configuration files used to configure the device.

In comparison with having to configure the same device with two classifiers, the authors estimated that this solution saves in reconfiguration time, area footprint, and in power consumption.

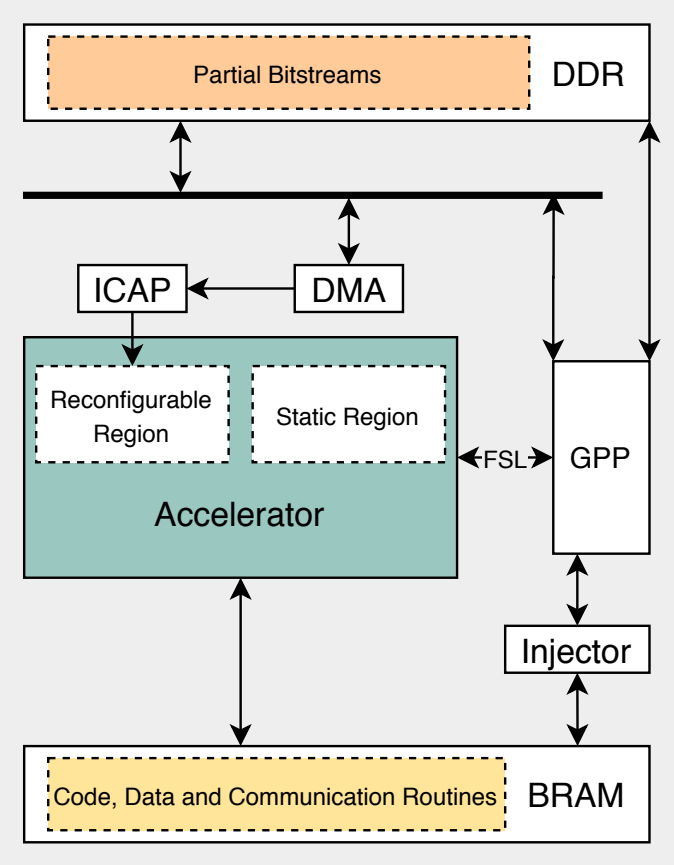

Figure 2.5: Custom loop accelerator system architecture from [23]

An approximately 8× speedup in reconfiguration time was achieved. The implementation has several advantages: 1. it saves time when modifying the defined reconfigurable regions to run another classifier, 2. it consumes much less power reconfiguring a particular region than having to reconfigure the whole device, and 3. it reduces the space and area footprint occupied by the hardware design, allowing the use of smaller and cheaper FPGAs for running multiple tasks.

2.4 Dynamic Partial Reconfigurable Tailored Accelerators

As previously stated in chapter 1, the use of specialized accelerator circuits is a viable solution to address performance and energy issues in embedded systems. In [23], an FPGA-based approach that automatically generates pipelined customized loop accelerators (CLAs) from run-time instruction traces is proposed. The CLA is also enhanced with DPR support to overcome the high area and resource requirements issue when offloading a large number of kernels form the target application. Each kernel to accelerate is implemented as a variant of a reconfigurable area of the CLA which hosts all functional units and configuration memory.

The proposed approach is based on the automated generation of a specialized hardware instance capable of executing selected portions of an application. Figure 2.5 presents the system

architecture overview of the CLA capable of autonomous run-time self-adaptivity. The system consists of a single MicroBlaze, one CLA instance, a local block RAM (BRAM) containing application code, data, and automatically generated communications routines (CRs) used to invoke the CLA. The processor and CLA communicate via fast simplex link (FSL) point-to-point connections. The external memory is used exclusively to store partial bitstreams. The partial reconfiguration of the CLA is performed via the FPGA's internal configuration access port (ICAP) primitive, which is fed by a DMA module.

At run-time, the injector monitors the instruction address bus of the MicroBlaze. When the address corresponds to the start address of a critical portion of the application, the injector replaces the fetched instruction with an unconditional branch to the address containing the respective CR. The routine has several tasks: 1. it sends operands from the register file of the processor to the CLA, 2. it calls the function that drives the DPR step by configuring the DMA with the start address and amount of data to fetch, 3. waits for CLA execution to complete, and 4. returns to the address at which the injector intervened. The execution of the application is thus accelerated, by pipelining loop iterations on the CLA and exploiting the latent instruction-level and data-parallelism present in the binary code.

The function that triggers the DPR step copies the partial bitstreams from flash to double data rate (DDR) memory at startup and drives the DMA engine which feeds the ICAP. When called, it checks if the CLA's reconfigurable area is already configured with the desired partial bitstream. If this is the case, the CR usually resumes. Otherwise, the reconfiguration routine writes the partial bitstream to the reconfigurable area of the CLA thought the ICAP peripheral.

The authors performed an experimental evaluation for a set of 23 kernels. The kernels were grouped so that similar workloads were found within the same application. Each group was used to generate two CLAs tailored for the execution of the respective kernels. For the comparison purpose, one of the CLAs was implemented without resorting to DPR but still capable of accelerating the same subset of kernels. Therefore, those CLAs were generated only by one partial bitstream and had a single-cycle reconfiguration time when switching between the execution of their supported kernels. The other CLAs benefited from DPR and were implemented by generating one partial bitstream per kernel supported.

The results indicate that resorting to DPR supports multiple loop kernels per CLA instance while minimizing area usage and maximizing operating frequency. With the increase of the number of supported kernels, the benefits of resorting to DPR were more evident, with area savings reaching upward of $1.61\times$. However, a penalty incurred with the initial DPR overhead. The time required for each partial reconfiguration varies with the partial bitstream size. For DPR performed with small partial bitstreams, the total overhead is more significant relative to less frequent reconfigurations with larger bitstreams.

2.5 Conclusion

Providing accurate information on human's activities and behaviours is one of the most challenging tasks and an active area of research due to its complexity and diversity. Several machine learning classifiers have been proposed in the literature to overcome the problem of correctly identify human actions.

An evaluation of the state-of-the-art of machine learning classifiers to classify human actions reveals that deep learning techniques, such as CNN, achieve higher accuracy, although it requires complex computation. It was also recognized that the KNN algorithm is a classifier that achieves near accuracy and is considered a straightforward classification method.

The use of specialized accelerator circuits to perform machine learning classifiers is a feasible solution to address performance and energy problems in embedded systems. The four main techniques exploited by the domain-specific accelerators to achieve performance gains are: 1. specialized logic on domain-specific data, 2. high levels of parallelism, 3. optimized storage of data structures, and 4. the elimination of the overhead of program interpretation.

Two hardware accelerators to perform machine learning classifiers were developed through an HLS-based implementation, and their performance was evaluated. For the KNN algorithm, the HLS-based implementation presets a performance 35.1$\times$ faster than a GPP-based implementation and close to the HDL-based implementation. For the BNN algorithm, the HLS-based accelerator obtained 15.1$\times$ better performance over an embedded GPU, 2.5$\times$ speedup versus a multicore processor and 8.1$\times$ worse performance against the GPU.

Another accelerator, to perform machine learning classifiers, was evaluated but this time by exploiting DPR techniques. The implementation consisted of two standard classifiers, namely, SVM and KNN, for a multi-classifier architecture where the accelerator could perform as either. An approximately 8$\times$ speedup in reconfiguration time was achieved in comparison with having to configure the same device with the two classifiers. This solution saves in reconfiguration time, area footprint, and power consumption.

A different approach to automatically generate custom accelerators from run-time instruction traces was also explored. The approach was based on the automated generation of specialized hardware instances capable of executing selected portions of an application. The proposed accelerator also benefits from DPR support to overcome resource requirements issues. Although this technique was not tested to perform machine learning classifiers, it would be an interesting approach that may prove to be beneficial to the current scenario.

Chapter 3

Dynamically Reconfigurable Multi-Classifier Architecture

3.1 Problem Description

As previously stated in chapter 1, the significant progress in the automation of human behaviour, through the use of machine learning classifiers, lead to an increase of accuracy requirements and complexity for practical applications. The amount of power consumed for machine learning classifiers is so staggering that a few years ago, GPUs did not have enough power to run many of the algorithms. Although the re-purposing of the GPUs gave the performance needed, they are not well suited for the task as most of the power consumed is dissipated.

To overcome the power consumption problem, a complete FPGA-based run-time reconfigurable infrastructure that supports the dynamic deployment of multi-classifiers accelerator cores is proposed. As already mentioned in chapter 1, reconfigurable hardware enables substantial speedups due to the ability to customize the hardware architecture to the specific parameters of the actual problem instances.

3.2 Proposed Mechanism

The main idea is to develop an architecture which accelerates machine learning classifiers, running on a GPP, by transparently mapping computationally intensive parts of their execution to reconfigurable hardware. With this strategy, the performance of the embedded application will be enhanced since the units will act as acceleration co-processors of the GPP. The generation of custom run-time reconfigurable hardware for the transparent binary acceleration of machine learning classifiers will be based on the approach proposed in [24].

3.2.1 Transparent Binary Acceleration

The transparent binary acceleration exploited in [24] consists of detecting computationally intensive parts of executed instructions, named Megablocks [25], and on transparently migrating the

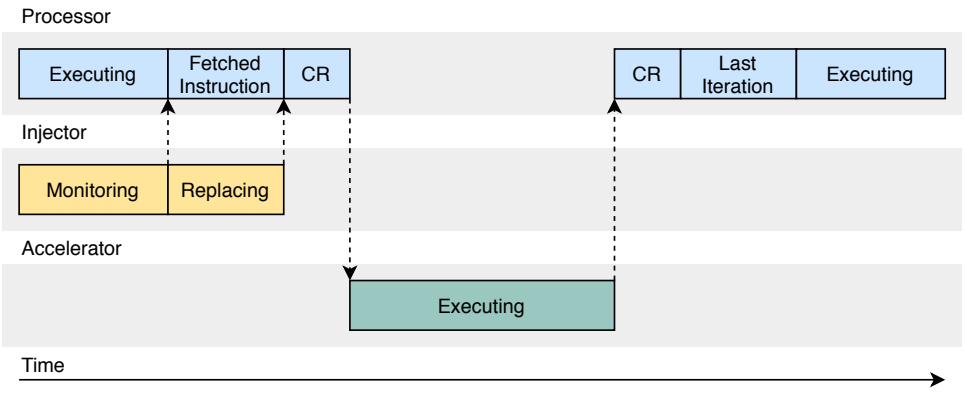

Figure 3.1: Transparently migration execution between processor and accelerator from [24]

execution of Megablocks to customized hardware accelerators. A Megablock is a pattern of instructions in the execution trace of a program and is extracted from execution trace. It represents a single-path control-flow execution forming a repeating instruction sequence with a single-entry and multiple-exit points. Usually, Megablocks are associated with loop behaviour [26].

Figure 3.1 summarizes the transparently migration execution between processor and accelerator exploited in [24]. The injector serves the purpose of monitoring the execution of the processor and modifies the content of the instruction bus. The processor executes the unmodified binary while the injector monitors its instruction address. The injector interferes when the address on the bus matches one of the addresses in its internal memory. These addresses are the start addresses of the Megablocks. The transparent migration starts with the injector replacing the fetched instruction, which results in an unconditional branch to a familiar memory position.

The branch target location contains a CR executed by the processor to communicate with the accelerator. When the processor executes the CR sends specific input operands to the accelerator from its register file. The accelerator executes the transparent loop after receiving all operands from the processor. The instructions from the Megablocks are implemented on the accelerator, including branch instructions which terminate the execution of that particular loop path. The last iteration is executed by the processor to allow the processor execution to follow the control flow, which triggered the end of the accelerator execution.

The processor idles while the accelerator is executing through a blocking wait. The results are fetched back into the processor's register file by executing the remainder of the CR. The last step is a branch back to the instruction address where the injector intervened. Execution resumes typically, and the accelerator can be called again further on. The tools and methodology required to detect the Megablocks and transform them into accelerator instances are exploited in chapter 4.

3.2.2 Hardware Generation

Figure 3.2 presents a generic architectural view of the accelerator-augmented system proposed in [24]. The system consists of the host processor, a local and/or external memory to hold the

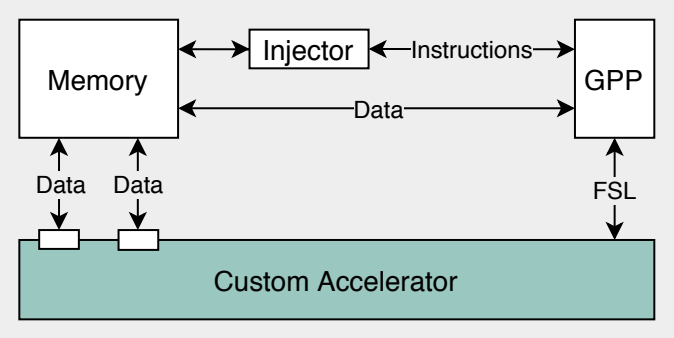

Figure 3.2: Transparent binary acceleration architecture from [24]

code and/or the data, the accelerator instance, and the injector module. The accelerator instance is connected to the host processor via a point-to-point FSL connection.

The injector's code is stored in the local memory and holds the automatically generated CRs used to establish the communication between the processor and the accelerator. When the application data is stored in the local memory, the processor and the accelerator share two local memory bus (LMB) BRAM interface controller cores to access it. When the application data is stored in the external memory, the processor uses a memory controller to which the accelerator also has access via a dual-port cache.

3.2.2.1 Target Platform

The host processor used in this work was the MicroBlaze soft-core processor. The target device for this implementation was a ZedBoard equipped with Xilinx Zynq-7000 multiprocessor system-on-chip (MPSoC) which integrates the software programmability of an ARM-based processor with the hardware programmability of an FPGA. It was used the Vivado Design Suite for IP integration design, synthesis, implementation, and bitstream generation and the Vitis Unified Software Platform for embedded software development.

The Zynq-7000 MPSoC architecture consists of two major sections: the processing system (PS) and the programmable logic (PL). The PS is a hard silicon dual-core ARM GPP consisting of: an application processor unit with two ARM Cortex-A9 processors, I/O peripherals and external memory interfaces. The PL consists of 7 series devices. PS boots from a selection of external memory devices, whereas PL is configured by and after PS boots. PS also provides clocking resources to PL.

The PS is equipped with two masters advanced extensible interface (AXI) channels to the PL and two slave channels mastered by the PL. The term master in this context means that the AXI channel is the initiator and can begin data exchanges, whereas a slave can only respond to arriving data. Also, there are four channels of high-performance AXI attachment points. These channels are slaves from the PS's perspective and are connected to the memory interface subsystem within

the PS. The purpose of these channels is to allow masters in the PL to initiate DDR memory transactions [27].

3.2.2.2 MicroBlaze + Zynq MPSoC Architecture

The MicroBlaze soft-core processor is highly configurable, allowing to select a specific set of features required by design. In this architecture, the MicroBlaze contained: 1. an extended floating-point unit (FPU) which enables add, subtract, multiply, divide, compare, convert, and square root instructions, 2. a hardware integer multiplier and divider, and 3. a reversed load/store and swap instructions. Also, data caches was enabled to improve performance when using external memory.

MicroBlaze is implemented with a memory architecture where instruction and data accesses are done in separate address spaces. Two LMB BRAM interface controller cores are used for connecting the MicroBlaze processor instruction and data ports to the on-chip BRAM. The amount of BRAM assigned in this architecture was 256KB. Additionally, a debug module core is connected to the MicroBlaze processor to enable JTAG-based debugging.

As previously stated in chapter 1, the use of machine learning classifiers is required to classify human activities due to the increasingly large amount of data present in human activity datasets. Besides, machine learning classifiers are complex algorithms which necessitate a significant amount of memory to perform. The considerable amount of memory required to compute the classification, for both instructions and data, impose the MicroBlaze to resort to the use of external memory.

Figure 3.3 presents the MicroBlaze plus Zynq MPSoC shared DDR memory bock diagram. The reference design has the PS generating the clocks for both itself and the MicroBlaze as well as sourcing the interrupts. The MicroBlaze can use these signals, but it can't influence or alter them. The easiest way for the MicroBlaze to access the PS DDR memory is to connect thought a high-performance AXI interface, as mentioned above. However, there are some caveats with this flow. The PL is configured before the PS DDR, so this would lead to the hanging of the MicroBlaze. The MicroBlaze needs to go to sleep state following the reset release, and only to be awoken and start executing code under control of the PS.

The extended multiplexed input/output (EMIO) general-purpose input/output (GPIO) was enabled. EMIO was used to exploit input/output peripheral (IOP) controllers available in PS to establish direct communication between PS and PL. The discrete ports were enabled in the MicroBlaze customization wizard. The discrete port reset_mode[0:1] was set to 0x01 to define the behaviour of the MicroBlaze. If it were 0x00, the MicroBlaze would start attempting to execute code immediately, at 0x01 the MicroBlaze will sleep until awoken. The discrete port wakeup[0:1] was connected to the GPIO_O pin in the Zynq MPSoC to woken the MicroBlaze [28].

Software Generation The target platform is a combination of the hardware components and software components, for instance, boot components. The first stage boot loader (FSBL) is responsible for loading the bitstream and configuring the PS at boot time. The MicroBlaze platform

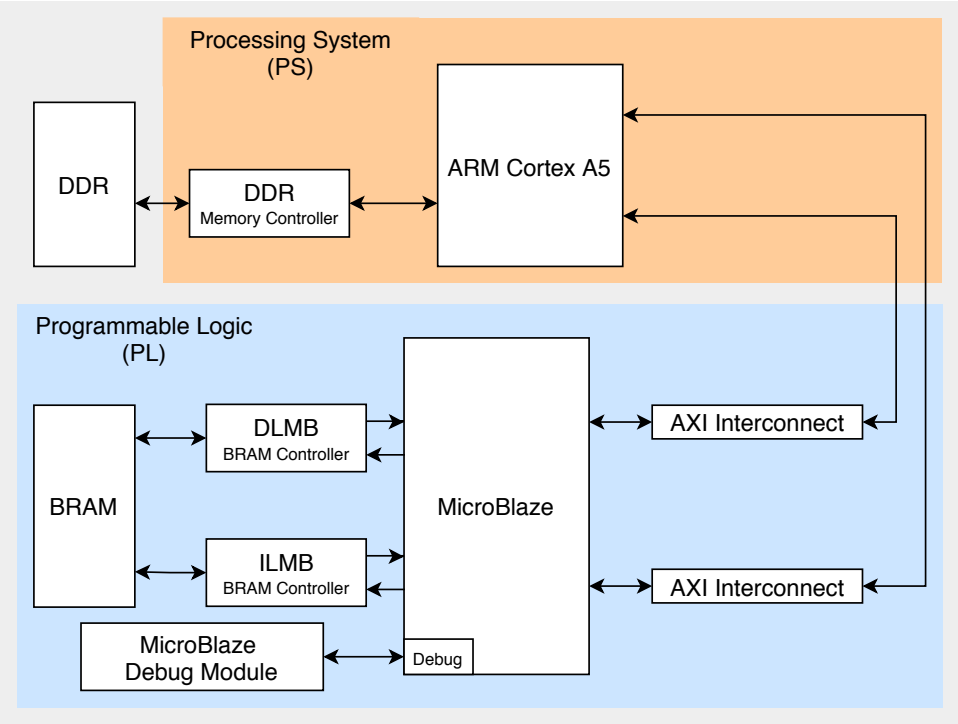

Figure 3.3: MicroBlaze + Zynq MPSoC shared DDR memory block diagram

needed to include the Zynq FSBL as a pre-built boot component to ensure the PS DDR initialization before the program execution. The Zynq FSBL template was used and updated to toggle the GPIO before it completes to wake up the MicroBlaze and start program execution.

3.2.2.3 Custom Loop Accelerator Architecture

Figure 3.4 presents a complete system with a MicroBlaze soft-core processor and the custom accelerator coupled as a co-processor. In this architecture, the MicroBlaze instruction cache needed to be disabled to allow the monitoring unit to interface with the instruction stream of the processor. Since the cache is internal to the processor, the interface between them cannot be probed. Herewith the processor would directly fetch trace instructions from the cache, and the migration mechanism would not function [24]. Since only data was stored in the external memory, the MicroBlaze instruction cache was disabled for both architectures.

The program instructions reside in the local BRAM memory. The MicroBlaze LMB BRAM controller core for instructions receives signals from one of the accelerator ports to enable code instruction accesses. In this architecture, the custom accelerator also needs to access the external memory other than the local BRAM memory. A system cache IP core was added to the design to establish the connection between the accelerator and the PS DDR. The system cache core can provide improved system performance for applications with repeated access of data occupying a specific address range. In this situation, external memory is used to buffer data during computations since the dataset exceeds the capacity of the MicroBlaze processor internal memory.

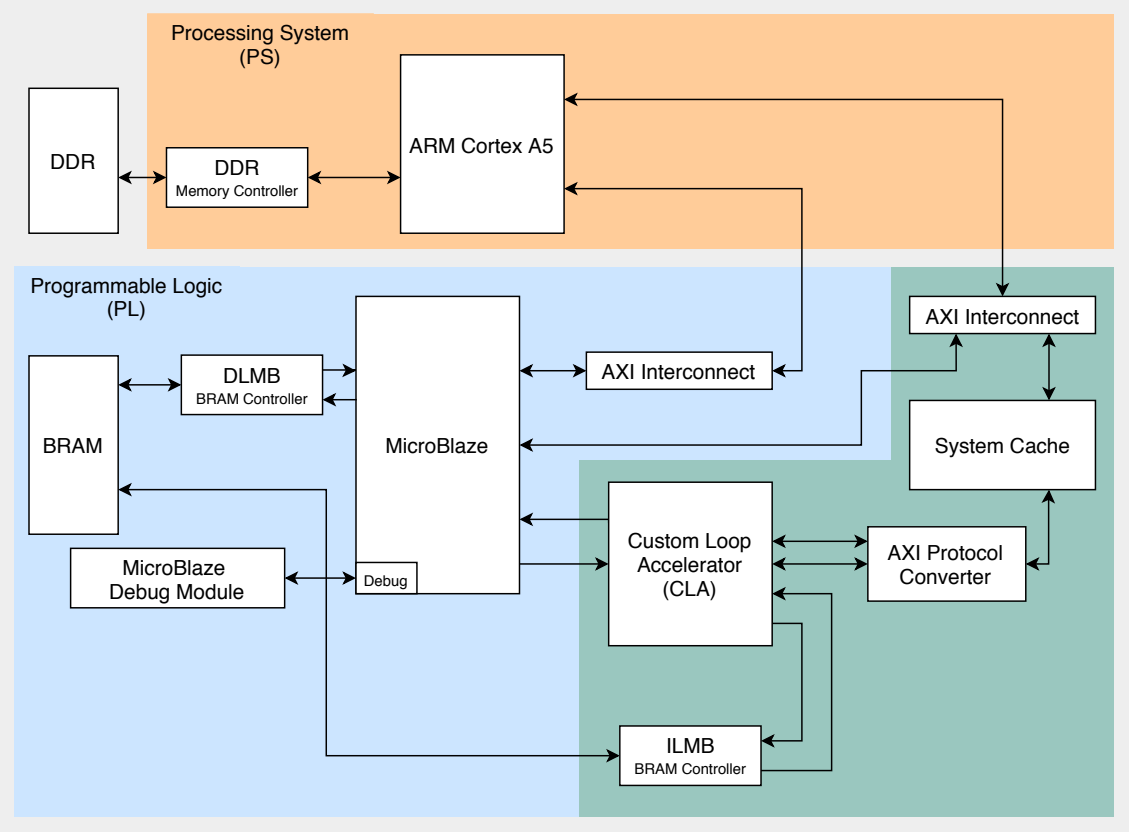

Figure 3.4: Custom loop accelerator system architecture block diagram

For both architectures an additional AXI timer/counter module, that is not represented in the architecture figures, was connected to the MicroBlaze's peripheral data interface. The AXI timer/-counter module provides an AXI4-lite interface to communicate with the host. Since the MicroBlaze does not have an internal counter this module is require to obtain the execution cycles of the application running.

3.2.2.4 Multi-Classifier Architecture

The previous custom loop accelerator architecture can be scaled to several MicroBlaze GPPs each one with one accelerator. This approach is hugely advantageous in the context of ensemble methods. As previously stated in chapter 2, an ensemble of classifiers can often overtake the performance of a single classifier. Besides, having an architecture with several MicroBlaze GPPs and custom accelerators to perform machine learning algorithms will exploit the ability of parallelism in just one device.

Figure 3.5 presents the proposed multi-classifier architecture. The system consists of three MicroBlaze GPPs, two of which perform the computation of the algorithms with the correspondent custom accelerator and the other one controls the execution and performs the ensemble of the classification. The MicroBlaze which performs the control, establishes a connection with the

others via a point-to-point FSL. The rest of the connections presented in the design are established as in the previous architectures. This simplicity of adding more processors indicates the easiness to scale this architecture for more classifiers.

Software Generation The MicroBlaze processor FSL macro descriptions were used to establish the software communication between the MicroBlaze GPPs. The macros used provide access to all of the functionality of the MicroBlaze FSL feature in one simple and parameterized interface. The control MicroBlaze starts the execution with a blocking `put` instruction. This instruction will tell to the other MicroBlaze GPPs to start execution the algorithms. Until they receive this notification, the MicroBlaze GPPs remain on hold. When the computation of the algorithm has finished, the result is sent to the control MicroBlaze through a blocking `get`. Until it receives both classifications, the control MicroBlaze remains on hold. Having both results, the MicroBlaze starts the performance of the ensemble method chosen and outputs the final classification.

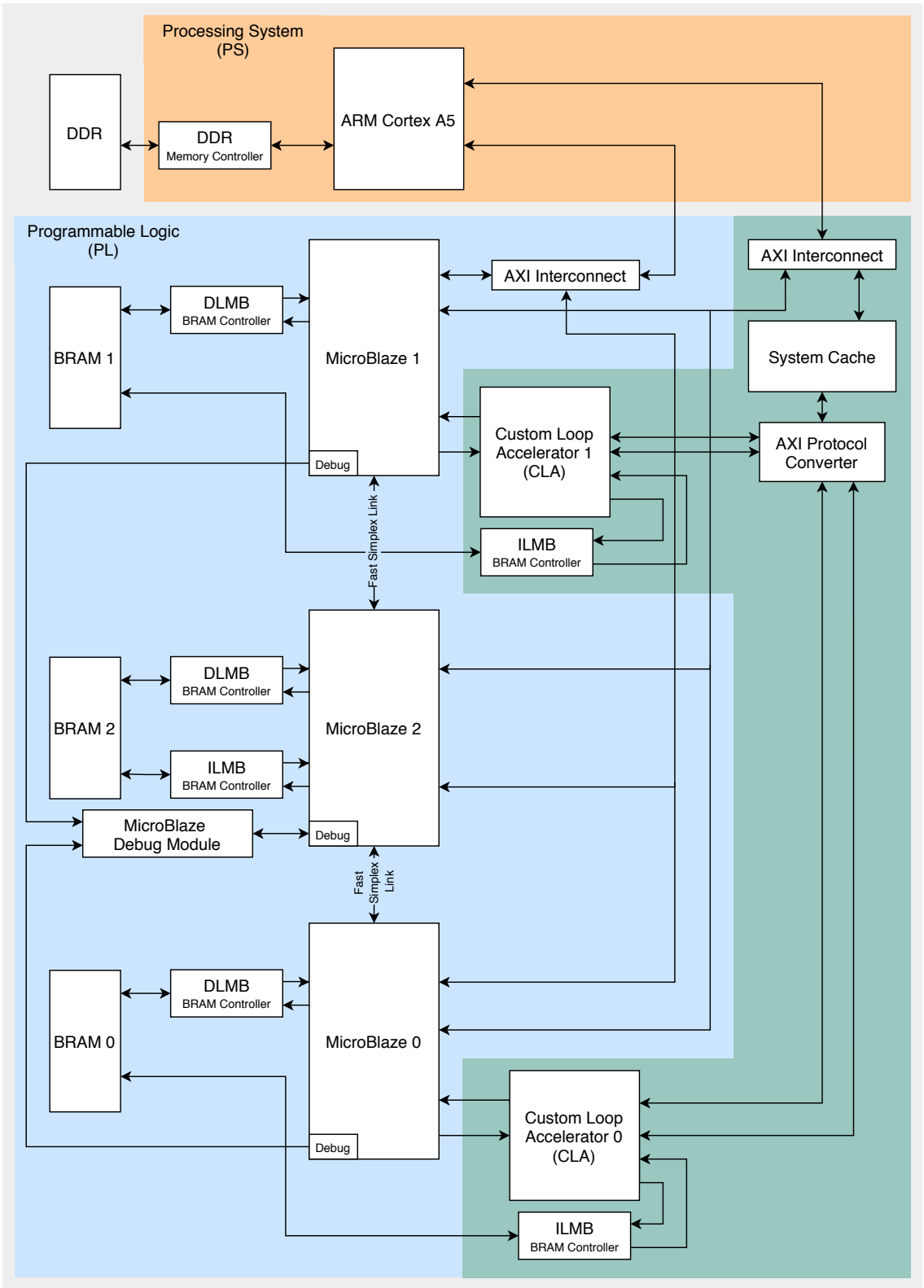

Figure 3.5: Multi-classifier architecture with two accelerator cores

Chapter 4

Implementation Overview and Experimental Evaluation

4.1 Generation of Customized Loop Accelerators

As per the general approach stated in chapter 3, the system architecture achieves transparent binary acceleration by relying on four main steps. The last two stages are the transparent migration and the acceleration of the execution to custom hardware, explained in the previous chapter. The first two steps consist of the detection of the Megablocks and their translation to customized accelerators, described in the present section. The tool flow for the generation of customized accelerators relies merely on the instruction traces obtained through simulation.

Figure 4.1 presents the tool flow with the main stages for the generation of customized loop accelerators. The flow begins with the application code. The application code is then compiled for the target platform architecture to generate an executable and linkable format (ELF) file. In order to do so, the bitstream of the MicroBlaze architecture proposed in chapter 3 figure 3.3 was generated and imported to Vitis. In Vitis, the target platform was created with the generated bitstream in order to compile the application code to it. The code was then compiled for the platform created. The result is an ELF file that when programmed in the FPGA executes the application code.

The generated ELF file is given as input to the Megablock extractor to start the detection of the Megablocks stage. In the following subsections, the two first stages for achieving transparent binary acceleration are described in more detail. Subsection 4.1.1 provides a brief description of the Megablock extractor appliance operation as well as an indication for the appliance's limitations. Subsection 4.1.2 explains the translation/scheduling process for the generation of the customized accelerator instance.

4.1.1 Megablock Extraction

The MicroBlaze executable is processed through the Megablock extractor appliance [25] to produce the frequent loop traces of the target application. The extraction process detects loop traces that respect the extractor detection criteria. The detection criteria include the maximum number of

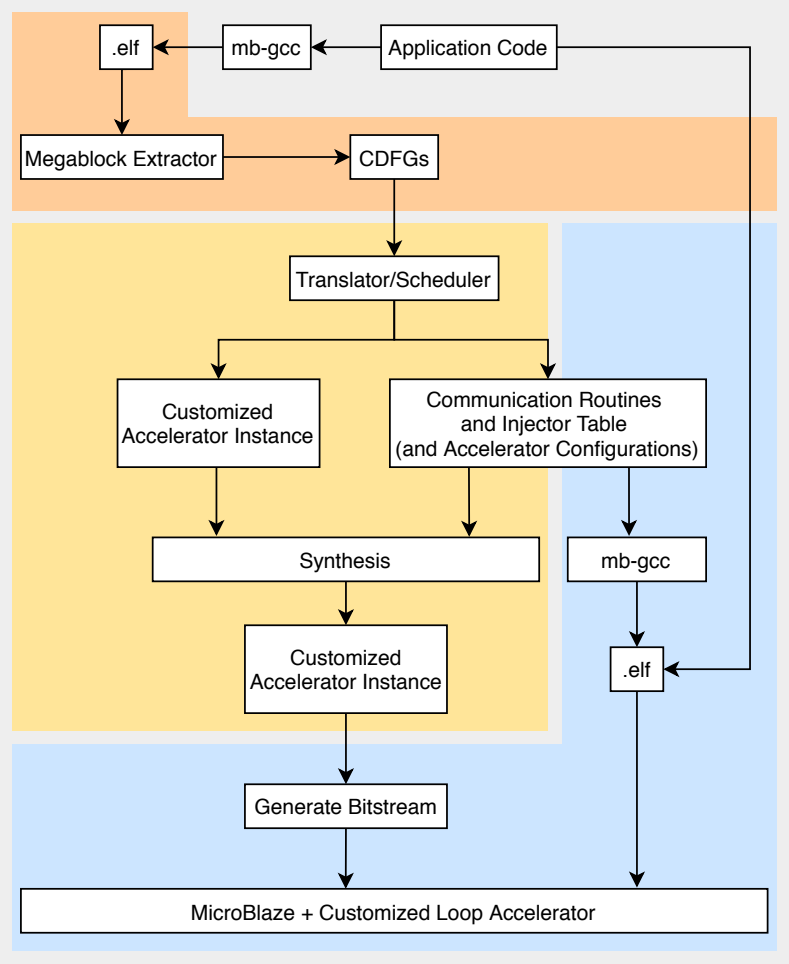

Figure 4.1: Tool flow for the generation of customized loop accelerators from [24]

instructions in the loop trace iteration, the minimum number of executed instructions by the loop trace iteration, the unrolling of inner loops which produce large unrolled traces, and other minor optimizations. Some of the detection criteria parameters are customizable in the appliance as well as the possibility to enable detection guards to detect Megablocks in a specific range of instruction addresses.

The extractor detection limitations condition the performance achieved by the accelerator. For instance, the extractor does not include indirect memory access [1] analysis, so it does not take into account data dependencies through memory. Additionally, the loop unrolling is performed up to the outermost loop. Moreover, the extractor cannot assume that coverage is fully representative since the loop trace is performed by simulation, functions may remain uncalled, or the counter of loop iterations may depend on data. Also, loop traces are detected for regions of code which are not candidates for acceleration, and the extractor is limited to the MicroBlaze instruction set. Knowing these limitations allows for optimizations in the application code.

[1] Indirect addressing is a scheme in which the address specifies which memory word or register contains not the operand but the address of the operand.

```
0x200011F4 addk r3, r5, r6      -> 0:add
0x200011F8 addk r4, r9, r6      -> 1:add
0x200011FC bslli r3, r3, 1026   -> 2:sll
0x20001200 bslli r4, r4, 1026   -> 3:sll
!0x20001204 imm 533              -> Removed
0x20001208 addik r3, r3, 25684 -> 4:add
!0x2000120C imm 16               -> Removed
0x20001210 addik r4, r4, 29780 -> 5:add
0x20001214 lwi r3, r3, 0        -> 7:load
0x20001218 lwi r4, r4, 0        -> 9:load
0x2000121C addik r7, r7, -1     -> 10:add
0x20001220 addik r6, r6, 1      -> 11:add
0x20001224 frsub r3, r4, r3     -> 12:frsub
0x20001228 fmul r3, r3, r3      -> 13:fmul
0x2000122C bneid r7, -56        -> 14:equalZero
0x20001230 fadd r8, r8, r3      -> 15:fadd
```

Listing 4.1: Example of extracted loop trace regarding the case study on subsection 4.2.2

The loops detected by the extractor contain a single entry point, which corresponds to the first instruction of the trace, and multiple exit points. The multiple exit points are the instructions that cause execution to escape from the pattern and follow a different path, such as, an `else if` within a `for` loop that produces two possible loop paths.

Regarding specific file outputs, the Megablock extractor produces text representations of the detected control-data flow graphs (CDFGs) constructed from the Megablock traces. The information present in these files includes the declaration of the input and output registers in the trace, the correspondent CDFG node that feeds them, the connections between nodes and some topological information. Listing 4.1 presents an example of extracted loop traces regarding the case study on subsection 4.2.2. Instructions starting with `!` were marked as not needed to be implemented.

The detected Megablocks are then translated/scheduled through the translator/scheduler tool proposed in [24]. The translation/scheduling process generates the HDL descriptions for the reconfigurable processing units (RPUs) and the injector and generates the CRs. The translator/scheduler receives a set of CDFGs and generates a customized accelerator instance by modulo-scheduling operations on pipelined or multicycle functional units.

4.1.2 Translation/Scheduling Process

The translation/scheduling process starts by performing some pre-processing steps for each CDFG. The pre-processing steps include the determination of the initiation interval [2] to establish control dependencies. The control dependencies ensure that the `branch` operations in an iteration execute before a new iteration is initiated and before any `store` operation belonging to the same iteration is performed.

[2] The number of clock cycles between the start times of consecutive loop iterations.

```
localparam NUM_PAR_CFGS = 32'd1;
localparam NUM_MEM_CHOICES = 32'd6;
localparam NUM_FU_CHOICES = 32'd6;
localparam NUM_MB_CHOICE = 32'd1;
localparam NUM_MB_INS = 32'd5;
localparam NUM_MB_OUTS = 32'd5;
localparam NUM_RF_REGS = 32'd11;
localparam NUM_FUS = 32'd6;

localparam NUM_ARRAY_INPUTS = 32'd13;
localparam NUM_ARRAY_OUTPUTS = 32'd6;

localparam [0 : (32*NUM_FUS)-1]
FU_ARRAY = {{`A_ADDC, `L_BLL, `B_EQU, `A_ADDC, `F_ADD, `F_MUL}};
```

Listing 4.2: Excerpt from HDL generation tool output regarding the case study on subsection 4.2.2

After pre-processing the CDFGs, each CDFG node is translated individually into functional unit placements and interconnections. The translation process attempts to find an existing functional unit in the current accelerator structure to reuse for the execution of that node. If it could not found any available functional unit, a new one is instantiated. The process of attempting to find available functional units involves a modulo-scheduling step proposed in [24].

Listing 4.2 presents an excerpt of the translation tool Verilog output for the extracted loop trace presented in listing 4.1 regarding the case study on subsection 4.2.2. It specifies some of the top level parameters of the accelerator instance along with the array of functional units. Listing 4.2 shows that, for this specific case, six different functional units are required: two arithmetic adds with carry `A_ADDC`, a barrel shift left logical `L_BLL`, a branch if equal `B_EQU`, a floating-point arithmetic add `F_ADD`, and a floating-point arithmetic multiplication `F_MUL`.

Communication Routine An additional tool proposed in [24] generates the CRs which implement the communication between the accelerator and the MicroBlaze. To achieve a transparent migration process to both the hardware infrastructure and the application programmer, the CRs are generated directly in MicroBlaze assembly and added to the application via a second compilation step. The tool is provided with a list of registers representing the CDFG inputs, a list of registers representing the CDFG outputs, and the start address of the Megablock trace. The tool outputs a sequence of MicroBlaze instructions which send operands and read results via a point-to-point FSL.

Listing 4.3 presents an example of tool-generated FSL-based CR regarding the case study on subsection 4.2.2. As a first approach, the implementation saves the entire register file and stack to memory. This step is required since the contents of the MicroBlaze's register file and stack could suffer unwanted modifications. The register file and stack contents are recovered after performing the `_wrapCaller` function. The `_wrapCaller` auxiliary function acknowledges that the accelerator's reconfigurable area is already configured with the desired partial bitstream. Subsequently, a single cycle `put` instructions send operands and a single cycle `get` instructions reads results.

```
unsigned int seg_0_opcodes[32] __attribute__ ((section (".CR_seg"))) = {
    0xf9e1ffc4, // Saving stack
    0xb0002000,
    0xb9fc9ac8,
    0x20000000,
    0xe9e1ffc4,
    0x3021ff80, // Call "_wrapCaller" function
    0xb0002000,
    0xb9fc9ab8,
    0x20a00000,
    0xb0000001,
    0x20c00001,
    0x6c06c000,
    0xb0002000, // Restoring stack
    0xb9fc9b48,
    0x20000000,
    0xe9e1ffc4,
    0x6c08c000, // Putting liveins
    0x6c07c000,
    0x6c09c000,
    0x6c06c000,
    0x6c05c000,
    0x6cc00000, // Getting control
    0xbc060010,
    0x6cc00000,
    0xb0002000,
    0xb64011f4,
    0x80000000,
    0x20000000,
    0x6c600000, // Getting liveouts (blocking gets)
    0x6c800000,
    0x6cc00000,
    0x6ce00000,
    0x6d000000,
    0xb0002000, // Return Jump
    0xb64011f4,
    0x80000000,
};

void __attribute__ ((section(".pr_seg"))) _wrapCaller(unsigned int i) {
    putfsl(1, 0);
    return;
}
```

Listing 4.3: Example of tool-generated FSL CR regarding the case study on subsection 4.2.2

Since the `get` instruction is blocking, the MicroBlaze idles waiting for the accelerator to output a single result.

If the accelerator were not configured with the desired bitstream, a reconfiguration routine would write the partial bitstream to the reconfigurable area of the accelerator thought an ICAP peripheral. For the presented case of study, the accelerator was already configured with the desired bitstream since only one Megablock is considered. For an application with multiple Megablocks,

```
// Communication routine location
localparam CR_NUM = 1;
localparam [0 : 32*(CR_NUM)-1] CR_ADDR = {32'h2000a287};

// Megablock start and end addresses
localparam [ 0 : 32*(CR_NUM)-1] CR_START_ADDR = {32'h200011f4};
localparam [ 0 : 32*(CR_NUM)-1] CR_NUM_EXITS = {32'h1};

localparam TOTAL_EXITS = 1;
localparam [ 0 : 32*(TOTAL_EXITS)-1] CR_END_ADDR = {{32'h20001234}};
```

Listing 4.4: Example of the Verilog include file regarding the case study on subsection 4.2.2

the translator/scheduler tool proposed in [24] would generate one accelerator configuration per CDFG along with a single configuration file with aspects relative to the static region. On this wise, there would be as many partial bitstreams as CDFGs. All bitstreams would be placed into a memory file system (MFS) that would be used to initialize the memory at boot time.

Both the sequence of MicroBlaze's instructions and the `_wrapCaller` auxiliary function are placed at a specified address, according to the custom linker script and via the respective section attribute. A custom linker script is required so that the re-compilation stage do not change the address of the trace loops and registers. Also, the injector needs to know the location address of the CR's in memory.

Injector Listing 4.4 presents an example of the small Verilog file produced for the injector. This file includes the start address of the Megablock trace and the location of the CR. It is also produced a read-only memory with the configuration information that the injector sends to the accelerator. Herewith, the injector controls the execution of the MicroBlaze by migrating the execution to the accelerator when the start address of the translated Megablock is detected.

According to [24], the injector operation consisted on: 1. monitoring for a Megablock's start address match, 2. when a match is found it replaces the fetched instruction with an unconditional jump to the same address, 3. when the match address is found again it injects instructions consisting on an unconditional jump to the respective CR, 4. idling during accelerator execution, 5. ignoring the next occurrence of the matched address since the last Megablock iteration is performed on software, and 6. returning to monitoring state.

After producing the translator/scheduler output files, the `.vh` files and the `.mem` file are used to generate the customized loop accelerator. The files are imported to a baseline Vivado project provided by [24] to be synthesized. After the synthesis is complete, the source files and information from the project are packaged to create a new IP for the Vivado IP catalogue. The created IP is then used in the custom loop accelerator architecture proposed in chapter 3 figure 3.4. The `.c` and `.ld` files produced by the translator/scheduler process are imported to the embedded software

platform. As previous stated, those files include the CRs between the MicroBlaze and the accelerator and the linker script for placing the content in the proper memory place. The application is recompiled using the custom linker script.

4.2 Case Studies

One machine learning classifier was implemented to evaluate the proposed architecture. The machine learning classifier was performed for both architectures presented in chapter 3 to measure the acceleration achieved by the proposed mechanism. The architecture without the accelerator shown in figure 3.3 and the architecture with the accelerator instance given in figure 3.4. The number of execution cycles of the classifier application was measured for both architectures. The acceleration was obtained with those results.

Subsection 4.2.1 introduces the human activity recognition dataset used to perform the classification as well as the method used to organize the data into a format favourable to the classifier implementation. Subsection 4.2.2 describes the implementation of the machine learning classifier, the outputs obtained on each stage of the generation of the customized loop accelerator and the implementation results obtained.

4.2.1 Dataset

As already mentioned in chapter 1, the machine learning classifier implemented in hardware will be tested for the human activity recognition environment. A standard human activity recognition dataset is the "Human Activity Recognition Using Smartphones Data Set, UCI Machine Learning Repository" available in [29]. The data was collected from a group of 30 subjects within an age bracket of 19-48 years. Each subject performed a set of six activities: walking, walking upstairs, walking downstairs, sitting, standing, and laying when wearing a smartphone on their waist. The movement data captured was 3-axial linear acceleration (accelerometer data) and 3-axial angular velocity (gyroscopic data) at a constant rate of 50Hz (i.e. 50 data points per second).

The dataset was randomly partitioned into two sets, where 70% of the subjects were selected for generating the training data and 30% the testing data. This partitioned resulted in a training set with 7352 feature vectors and a testing set with 2947 feature vectors. For each record the authors provided: a triaxial acceleration from the accelerometer and the estimated body acceleration, a triaxial angular velocity from the gyroscope, a 561-feature vector with time and frequency domain variables, the correspondent activity label and an identifier of the subject who performed the experiment. The sensor signals (accelerometer and gyroscope) were pre-processed by applying noise filters and then sampled in fixed-width sliding windows of 2.56 seconds (i.e. 128-time steps) and 50% overlap. These windows of data correspond to the windows of engineered features (rows), one row of data has 128 times 9 (i.e. 1152) elements.

The signals were stored by the authors in a directory under the train and test subdirectories. Each axis of each signal was stored in a separate file, meaning that each of the training set and the

```
# Load a single file as a numpy array
def load_file(filepath):
        dataframe = read_csv(filepath, header = None,
        delim_whitespace = True)
        return dataframe.values

# Load a list of files into a 3D array of [samples, time steps, features]
def load_group(filenames, prefix = ''):
    loaded = list()
        for name in filenames:
            data = load_file(prefix + name)
                loaded.append(data)
        # Stack group so that features are the 3rd dimension
        loaded = dstack(loaded)
        return loaded
```

Listing 4.5: Python code for loading the input signal data for a given group

testing set had nine input files and one output file to load. The method proposed in [30] was used to cluster all the files as a single array.

The loading of those files was batched into groups by specifying the consistent directory structures and file naming conventions. Each of those files was loaded as a `NumPy` array. The `load_file()` function presented in listing 4.5 loads the dataset given the file path and returns the loaded data as a `NumPy` array. Then function `load_group()` loads the data for a given group (train or test) into a single three-dimensional `NumPy` array, where the dimensions are samples, time steps and features.

The `load_dataset_group()` function presented in listing 4.6 loads the input signal data and the output data for a single group through the use of the consistent naming conventions. The output data is an integer number which defines the assigned class. The function `load_dataset()` one hot encodes these class integers to make the data more suitable for fitting multi-class classification machine learning algorithms. This function returns `trainX`, `testX`, `trainy` and `testy` corresponding to the arrays of the input signal data and the output data respectively. Those arrays are then used as inputs for the following machine learning classifier.

4.2.2 K-Nearest Neighbours

As stated in chapter 2, the KNN classification is one of the most fundamental and straightforward classification methods. It was the chosen algorithm to be implemented in the context of hardware accelerator classifiers due to its simple implementation. The C code was based on the one available on [31]. Some significant changes were made to simplify even more the code and optimize the performance of the Megablock extractor.

The implemented algorithm received as input the feature vector `p` to be classified. Then, it calculates the Euclidean distance between the feature vector `p` and the training vectors `q`. The distance result for each training vector is stored in an array `d`. When every distance is calculated,

```
# Load a dataset group, such as train or test
def load_dataset_group(group, prefix = ''):
    filepath = prefix + group + '/Inertial Signals/'
        filenames = list()
        # Total acceleration
        filenames += ['total_acc_x_'+group+'.txt',
        'total_acc_y_'+group+'.txt', 'total_acc_z_'+group+'.txt']
        # Body acceleration
        filenames += ['body_acc_x_'+group+'.txt',
        'body_acc_y_'+group+'.txt', 'body_acc_z_'+group+'.txt']
        # Body gyroscope
        filenames += ['body_gyro_x_'+group+'.txt',
        'body_gyro_y_'+group+'.txt', 'body_gyro_z_'+group+'.txt']
        # Load input data
        X = load_group(filenames, filepath)
        # Load class output
        y = load_file(prefix + group + '/y_'+group+'.txt')
        return X, y

# Load the dataset, returns train and test X and y elements
def load_dataset(prefix = ''):
        # Load all train
        trainX, trainy = load_dataset_group('train',
        prefix + 'HARDataset/')
        trainX.reshape(-1, 1152)
        # Load all test
        testX, testy = load_dataset_group('test',
        prefix + 'HARDataset/')
        testX.reshape(-1, 1152)
        # Zero-offset class values
        trainy = trainy - 1
        testy = testy - 1
        # One hot encode y
        trainy = to_categorical(trainy)
        testy = to_categorical(testy)
        return trainX, trainy, testX, testy
```

Listing 4.6: Python code for loading the input signal data and the output data for a single group

a comparison between them is performed to identify the K smallest distances, corresponding to the K nearest neighbours. Subsequently, the labels of the points in the array with the K smallest distances `dk` are stored in another array `c`. The array `l` has the labels of each training vector. The array with the labels of the K nearest neighbours `c` is then sorted, and the mode is performed. The result corresponds to the label that is going be attributed as the classification of the feature vector.

Listing 4.7 and 4.8 present two excerpts from the code of the KNN algorithm implementation. The function `knn_search()` implements the K nearest neighbour search where the Euclidean distance to every neighbour is calculated and the K closer neighbours are found. It also calls the `mode()` function given as input the array with the K nearest neighbours labels and the length of the array. The `mode()` function calculates the mode, as the name implies. The `qsort()` function is called to sort the array and the `compare_int()` function is given as input. The

```
// Performing the K nearest neighbour search
int knn_search(int K, float p[]) {
    // Calculating the Euclidean distance to every neighbour
    for(int i = 0; i < 7352; i++) {
        for(int j = 0; j < 1152; j++)
            d[i] = d[i]+((p[j]-q[i][j])*(p[j]-q[i][j]));
    }
    // Finding the K smallest distances
    for(int i = 0; i < 7352; i++) {
        float max = 0;
        int update = 0;
        for(int j = 0; j < K; j++) {
            if(dk[j] > max) {
                max = dk[j];
                update = j;
            }
        }
        if(dt[update] > d[i]) {
            dk[update] = d[i];
            // Corresponding the distances to labels
            c[update] = l[i];
        }
    }
    return mode(c, K);
}
```

Listing 4.7: Excerpt from the code of the KNN algorithm implementation

`compare_int()` function performs the comparison between two integers, as the name also implies.

As previously stated, the code available in [31] was extensively simplified to optimize the performance of the Megablock extractor. Local and global variables replaced the datatype structures, the debug prints were deleted, and the dynamically allocated variables were replaced for statically allocated variables. The type of memory allocation was changed since the exact size and type of memory were known. Regarding the MicroBlaze instructions, this change will eliminate the jump to the memory address where the memory allocation is performed when declaring the variables. Thus, without the function call, the Megablock extractor could easily detect the loop traces.

Another performed modification was the simplification of the Euclidean distance calculation. In Cartesian coordinates, if `p` and `q` are two points in Euclidean 1152-space, then the Euclidean distance `d` from `p` to `q` is given by the square root of the sum of the squared difference of the `p` and `q` dimensions. Since the calculated distances are only used to found which are the K smallest, the square root does not need to be performed.

The next step after completing all the simplifications to the code was to found the optimal K value to achieve the maximum accuracy of the model. The code with the previous dataset was executed in software with the K parameter varying since it depends upon the data. For this particular dataset, the optimum K was 4 with the correspondent accuracy of 63.62%. With the previous steps completed, the code was compiled for MicroBlaze to generate the ELF file. The

```
//Calculating the mode
int mode(int *values, int num_values) {
    int current_counter = 1, max_count = 1, max_index = 0, i;
    // Sorting the K nearest neighbours labels
    qsort((void*)values, num_values, sizeof(int), compare_int);
    // Calculating the mode of the K nearest neighbours labels
    for(i = 1; i < num_values; i++) {
        if(values[i-1] == values[i])
            current_counter += 1;
        else if(current_counter > max_count) {
            max_count = current_counter;
            max_index = i-1;
            current_counter = 0;
        }
        if(current_counter > max_count) {
            max_count = current_counter;
            max_index = i;
        }
    }
    return values[max_index];
}

// Comparing two integers
int compare_int(const void *v1, const void *v2) {
    int n1 = *(int*)v1;
    int n2 = *(int*)v2;
    if(n1-n2 > 1)
        return 1;
    else if(n1-n2 < -1)
        return -1;
    return n1-n2;
}
```

Listing 4.8: Excerpt from the code of the KNN algorithm implementation

generated ELF file was given as input to the Megablock extractor to detect the frequent loop traces.

For the personalized parameters in the Megablock extractor appliance, no limitation were imposed on the trace size and loop unrolling was performed on a per-case basis. The results presented in listing 4.9 showed that the extractor managed to find a Megablock with 90.4% coverage. This Megablock corresponded to the inner loop of the Euclidean distance calculation. It calculates the distance for each feature of the feature vector to each feature of one of the training vectors. Ideally, the outer loop, the one that scrolls through the training vectors, would be a better option, but the extractor cannot unroll the inner loop.

Listing 4.10 presents an output of the extractor with some details of the Megablock found. It shows the start and the exit addresses, a liveness analysis with the live-ins and live-outs registers, some details on the Megablock execution, the operations computed by the Megablock and information about the graph stats. The number of iterations performed corresponds to the number of iterations of the `for` cycle that this Megablock covers. The maximum instruction-level parallelism (ILP) is 4, which means that, for the set of the trace instructions, a maximum of 4 instructions can

```
Execution Cycles: 1323402601
Executed Instructions: 619075122
CPI: 2.13770922779869
Total Megablock Coverage: 99.7%

-- Individual Megablock Coverage (Total) --
Megablock 0x200011F4-1: 90.4%
Megablock 0x20000234-1: 9.2%
Megablock 0x20001158-1: 0.1%

-- Megablock Detection Options --
Executed Instructions Threshold: 50
Trace Unit: BasicBlock
Maximum Units in Pattern: 32
Unroll Inner Loops: true

-- Megablock Iterations --
Megablock '0x200011F4-1' total/average/max: 34173558/1,149/1149
Megablock '0x20000234-1' total/average/max: 17372575/17,372,575/17372575
Megablock '0x20001158-1' total/average/max: 22980/1,149/1149
```

Listing 4.9: Output of the Megablock extractor

be executed simultaneously. It also presents the average number of instructions executed for each clock cycle (IPC) which is 1.75.

The extractor also produces a graph description language (DOT) that could be visualized with a graph visualization software. Figure 4.2 presents the CDFG representation of the trace instructions of the detected Megablock produced by the extractor. The graph presents 7 levels of nodes as stated in the graph stats presented in listing 4.10. With this representation, it is possible to obtain a better visualization of all the graph operations implemented and their level placement. It is also possible to observe the live-ins and live-outs registers and identify the ones with feedback.

In an ideal scenario, the execution time would be fully proportional to the code coverage. For this scenario, an 90.4% coverage corresponded to an approximately 10× acceleration. However, as already mentioned in subsection 4.1.1, the coverage obtained is not entirely viable since the loop trace is performed by simulation. Also, the extractor does not consider the number of cycles to access external memory. It always assumes that all the storage is local. For this example, the data is stored in external memory, so it will take significantly more cycles that were not considered.

It was obtained a 1.08× acceleration corresponding to 8%. These results were well below expectations. There are two main reasons for these results. The first one is that the Megablock performs an inner loop. The extractor detects a large coverage percentage because this inner loop is performed for all of the training vectors, which consists of the significant computation of the algorithm. Since the outer loop is not included in the Megablock, for each training vector, the execution will migrate from the processor to the hardware. This migration involves CRs which introduce some delay. The delay is usually negligible, but in this situation, it occurs to many times.

The second one, and probably the most relevant to the accelerator performance, consisted of

```
Has 0 memory store instructions
Has 2 memory load instructions
#Exit Points: 1
StartPc: 0x200011f4

-- Liveness Analysis --
Live-ins without feedback: r5, r9
5 live-ins (r5, r6, r7, r8, r9)
5 live-outs (r3, r4, r6, r7, r8)

-- Megablock Execution --
#iterations: 1149
#Megablock instructions (assembly): 16
#Graph operations: 14
#Megablock instructions x iterations: 18384
#Graph operations x iterations: 16086
#Megablock Cycles: 35
#Megablock Cycles x iterations: 40215

-- Megablock Operations --
total: 14
arithmetic: 6
load: 2
shift: 2
zeroComparator: 1
floating: 3

-- Graph Stats (using latencies) --
#levels: 7
ILP Max: 4
ILP Min: 1
CPL: 16
IPC: 1.75

-- Exit Addresses --
Exit1: 0x20001234
```

Listing 4.10: Detailed information of the Megablock

the external memory accesses. In addition to the coverage not being entirely accurate, the accelerator instance has a different mechanism to access the data stored in the external memory than the MicroBlaze. Since the acceleration result is obtained by comparing the number of execution cycles for both architectures, with and without the accelerator instance, this differentiation will influence.

When the accelerator instance is not considered, all the external memory accesses are performed by the MicroBlaze. The MicroBlaze uses its internal data cache. Since the accelerator does not have its own cache, an system cache IP modulo was used to allow a fair comparison between the two. Although the parameters for both caches, such as data width and size of word cache-lines, were the same, the communication protocol with the external memory differs. The MicroBlaze caches over AXI4 interface and the accelerator caches over AXI4-Lite interface. Therefore, the

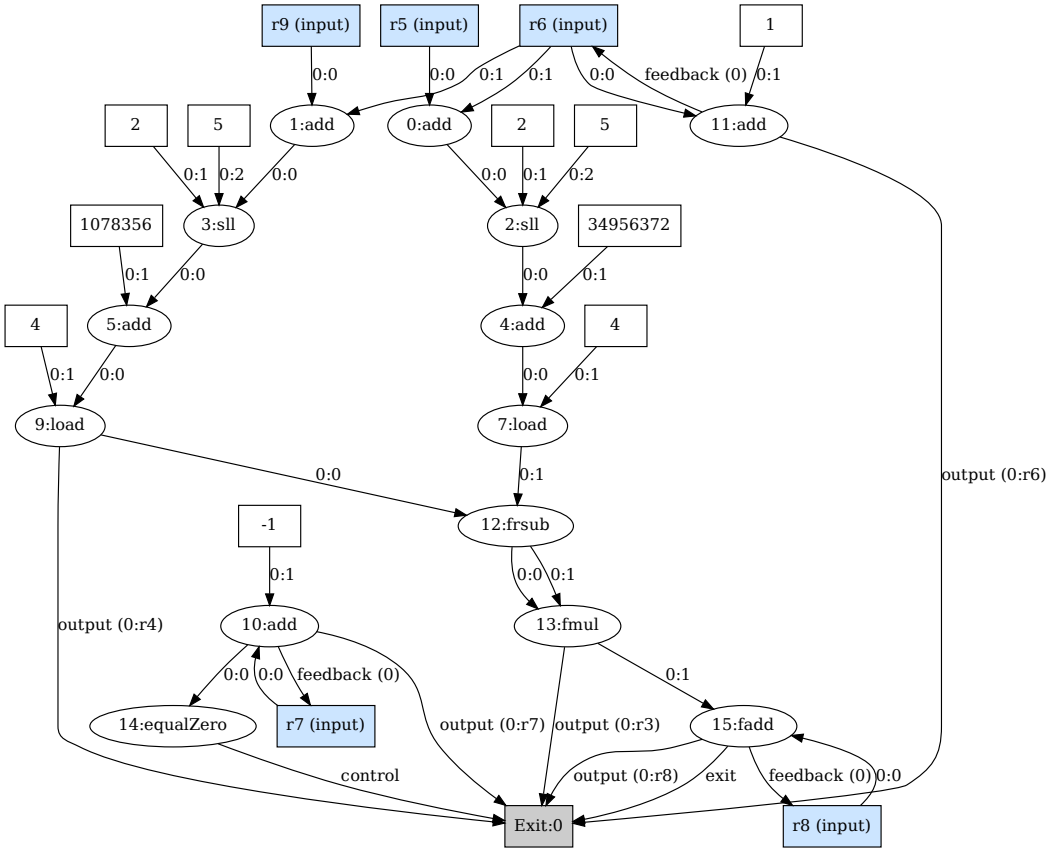

Figure 4.2: CDFG representation of the trace instructions of the detected Megablock

use of an AXI interconnect IP between the accelerator instance and the system cache IP is required to convert the protocol AXI4-Lite to AXI4.

An experiment was performed to understand if the different communication protocols will influence the number of cycles to read data from the external memory. An integrated logic analyzer (ILA) IP core was added to both architectures to monitor the internal signals of the designs. The signals of the external memory connection are connected to the ILA core clock and probe inputs. These signals, attached to the probe inputs, are sampled at design speeds and stored using on-chip BRAM. After the design is loaded into the FPGA, the Vivado logic analyzer software was used to set up a trigger event for the ILA measurement. After the trigger occurs, the sample buffer was filled and uploaded into the Vivado logic analyzer. The data was viewed by using the waveform window.

In the waveform data, the signals corresponding to the read transfer with the external memory (moving data from the slave to the master) were observed. The number of cycles to read data from the external memory was obtained by calculating the difference between when the master drives the read address channel `ARADDR[31:0]` and when the slave drives the read data channel

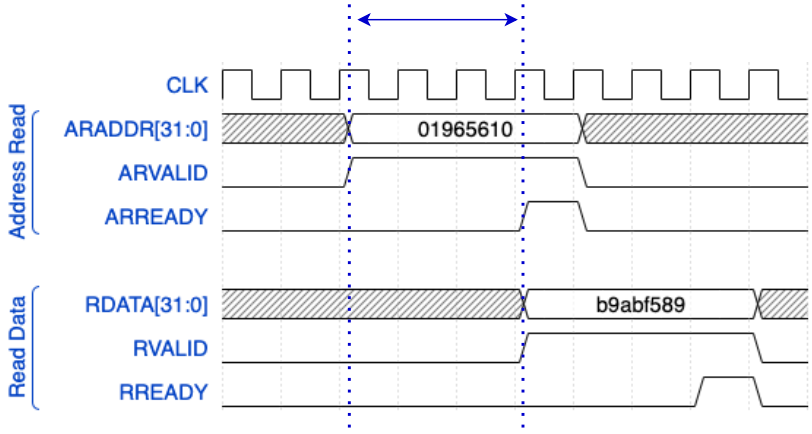

Figure 4.3: Read transaction timing from 2[32]

`RDATA[31:0]`. Figure 4.3 presents an example of the read transition timing.

The architecture without accelerator takes 20 cycles, and the architecture with accelerator takes 28 cycles to read data from the external memory. The communication protocol conversion must have introduced the extra 8 cycles. The computation of this algorithm requires for each feature vector to classify the distance calculation between each feature of the training vector. As previously stated, the training vector consists of 7359 feature vectors, each one with 1152 `float` features. This sizes will give a total of approximately 34M bytes. Since it is not possible to store so many bytes in the cache, for each feature vector to classify, the cache needs to access the external memory to fetch the same data. Duo to the frequent fetch, the extra 8 cycles will introduce some overhead.

The accelerator has an internal timer that could be used to complement the accuracy of the previous analysis. With this internal timer, it outputs the accelerator run time, the time that the accelerator is waiting for the external memory, the total number of execution cycles performed by the accelerator and the waiting time introduced due to the jump to the CR. By observing those results, it was concluded that the accelerator is 21% of its execution waiting for the external memory.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This work had the goal to devise and evaluate a complete FPGA-based, run-time reconfigurable architecture that supports the dynamic deployment of multiple accelerator cores. To achieve this goal, an available flow to generate custom run-time reconfigurable hardware for transparent binary acceleration was used.

The proposed architecture was implemented, and its performance quantified while running on hardware, a ZedBoard target device equipped with Xilinx Zynq-7000 MPSoC. The results obtained during this research, show that the generated custom loop accelerator could be used to compute complex machine learning classifiers with an increasingly large amount of data. The proposed design has the ability to scale the system architecture to several processing elements, each one with the correspondent accelerator as a co-processor. Theoretically, it would be possible to have up to 16 accelerated classifiers running since the limitation is imposed by the number of stream ports available on the MicroBlaze. Naturally, the amount of resources available on the target device would influence that number.

Several issues were also identified in the proposed approach. For one, the access to the external memory imposes a limitation in the accelerator performance. The algorithm implemented in hardware was I/O bounded. The time it takes to complete the algorithm computation was imposed by the time spent waiting for memory accesses (I/O operations) to be completed. The bottleneck occurs since the computation of the algorithm is simple and does not require many operands. Another is that the accelerator strongly depends on the detection criteria of the Megablock extractor. For this scenario, the detection limit of the loop unrolling not be performed up to the outermost loop impacted the acceleration performance. The overhead penalty in the communication processor-accelerator was emphasized because the outer loop needs to be performed on the processor. Thus, although the Megablock detected computed several iterations, each one of them requires a new GPP-accelerator communication and migration of the execution flow.

5.2 Future Work

The implementation developed during this research come across some issues of the proposed mechanism. Taking into account the implementation stage aimed by this current work, the following improvements could be implemented to achieve better performance and exploit the scalability of the proposed architecture.

1. Further characterization of the external memory access bottleneck. A target device with more resources could be used to implement a MicroBlaze with a larger internal memory. Thus, the data would be stored in the internal memory, and no external memory was needed. A comparison study between the performance of the accelerator with and without external memory access would produce some interesting results. Another implementation issue still concerning in the external memory context is to study the possible interfaces to stream data between the external memory and the accelerator.

2. Research machine learning classifiers without an I/O bounded limitation that still suit well the classification of human activities. A classifier where its computation requires more quantity and diversity of operands for the computation to dominate the memory accesses.

3. Investigate machine learning classifiers with multiple Megablocks take advantage of the dynamic reconfiguration ability of the custom loop accelerator. Study the resources improvements obtained and associated overheads.

4. Implement the multi-classifier architecture proposed with a set of machine learning classifiers. Take some conclusions about the classification accuracy improvements and study the time improvements in performing the classifiers in parallel.

5. Investigate the possibility to accelerate the processor which computes the merging of the classifications of all the classifiers. This assumes that a complicated ensemble method must be implemented, which would be worth accelerating.

6. Develop a study for a set of different classifiers to exploit the necessity of acceleration. For instance, if in the set of classifiers one of them takes a lot longer to compute maybe the others do not have the need to be accelerated since they would have to wait for the longest one to finish.

The use of a multi-classifier architecture that supports the deployment of multiple accelerator cores to classify human activities is an unexplored subject in the literature. It brings a vast of benefits not only for the hardware acceleration field but also to improve performance on the use of machine learning classifiers.

References

[1] Deepika Singh, Erinc Merdivan, Ismini Psychoula, Johannes Kropf, Sten Hanke, Matthieu Geist, and Andreas Holzinger. Human Activity Recognition Using Recurrent Neural Networks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10410 LNCS, pages 267–274. 4 2017. URL: http://link.springer.com/10.1007/978-3-319-66808-6_18http://arxiv.org/abs/1804.07144http://dx.doi.org/10.1007/978-3-319-66808-6_18, doi:10.1007/978-3-319-66808-6{_}18.

[2] Bo Yao, Hani Hagras, Mohammed J Alhaddad, and Daniyal Alghazzawi. A fuzzy logic-based system for the automation of human behavior recognition using machine vision in intelligent environments. *Soft Computing*, 19(2):499–506, 2 2015. URL: http://link.springer.com/10.1007/s00500-014-1270-4, doi:10.1007/s00500-014-1270-4.

[3] Lev Kirischian. Introduction to Reconfigurable Computing Systems. In *Reconfigurable Computing Systems Engineering*, pages 1–24. CRC Press, Boca Raton : Taylor & Francis Group, 2016., 12 2017. URL: https://www.taylorfrancis.com/books/9781482282245/chapters/10.1201/9781315374697-1, doi:10.1201/9781315374697-1.

[4] Thomas G. Dietterich. Ensemble Methods in Machine Learning. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1857 LNCS, pages 1–15. 2000. URL: http://link.springer.com/10.1007/3-540-45014-9_1, doi:10.1007/3-540-45014-9{_}1.

[5] Charmi Jobanputra, Jatna Bavishi, and Nishant Doshi. Human Activity Recognition: A Survey. *Procedia Computer Science*, 155(2018):698–703, 2019. URL: https://doi.org/10.1016/j.procs.2019.08.100https://linkinghub.elsevier.com/retrieve/pii/S1877050919310166, doi:10.1016/j.procs.2019.08.100.

[6] Koldo Basterretxea, Javier Echanobe, and Ines del Campo. A wearable human activity recognition system on a chip. In *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*, number November, pages 1–8. IEEE, 10 2014. URL: http://ieeexplore.ieee.org/document/7115600/, doi:10.1109/DASIP.2014.7115600.

[7] Athanasios Lentzas, Andreas Agapitos, and Dimitris Vrakas. Evaluating state-of-the-art classifiers for human activity recognition using smartphones. In *IET Conference Publications*, volume 2019. IET, Stevenage, UK, 2019. URL:

https://www.engineeringvillage.com/share/document.url?mid=inspec_189c94f516c726b6787M5dc210178163167&database=ins, doi:10.1049/cp.2019.0098.

[8] Eunju Kim, Sumi Helal, and Diane Cook. Human activity recognition and pattern discovery. *IEEE Pervasive Computing*, 9(1):48–53, 2010. doi:10.1109/MPRV.2010.7.

[9] Leif Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009. URL: http://www.scholarpedia.org/article/K-nearest_neighbor, doi:10.4249/scholarpedia.1883.

[10] S. Rasoul Safavian and David Landgrebe. A Survey of Decision Tree Classifier Methodology. *IEEE Transactions on Systems, Man and Cybernetics*, 21(3):660–674, 1991. URL: http://ieeexplore.ieee.org/document/97458/, doi:10.1109/21.97458.

[11] S.K. Pal and Sushmita Mitra. Multilayer perceptron, fuzzy sets, and classification. *IEEE Transactions on Neural Networks*, 3(5):683–697, 1992. URL: http://ieeexplore.ieee.org/document/159058/, doi:10.1109/72.159058.

[12] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9(4):611–629, 8 2018. URL: https://doi.org/10.1007/s13244-018-0639-9https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9, doi:10.1007/s13244-018-0639-9.

[13] Tehseen Zia and Usman Zahid. Long short-term memory recurrent neural network architectures for Urdu acoustic modeling. *International Journal of Speech Technology*, 22(1):21–30, 3 2019. URL: http://link.springer.com/10.1007/s10772-018-09573-7, doi:10.1007/s10772-018-09573-7.

[14] Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multiclass support vector machines. *IEEE transactions on neural networks*, 13(2):415–25, 3 2002. URL: http://ieeexplore.ieee.org/document/991427/http://www.ncbi.nlm.nih.gov/pubmed/18244442, doi:10.1109/72.991427.

[15] Elie Track, Nancy Forbes, and George Strawn. The End of Moore's Law, 3 2017. URL: http://ieeexplore.ieee.org/document/8226056/http://ieeexplore.ieee.org/document/7878957/, doi:10.1109/MCSE.2017.25.

[16] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57, 6 2020. doi:10.1145/3361682.

[17] Frede Blaabjerg. *Control of power electronic converters and systems*. Elsevier, 1 2018. doi:10.1016/C2017-0-04756-0.

[18] Zhe Hao Li, Ji Fang Jin, Xue Gong Zhou, and Zhi Hua Feng. K-nearest neighbor algorithm implementation on FPGA using high level synthesis. *2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology, ICSICT 2016 - Proceedings*, (3):600–602, 2016. doi:10.1109/ICSICT.2016.7998989.

[19] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable FPGAs. *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 15–24, 2017. doi:10.1145/3020078.3021741.

[20] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *journal=arXiv preprint arXiv:1602.02830,*, 2 2016. URL: https://github.com/MatthieuCourbariaux/http://arxiv.org/abs/1602.02830.

[21] Lie Wang and Wu Feng-yan. Dynamic partial reconfiguration in FPGAs. In *3rd International Symposium on Intelligent Information Technology Application, IITA 2009*, volume 2, pages 445–448, 2009. doi:10.1109/IITA.2009.334.

[22] Hanaa M. Hussain, Khaled Benkrid, and Huseyin Seker. Dynamic partial reconfiguration implementation of the SVM/KNN multi-classifier on FPGA for bioinformatics application. *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBS*, 2015-Novem:7667–7670, 2015. doi:10.1109/EMBC.2015.7320168.

[23] Nuno M.C. Paulino, Joao Canas Ferreira, and Joao M.P. Cardoso. Dynamic Partial Reconfiguration of Customized Single-Row Accelerators. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(1):116–125, 2019. doi:10.1109/TVLSI.2018.2874079.

[24] Nuno Miguel Cardanha Paulino. Generation of Custom Run-time Reconfigurable Hardware for Transparent Binary Acceleration. (June), 2016.

[25] João Bispo, Nuno Paulino, João M.P. Cardoso, and João Canas Ferreira. Transparent runtime migration of loop-based traces of processor instructions to reconfigurable processing units. *International Journal of Reconfigurable Computing*, (February), 2013. doi:10.1155/2013/340316.

[26] Nuno Paulino, João Canas Ferreira, João Bispo, and João M P Cardoso. Transparent Acceleration of Program Execution Using Reconfigurable Hardware. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015. doi:10.7873/DATE.2015.1122.

[27] Bill Kafig. How a MicroBlaze Can Peaceably Coexist with the Zynq SoC. *Xcell Journal*, (82), 2013. URL: www.xilinx.com/xcell/.

[28] MicroZed Chronicles: Combining MicroBlaze & the Zynq MPSoC - Hackster.io, 2018 (accessed 08/06/2020). URL: https://www.hackster.io/news/microzed-chronicles-combining-microblaze-the-zynq-mpsoc-94501295dd3.

[29] UCI Machine Learning Repository: Human Activity Recognition Using Smartphones Data Set, (accessed 15/04/2020). URL: https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones.

[30] 1D Convolutional Neural Network Models for Human Activity Recognition, 2018 (accessed 15/04/2020). URL: https://machinelearningmastery.com/cnn-models-for-human-activity-recognition-time-series-classification/.

[31] Chris Dilger. C implementation of a K-Nearest Neighbour algorithm, 2017 (accessed 24/03/2020). URL: https://github.com/cdilga/knn-c.

[32] AXI Reference Guide, (accessed 28/07/2020). URL: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.