ASHESI UNIVERSITY

Using Commercial FPGAs as External Accelerators for Artificial Neural Networks in Embedded Applications

CAPSTONE PROJECT

B.Sc. Computer Engineering

Oracking Amenreynolds

2020

ASHESI UNIVERSITY

Using Commercial FPGAs as External Accelerators for Artificial Neural Networks in Embedded Applications

CAPSTONE PROJECT

Capstone Project submitted to the Department of Engineering, Ashesi University, in partial fulfilment of the requirements for the award of Bachelor of Science degree in Computer Engineering.

Oracking Amenreynolds

2020

DECLARATION

I hereby declare that this capstone is the result of my own original work and that no part of it has been presented for another degree in this university or elsewhere.

Candidate’s Signature:

…………………………………………………………………………………………

Candidate’s Name:

Oracking Amenreynolds

Date:

29th May, 2020

I hereby declare that preparation and presentation of this capstone were supervised in accordance with the guidelines on supervision of capstone laid down by Ashesi University

Supervisor’s Signature:

…………………………………………………………………………………………

Supervisor’s Name:

…………………………………………………………………………………………

Date:

…………………………………………………………………………………………

Acknowledgements

I wish to express my gratitude to my supervisor Dr. Nathan Amanquah, whose continuous encouragement and advice have helped me throughout this journey.

Abstract

Artificial Neural Networks (ANNs) is a branch of Machine Learning that has seen recent widespread adoption for solving computational problems that seem impractical to solve with traditional algorithmic approaches. ANNs have achieved high accuracy on tasks such as facial recognition, object detection and speech recognition. And recently, ANNs have also seen applications in embedded systems, where it has been used to train robots to learn from their environment and cameras to detect faces in a crowd. However, achieving reasonable performance on a traditional microcontroller can be difficult since ANNs are computationally expensive. This paper investigates the possibility of using a Field Programmable Gate Array (FPGA) as an external accelerator for a microcontroller unit. The aim is for the combined performance of the FPGA and the microcontroller, for running the ANN, to be better than just the microcontroller. For the tested neural network, the results show that the combined system with the FPGA and microcontroller runs at more than twice the speed of a system with just a microcontroller.

Table of Contents

Chapter 1: Introduction

Machine learning, as an approach to artificial intelligence, has seen widespread adoption within the past two decades[1]. In conventional algorithm design, a human would have to infer patterns from data and subsequently write a set of rules, in the form of a program, for a computer to execute. However, machine learning systems provide a framework that allows the computer to implicitly infer these rules directly from the data[2]. In fact, several artificial intelligence developers have realized that for complex problems, like image recognition, it is easier for the machine to infer patterns than it is to manually design a program[1].

With that said, there are several types of algorithms within machine learning, including linear classifiers, quadratic classifiers, Bayesian networks and neural networks [3]. However, of interest to this paper are neural networks – artificial neural networks to be precise. Artificial neural networks (ANNs) refer to a paradigm in machine learning where computational units, referred to as neurons, are connected in a pattern [4], to perform distributed computing. Unlike other machine learning algorithms, ANNs are designed to operate like biological neurons in the human brain[5].

Neurons in neural networks can be organized in several ways to produce varying behaviour. For instance, neurons can be grouped into layers, with neurons in each layer processing information and feeding their outputs to neurons in the next layer. This technique of stacking multiple layers of neurons is referred to as deep learning[6].

Since 2006, deep learning neural networks (DLNNs) have made several breakthroughs in solving computationally complex tasks. DLNNs have produced remarkable results for computer vision, speech recognition and natural language processing tasks[7]. Since 2009, developers have leveraged DLNNs to win many official pattern recognition competitions, using it to achieve the first superhuman visual pattern recognition results in some areas[8]. Industrial applications of DLNNs are vast, including applications in financial forecasting[9] and medical diagnosis systems[10], [11].

One such area where DLNNs are beginning to see increased adoption is in embedded system design. An embedded system refers to a combination of computer hardware and software designed to perform a particular task[12] – for instance, the microcontroller unit within a security camera or an exploration rover. And within the field of embedded systems, there has been research to leverage neural networks to achieve intelligence within these systems.

In the field of embedded systems, neural networks have been leveraged to design autonomous robot driving[13], build smart home systems[14] and achieve real-time face tracking and identity verification[15]. On space exploration systems, DLNNs can be leveraged to improve the autonomy of rovers to compensate for high latencies during communications[16]. These are just a few ways neural networks are being leveraged in embedded systems. However, running neural networks on embedded systems presents a unique set of challenges that may not be encountered when running them on a desktop or server.

Generally, embedded systems must be designed within some constraints. These design constraints include but are not limited to production cost, processing power, memory and development cost[12]. Incorporating DLNNs into an embedded system design, while satisfying all these constraints, can be a challenge.

Considering just processing power, the ATMEGA4809, released in 2018, is an 8-bit single-core microprocessor that operates at a maximum frequency of 20 MHz. This is the microprocessor used in the Arduino Uno WiFi rev 2, a development board for embedded system design, popular amongst hobbyists. Yet, it does not have a fraction of the power delivered by the Intel i3-330M, which was released in 2010 for consumer laptops. The i3-330M is a dual-core 64-bit processor that operates at 2,133.33 MHz. The i3-330M, despite being released 8 years before, operates at more than 100 times the frequency of the ATMEGA4809.

Evidently, there is a gap between the performance of embedded systems and laptops/desktops/servers. As such, a DLNN that runs well in a desktop environment may not run satisfactorily on an embedded system without extra effort from the designer. A good example, is the work done in [16], where neural networks are leveraged to increase autonomy of robots. The neural networks are used in a reinforcement learning algorithm to allow the robots learn the dynamics of their environment without having explicit models pre-programmed. It states that “autonomous robotic systems running such algorithms tend to be slow in performance and energy inefficient when implemented on a traditional microcontroller”[16]. For this reason, exploring various architectures for accelerating DLNNs in embedded systems is not only beneficial but necessary in some cases.

In the wider search for DLNN accelerators, not restricted to embedded systems, one such architecture that has been explored is field programmable gate array (FPGA)[17]. An FPGA is an off-the-shelf development board that allows the implementation of hardware functionality via high-level design paradigms. FPGAs are part of a wider group of programmable logic devices and can be considered as reconfigurable integrated circuits[18]. FPGAs present a good case for accelerators of DLNNs in embedded systems for a few reasons. Firstly, they are energy efficient; they provide good performance per watt[18]. This makes them a good option for battery-operated devices. Secondly, they allow the implementation of low-level parallelism and pipelining, which makes them a good choice for compute-intensive applications such as DLNNs[18]. Furthermore, if we consider FPGA designs as an alternative to designing software for general-purpose processors, then they can provide better development cost to performance ratios in certain applications[19].

For these reasons, multiple research papers have explored FPGAs as accelerators for DLNN. However, a good number of these papers test their designs on relatively expensive FPGA boards. Some papers leverage certain components such as, hard floating-point units, which may not be available on cheaper FPGA boards. The cost of the boards used in these research papers makes their designs less attractive for small embedded system projects with tight cost constraints. Furthermore, few of these papers provide an end-to-end development framework for utilizing their designs, making it less appealing to developers with little to no VHDL knowledge.

1.1 Objective

The objective of this paper is to extend the designs already explored but to utilize a low tier FPGA board as an external accelerator chip in an embedded system. The FPGA will act as a coprocessor, working alongside a microcontroller unit. Further, he final design will be incorporated in an end-to-end framework that will make it more appealing for embedded system designers to use.

The FPGA board that will be used for testing designs will be the Xilinx Basys 3 Artix-7. This board is described on the Xilinx webpage as an entry-level FPGA development board for student and beginners. As such, it presents a good case for a “low tier” FPGA board. The Xilinx Basys3 FPGA will be used as a DLNN accelerator for an Arduino Uno, a development board popular amongst embedded system hobbyists.

Chapter 2: Related Literature

The objective of this paper is to design an FPGA accelerator for a microcontroller unit. Achieving this requires sufficient understanding of core concepts including the structure of neural networks and FPGAs. This Chapter will serve to review the preliminary concepts and important considerations for designing the proposed system. It will also review similar work that has been done.

2.1 Neural Network Overview

This section will be focused on describing the general structure of deep neural networks[6]. Exploring the structure of DLNNs will help identify which operations can be optimized and how the overall structure will translate onto an FPGA board.

Neural networks are composed of small computational units called neurons. In deep learning, these neurons are organized into layers. Each neuron in a given layer accepts as input, information from some or all the neurons in the previous layer. It then performs some set of arithmetic operations on these inputs to produce a value. This value is then fed into an activation function which adds some non-linearity to the value. The output of the activation function is then passed to the next layer as the output of that single neuron.

There are different types of layers in deep learning, such as fully connected layers, convolutional layers and softmax layers. The type of layer determines how neurons are organized within that layer, and how they connect and process information from neurons in the previous layer. Similarly, there are different types of activation functions. However, for the scope of this paper, we will focus on neural networks composed of fully connected layers with rectified linear unit (ReLU) activation functions. These two are sufficient for a proof of concept design. And the final design can always be extended to include other layers.

2.1.1 Operation of Neural Networks

A neuron is simply a node that holds a number. The number it holds is referred to as its activation. If we consider a neural network with fully connected layers, we have a structure that looks like Figure 2.1.1.

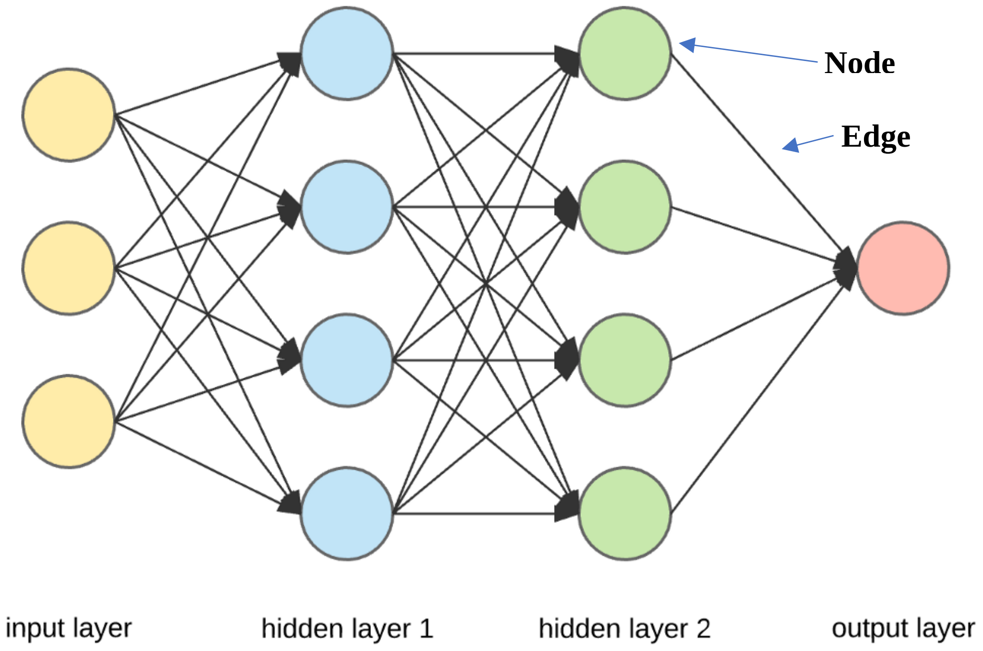

Figure 2.1.1[20]: Structure of sample neural network, consisting of fully connected layers

A neuron connects to another neuron via an edge. In a fully connected layer, each neuron in one layer has edges to all neurons in the previous layer. The first layer to a neural network is normally referred to as the input layer. Usually, the neurons in the input layer encode real-world information. For instance, the activations of the three input neurons could represent three ultrasonic sensor readings or the value of three pixels in an image. These values are then fed into the network to make some inference.

Neurons in subsequent layers compute their values based off their connections (edges) to neurons in the previous layer. The edges that connect one neuron to another have weights associated with them. This is illustrated in Figure 2.1.2.

2.1.2 Single Neuron Analysis

The activation that a given neuron holds is denoted by a_k^L, where L is the index of the layer the neuron is in and k is the index of the neuron within that layer. The weight associated with the edge that connects the neuron in one layer, a_j^{L-1}, to the neuron in the next layer a_i^L is denoted by $w_{i,j}^L$. Figure 2.1.2 illustrates this by highlighting all neurons in the first layer and a single neuron in the second layer.

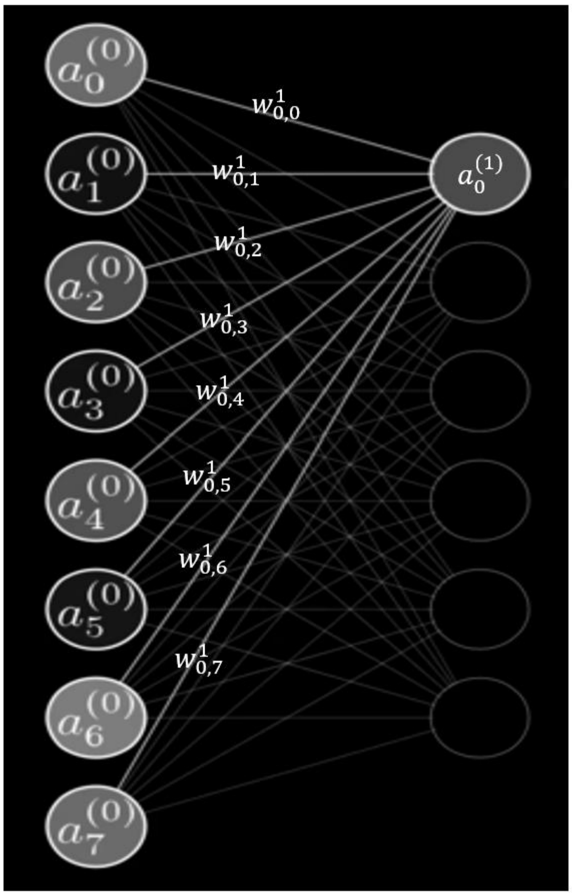

Figure 2.1.2[21]: Neural network structure with the focus on a single neuron in the second layer

To compute the value of the neuron a_0^1, the activation of each neuron in the first layer is multiplied by the weight of the edge that connects it to a_0^1. The result of all the multiplications are then summed together:

$$a_0^1 = (w_{0,0}^1 \cdot a_0^1 + w_{0,1}^1 \cdot a_1^1 + w_{0,2}^1 \cdot a_2^1 + w_{0,3}^1 \cdot a_3^1 + w_{0,4}^1 \cdot a_4^1 + w_{0,5}^1 \cdot a_5^1 + w_{0,6}^1 \cdot a_6^1 + w_{0,7}^1 \cdot a_7^1) + b_0^1$$

b_0^1 is the bias associated with the neuron a_0^1. The bias is a constant number that is added to the result of the summation. It acts as a threshold for the neuron. Generally, to compute the activation of any neuron, the formula is:

$$a_i^L = \left(\sum_{k=0}^{m-1} w_{i,k}^L \cdot a_k^{L-1} \right) + b_i^L$$

where, $m = Number\ of\ neurons\ in\ layer\ L - 1$.

The expression, $\sum_{k=0}^{m-1} w_{i,k}^L \cdot a_k^{L-1}$, in the equation however, can be described as a multiply and accumulate (MAC) operation. It begins with a value of zero, and for each value of k, it multiplies the kth weight by the kth activation and adds the result to its current value. If all the activations of the previous layer are stored in a vector, and all the weights of the edges that connects the stored activations to the neuron a_i^L are also stored in another vector, the summation can be explicitly written as a MAC operation as follows:

$$\sum_{k=0}^{m-1} w_{i,k}^L \cdot a_k^{L-1} = MAC\left(\begin{bmatrix} w_{i,0}^L \\ \vdots \\ w_{i,k}^L \end{bmatrix}, \begin{bmatrix} a_0^{L-1} \\ \vdots \\ a_{m-1}^{L-1} \end{bmatrix} \right)$$

Therefore, our initial equation for the activation of a neuron becomes:

$$a_i^L = MAC\left(\begin{bmatrix} a_0^{L-1} \\ \vdots \\ a_{m-1}^{L-1} \end{bmatrix}, \begin{bmatrix} w_{i,0}^L \\ \vdots \\ w_{i,k}^L \end{bmatrix} \right) + b_i^L \tag{1}$$

This produces the output of a single neuron in the a given layer. However, the value a_i^L is normally not the final activation. It is usually passed through an activation function such as the ReLU function. The ReLU is a piece-wise function that takes an input, x. If x is less than 0, the output of the ReLU function is 0. Otherwise, the output of the ReLU function is x. In this sense, the ReLU acts as an ideal diode. Therefore, we can rewrite equation (1) to include the ReLU function:

$$a_i^L = \sigma\left(MAC\left(\begin{bmatrix} a_0^{L-1} \\ \vdots \\ a_{m-1}^{L-1} \end{bmatrix}, \begin{bmatrix} w_{i,0}^L \\ \vdots \\ w_{i,k}^L \end{bmatrix}\right) + b_i^L\right) \tag{2}$$

Where σ is the ReLU function.

2.1.3 Layer Analysis

Applying equation (1), to calculate the activations all the neurons in "hidden layer 1" of Figure 2.1.1, gives:

$$a_0^1 = (w_{0,0}^1 \cdot a_0^1 + w_{0,1}^1 \cdot a_1^1 + w_{0,2}^1 \cdot a_2^1) + b_0^1$$

$$a_1^1 = (w_{1,0}^1 \cdot a_0^1 + w_{1,1}^1 \cdot a_1^1 + w_{1,2}^1 \cdot a_2^1) + b_1^1$$

$$a_2^1 = (w_{2,0}^1 \cdot a_0^1 + w_{2,1}^1 \cdot a_1^1 + w_{2,2}^1 \cdot a_2^1) + b_2^1$$

$$a_3^1 = (w_{3,0}^1 \cdot a_0^1 + w_{3,1}^1 \cdot a_1^1 + w_{3,2}^1 \cdot a_2^1) + b_3^1$$

As such, the output of all neurons in a layer can be computed through a series of parallel MAC operations. These set of expressions can be written as a single matrix expression:

$$\begin{bmatrix} a_0^1 \\ a_1^1 \\ a_2^1 \\ a_3^1 \end{bmatrix} = \begin{bmatrix} w_{0,0}^1 & w_{0,1}^1 & w_{0,2}^1 \\ w_{1,0}^1 & w_{1,1}^1 & w_{1,2}^1 \\ w_{2,0}^1 & w_{2,1}^1 & w_{2,2}^1 \\ w_{3,0}^1 & w_{3,1}^1 & w_{3,2}^1 \end{bmatrix} \begin{bmatrix} a_0^0 \\ a_1^0 \\ a_2^0 \end{bmatrix} + \begin{bmatrix} b_0^1 \\ b_1^1 \\ b_2^1 \\ b_3^1 \end{bmatrix} \tag{3}$$

In equation (3), The matrix that contains all the weights, is called the weight matrix. The vector that contains all the biases is called the bias vector. And the vector that contains all the activations of the previous neurons is the input vector. Despite having this form, the computation is still a series of MAC operations. If the first row of the weight matrix is denoted as R_0 and the input vector is denoted as A, then the first element of the resultant vector can be expressed as:

$$a_0^1 = MAC(R_0, A) + b_0^1$$

And in general:

$$a_k^L = MAC(R_\mathrm{k}, A) + b_k^L \tag{4}$$

Finally, the general expression to compute the output of all the neurons in any layer, L, can be expressed as:

$$\begin{bmatrix} a_0^L \\ \vdots \\ a_{n-1}^L \end{bmatrix} = \sigma \left(\begin{bmatrix} w_{0,0}^1 & \cdots & w_{0,m-1}^1 \\ \vdots & \ddots & \vdots \\ w_{n-1,0}^1 & \cdots & w_{n-1,m-1}^1 \end{bmatrix} \begin{bmatrix} a_0^{L-1} \\ \vdots \\ a_{m-1}^{L-1} \end{bmatrix} + \begin{bmatrix} b_0^L \\ \vdots \\ b_{n-1}^L \end{bmatrix} \right) \tag{5}$$

where:

$$n = Number\ of\ neurons\ in\ layer\ L$$

$$m = Number\ of\ neurons\ in\ layer\ L - 1$$

2.2 Embedded System Overview

From our television sets to microwave ovens and cars, embedded systems form an integral part of our lives. An embedded system refers to any computer system that is designed to perform a limited set of functions within a larger system. An example is the embedded system within a television set that allows the control of volume and switching of channels.

The field of embedded systems is continuously expanding as the world turns to small assistive devices. The increased popularity of smart watches is an example of this. These systems normally have tight development constraints. These constraints include: production cost, processing power, memory and development cost[12]. Embedded systems are normally deployed on microcontroller units, which are easy to interface with sensors and have low development costs.

2.2.1 Neural Networks in Embedded Systems

Neural networks have proved to be useful in modelling systems that are difficult to model or solve analytically. For example, describing an algorithm that can recognize images of handwritten characters can be particularly tasking. However, this is an introductory problem for neural network courses. Other more complex tasks such as facial detection, object detection and speech recognition are solved by neural networks.

A good number of the problems solved by neural networks, such as facial detection and speech recognition, involve the analysis of information collected about the real-world through sensors. Likewise, a good number of embedded system projects involve collecting and processing information about the real-world through sensors. Therefore, it is unsurprising that several papers have investigated the use of neural networks in embedded systems to aid in information processing.

Neural networks have also been considered in embedded systems for real-time facial recognition[15]. The result of this project is useful in areas of law enforcement and security, such as the airport where identity needs to be quickly verified. [16] also considers the use of neural networks to allow a rover to learn to navigate its surroundings without being explicitly programmed, which also has various applications in the wider context of robotics.

However, despite their benefits, neural networks are computationally expensive, making them difficult to use on microcontroller units, which have limited computational power. In fact, neither [15] nor [16] tests the final design on a microcontroller unit. Therefore, finding a way to efficiently run neural networks on microcontroller units, while reaping the benefits of using a microcontroller unit will be extremely beneficial.

2.3 Hardware Accelerators for Neural Networks

Due to the peculiar nature and perceived future importance of neural networks, various architectures have been explored for accelerating the execution of these models. Three of main contending architectures are FPGAs, Application Specific Integrated Circuits (ASICs) and Graphics Processing Units (GPUs). GPUs provide the architecture to perform fast floating-point operations in parallel, which allows the value of multiple neurons in a layer to be computed simultaneously. ASIC accelerators are designed at the hardware level to speed up operations of the network. This means computations can be sped up through low-level pipelining and parallelism. FPGA accelerators are like ASIC accelerators in the sense that they speed up computations through low-level pipelining and parallelism. The key difference is that FPGAs are usually slower than ASICs but offer the benefit of low development cost and reconfigurability.

In the case of accelerators for embedded systems, however, FPGAs are the strongest contenders. GPUs normally require more power to run than both FPGAs and ASICs. Their high-power requirement makes them a poor choice for embedded applications, which usually have tight power constraints. ASICs have a reasonable power consumption but have high development costs, making them impractical for small and medium scale embedded system projects with tight cost constraints. Furthermore, their lack of reconfigurability means updating architecture design involves creating a new set of ASIC accelerator chips. FPGAs on the other hand allow custom accelerators to be created in a short amount of time and run at a reasonable power consumption, making them a good choice for embedded applications. Furthermore, they can easily be reprogrammed if the target application changes. Consequently, this paper will explore the use of FPGAs as accelerators for embedded systems.

2.4 Field Programmable Gate Arrays

FPGAs are semiconductor devices that can be electronically reprogrammed to function as any digital circuit[22]. FPGAs are comparable to integrated circuits. However, whereas the function of an integrated circuit is fixed after production, the FPGA can have its function reprogrammed. As such, it offers hardware level speeds while providing software-like flexibility. Updating the function of an FPGA is as simple as downloading a new application bitstream. This makes FPGAs useful in applications where developers wish to minimize development time and cost but achieve reasonable gains in hardware. FPGAs are used in several applications including signal processing applications as well as the implementation of custom computing machines.

2.4.1 Structural Overview

In order to achieve flexibility and performance, FPGA architecture is designed to be intrinsically modular. The FPGA architecture comprises of three main elements: configurable logic blocks (CLBs), programmable routing channels and I/O blocks. The CLBs implement logic functions using elements like lookup tables and flip-flops. The programmable routing channels connect the logic functions produced by the CLBs. And the I/O blocks make off-chip connections possible, allowing information to be sent to and from the FPGA to another device[22]. These blocks together form the FPGA. The architecture of the FPGA is shown in Figure 2.4.1

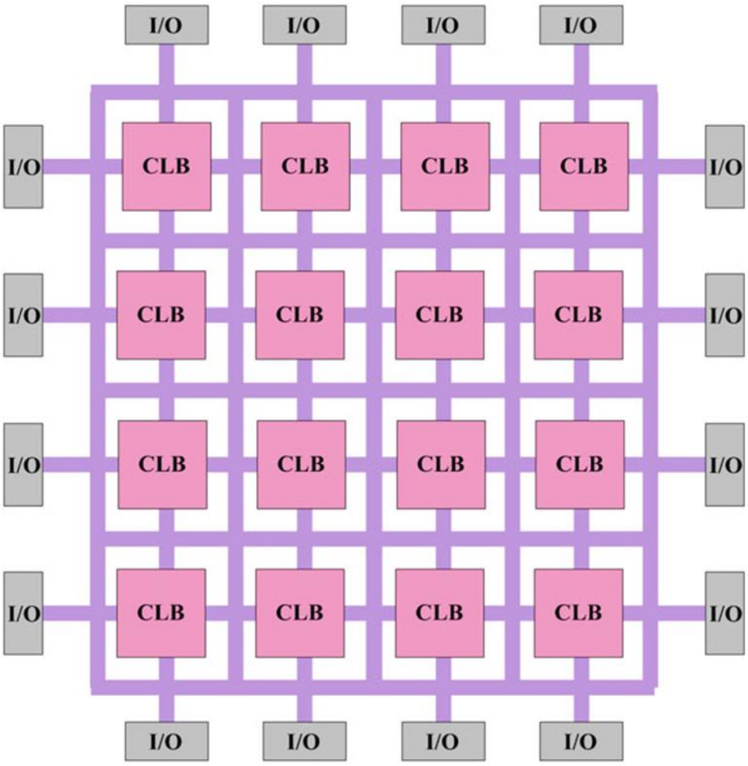

Figure 2.4.1: Overview of FPGA architecture[22]

2.4.2 Digital Signal Processor Unit

FPGAs have been used in several applications in the past. And across the numerous applications, there are common logical components that are implemented by designers in almost all projects. To prevent the hurdle of always reimplementing these components, FPGA manufacturers have implemented these components as special blocks that come with the FPGA board. An example of such a special block is the Digital Signal Processor (DSP). This block performs common operations such as addition, multiplication and accumulation. Furthermore, these special blocks can usually execute their tasks faster than components built from the fundamental FPGA blocks. Developers can, therefore, easily utilize these special blocks in their design.

The DSP block is particularly interesting because of its ability to perform the MAC operation efficiently. As was seen in section 2.1.3, evaluating the activation of neurons in a fully connected layer involves computing parallel MAC operations. This implies that DSPs can be leveraged to speed up this computation.

2.4.3 FPGA Design Language

FPGAs are usually programmed in high-level languages that allow the hardware architecture to be described like software. The difference, however, is that most of these high-level languages are not procedural like traditional programming languages for processors. They require some knowledge of digital circuit design.

The two main languages used are Verilog and VHDL. These languages are, for the most part, interchangeable. In this paper, however, all the proposed designs will be implemented in VHDL.

2.5 FPGA Accelerators

Prior research has explored the utilization of FPGAs as accelerators for deep learning. Most of these papers analyse and develop design strategies to optimize the core neural network operations on the FPGA. Some of these papers even provide an end-to-end framework that allows the neural network to be described in a high-level language, which automatically compiles to an FPGA design. This section will review the work done and indicate which ideas this paper intends to adopt and develop on.

2.5.1 DeepBurning

Designing FPGA accelerators requires substantial knowledge of FPGA design and profound knowledge of neural networks to generate an efficient accelerator architecture. These requirements make FPGA accelerators unfriendly to high-level developers and machine learning researchers. The DeepBurning project addresses this problem by designing an end-to-end framework for creating custom FPGA accelerators[23].

The framework accepts two inputs: a script, which gives a high-level description of the neural network design, and a file that describes the hardware constraints. The descriptive script for the network can be written manually or generated by Caffe, a python deep learning framework, which is more friendly to researchers. The constraints file describes the maximum available resources, such as the number of programmable logic elements that can be utilized. The framework then uses these two inputs to map the provided neural network to a custom FPGA design. This design can then be used to accelerate learning and inference for the neural network.

DeepBurning aims to accommodate a wide range of neural network designs. It does this by building up a library of configurable hardware components for the different neural network layers. For instance, there are components for fully connected layers and activation layers like the ReLU layer. For some components, certain parameters, such as the input size, is left to be configured at the compilation stage. When the descriptive script is passed to the framework, it identifies the different layers and fetches the appropriate components from the library. These components are configured and wired together to produce the final design.

2.5.2 A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks

In [24] the design of accelerators for neural networks is explored, with a focus on mobile devices. A system is developed that allows neural networks to be used in real-time applications with desirable performance and low energy consumption. The proposed system consists of two components: a host processor and a coprocessor. The coprocessor is designed on programmable logic and accelerates the core neural network operations. The host processor is a Central Processing Unit (CPU) that handles the compilation of the design as well as the operations that are not implemented on the co-processor. It also handles the transfer of data to the co-processor. The final design is tested on a Xilinx ZC706 platform which houses two ARM Cortex-A9 cores, 1GB of DDR3 and an FPGA. The ARM Cortex-A9 cores act as host processors and the FPGA is programmed to be the coprocessor.

2.5.3 Deep Learning with Limited Numerical Precision

Several research papers explore the parallel nature of neural networks to provide performance gains. However, not as often considered is the high error resiliency of neural networks and learning algorithms. Unlike traditional workloads, neural networks don’t require precise computations for learning to occur. In [25] an architecture is proposed that capitalizes on this idea, to achieve gains in computational speed and power efficiency. The final design is tested on an FPGA.

Typically, neural networks are trained with a floating-point data representation for model parameters such as weights and biases. Floating-point representation is appealing due to its dynamic range, allowing extremely large and small numbers to be stored with reasonable precision. This property is desirable since, during learning, updates to model parameters can have a wide range of values. In [25], the use of low precision fixed-point numbers in learning algorithms is considered. The use of low precision fixed-point presents two major benefits. Firstly, compared to floating-point units, computational units for fixed-point are usually faster, more energy-efficient and require fewer logic resources. Secondly, memory footprint and required bandwidth for data transfer are minimized with low precision representation. On the other hand, the low precision and lack of dynamic range of this form of data representation can cause significant loses in accuracy and sometimes prevent learning algorithms from ever converging.

It is possible, however, to reap the benefits of low precision fixed-point representation while achieving similar accuracy as models trained with higher-precision floating-point numbers[25]. However, this is contingent on the rounding mode adopted for fixed-point operations.

2.5.4 Stochastic Rounding vs. Round-to-Nearest

The form of rounding that yields the best results when learning in fixed-point is stochastic rounding[25]. Generally, fixed-point numbers are of the form [QI.QF]. QI represents the integer part while QF represents the fractional part of the number. The number of bits used to represent QI is the integer length (IL). Similarly, the bit width of QF is the fractional length (FL). The word length (WL) is defined as IL + FL. For a signed fixed-point number, the possible range of values is, thus, given as $[-2^{IL-1}, 2^{IL-1} - 2^{-FL}]$. Additionally, the smallest positive number that can be represented is denoted by ε and is given by 2^{-FL}.

This form of data representation can store the model parameters for neural networks. Furthermore, the overall choice of IL and FL will determine the range and precision that can be represented. However, during fixed-point operations such as multiplication, intermediate results of higher precision than the operands are usually produced. These values must be rounded to lower precision values. In [25] two modes of rounding are considered for solving this problem: round-to-nearest and stochastic rounding.

Given a high precision value x, $\lfloor x \rfloor$ is defined as the largest integer multiple of ε such that $\lfloor x \rfloor \leq x$. For both modes of rounding, x is rounded to either $\lfloor x \rfloor$ or $\lfloor x \rfloor + \varepsilon$. Round-to-nearest deterministically rounds x to whichever of the target values is closest to x. Stochastic rounding, on the other hand, does a probabilistic rounding. It rounds x to $\lfloor x \rfloor$ with a probability, p, that is proportional to how close x is to $\lfloor x \rfloor$. If x is not rounded to $\lfloor x \rfloor$, it is rounded to $\lfloor x \rfloor + \varepsilon$. Formally, the two modes of rounding can be defined as follows:

- Round-to-nearest

$$Round(x, \varepsilon) = \begin{cases} \lfloor x \rfloor & if\ \lfloor x \rfloor \leq x \leq \lfloor x \rfloor + \frac{\varepsilon}{2} \\ \lfloor x \rfloor + \varepsilon & if\ \lfloor x \rfloor + \frac{\varepsilon}{2} < x \leq \lfloor x \rfloor + \varepsilon \end{cases}$$

- Stochastic rounding

$$Round(x, \varepsilon) = \begin{cases} \lfloor x \rfloor & with\ probability\ 1 - \frac{x - \lfloor x \rfloor}{\varepsilon} \\ \lfloor x \rfloor + \varepsilon & with\ probability\ \frac{x - \lfloor x \rfloor}{\varepsilon} \end{cases}$$

Stochastic rounding takes a non-deterministic approach to rounding that allows it to preserve information statistically. To illustrate this, consider this algorithm that starts with an initial value of 0 and repeatedly adds 0.3 for a hundred times, rounding the result at each step to the nearest integer (note that $\varepsilon = 1$ in this case):

```
// ALGORITHM example_1()
value = 0
for i → 1 to 100 do
      x = value + 0.3
      value = Round(x, 1)
return value
```

With the round-to-nearest approach, the value returned will be zero. However, for stochastic rounding, the expected value will be 30, which is equal to the result that would have been obtained without rounding. This property of stochastic rounding is desirable since it statistically preserves the effect of small parameter updates that would have otherwise been rounded to zero with the round-to-nearest approach.

A hardware architecture can be designed that trains neural networks using low precision fixed-point numerical representation. The results show that fixed-point with WL of 16 can achieve similar results as 32-bit floating-point. The effect of changing the precision by varying FL while maintaining WL of 16 is also investigated. And for configurations where $FL \geq 8$, stochastic rounding converges, whereas round-to-nearest fails to converge for some configurations. Furthermore, for situations where they both converge, stochastic rounding usually converges faster and with better accuracy than round-to-nearest[25].

2.6 Areas of Development and Relevant Ideas

Clearly, much work has been done in the design of FPGA accelerators. The focus of previous work seems to be centred around building user-friendly frameworks and exploring underlying structure of neural networks. Few of these concentrate on building an accelerator for microcontroller units. In [24] this idea is explored but the proposed architecture involves an FPGA and an ARM processor that are closely coupled. The aim of this paper, however, is to build a system that allows any FPGA board to be converted to an external accelerator chip for an arbitrary microcontroller.

Nevertheless, certain ideas in the related literature will be utilized, such as the idea of using fixed-point representation with stochastic rounding to achieve performance gains. Furthermore, the idea of designing a user-friendly framework for the final system will also be explored.

Chapter 3: System Requirements and Design

As evidenced in Chapter 2, there are several research papers that explore the utilization of FPGAs as accelerators for neural networks. This chapter will provide clarification on the specific needs this paper intends to address. This will be followed by an outline of the requirements that have to be met to satisfy these needs. Finally, the requirements will be translated into a system design.

3.1 User and System Requirements

The users of this system will be embedded system designers that wish to incorporate neural networks in their applications. For such users, the system provides a viable way to achieve desirable performance through an external FPGA accelerator. The scenario below illustrates how the proposed system addresses the needs of such a user.

3.1.1 Use Case

Scenario

Hannah wishes to design an embedded application that utilizes a neural network to control a robotic arm. Currently, Hannah is unable to achieve reasonable performance with her microcontroller unit due to high computational requirements of the neural network. Hannah decides to improve the performance of her system by leveraging a hardware accelerator. She purchases an FPGA and uses the proposed system to convert her FPGA into a custom external hardware accelerator for her microcontroller unit. She interfaces her accelerator with her microcontroller unit and achieves desired performance.

3.1.2 User Requirements

The user should be able to:

- Design and train a neural network targeted for an FPGA on a computer
- Compile the trained model to an FPGA design
- Upload the final design to an FPGA
- Interface their FPGA with a microcontroller unit
- Invoke routines on the microcontroller that will leverage the FPGA to perform computations

3.1.3 Non-functional Requirements

The system should:

- Be inexpensive
- Easily integrate into existing systems
- Have a gentle learning curve

3.2 System Design Overview

The requirements identified necessitate a system composed of both high-level software and low-level architecture. Consequently, the proposed system will consist of multiple elements that will have to work in tandem. The elements that make up the system can be grouped into three major modules:

- *Training Environment*: This module consists of all software that will run on a computer. These include libraries that will allow developers to train their neural networks on a computer. It will also include scripts that will help compile the final model for an FPGA platform.

- *Component Library*: This module will contain all configurable FPGA components that are necessary to implement a neural network on the FPGA. It will accept a design file produced from the *Training Environment*, which will describe how the components should be connected. And it will generate an FPGA design as its output.
- *Interface Module*: All libraries as well as FPGA components that coordinate the exchange of information between the FPGA and the microcontroller unit will belong to this module. This module will also include the design of hardware that facilitate data transfer. This will include the pin configurations for interfacing the FPGA and microcontroller unit.

3.3 Training Environment – Design

The first step in building the custom accelerator involves designing and training the neural network. Currently, neural networks are designed and trained in high-level languages such as Python, MATLAB or Lua. These languages are usually easy for researchers and application developers to learn and use in projects. For this reason, the *Training Environment* of this system will also be implemented in a high-level language. Of the languages employed, Python is the most widely used non-proprietary language for machine learning. Therefore, Python will be the language of choice.

3.3.1 Numerical Representation in Training Environment

Traditionally machine learning toolkits build models that are to be run on CPUs or Graphics Processing Units (GPUs). As such, model parameters are represented in floating-point format, which is supported by CPUs and GPUs. However, our final model must have its parameters in fixed-point format since that is the format that will be used on the FPGA. To solve this problem, the *Training Environment* will simulate fixed-point operations using integers, making it easier to cast the final design to true fixed-point.

3.3.2 Module Input and Output

The input to this module will be a description of the network. The number of neurons in each layer of the network will be passed as parameters to a python class to instantiate an object. The object will then be used to train the network. The output of the module will be the python object which will contain the trained parameters (weights & biases).

3.4 Component Library – Design

Several papers that seek to build neural network accelerators often create a library of configurable FPGA components. [23] and [24] are examples of this. The different components usually represent the digital building blocks of the neural network. For instance, in [23] a library of components is created. These components can be combined to build the common layers of neural networks such as the fully connected layer and the ReLU layer. The idea of having a component library is advantageous because it makes the design process modular. This makes it easy to extend the design of individual components without affecting the entire design. Also, adding support for more neural network layers is as simple as adding new components to the *Component Library*.

For this paper, the two ANN structures being considered are the fully connected layer and the ReLU activation. From Chapter 2, the fully connected layer can be described as a matrix-vector multiplication followed by a vector-vector addition. As such, this will be represented in the FPGA as a Matrix Multiplier and Summer Unit (MMSU). The output of this is usually fed into the activation unit, which in this case is the ReLU. This will be represented on the FPGA as RELU. Finally, the output of the ReLU will have to be rounded using stochastic rounding. This will be achieved by feeding the output of the RELU into the Stochastic Rounding Unit (SRU). Figure 3.4 demonstrates how these layers will work together.

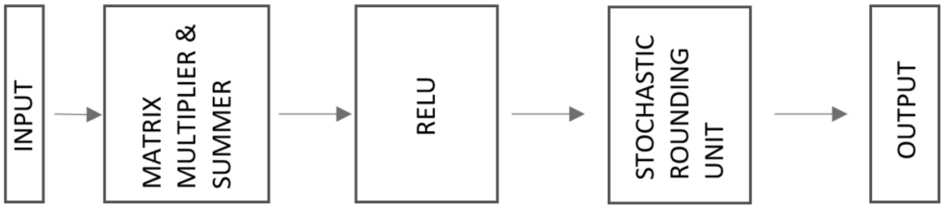

Figure 3.4: Layout of components will be cascaded

In Figure 3.4, the input refers to the output from a previous layer. And the output will feed its data into the next layer of the network.

3.4.1 Matrix Multiplier and Summer

This unit represents the fully connected layer and it will accept as its input, a vector. The vector will come from the output of a previous layer. The input vector will be used to multiply a weight matrix stored within the matrix multiplier. The result of the multiplication will produce an intermediate vector, which will then be added to the bias vector. In this process, the most compute-intensive task is the matrix-vector multiplication. The multiplication can, however, be broken down into multiple MAC operations that can be executed in parallel as shown Chapter 2. Conveniently, FPGAs have DSP units capable of performing MAC operations efficiently.

From equation (4), each element of the output vector can be represented as the result of the MAC operation performed on a row of the weight matrix and the input vector. As such, the MMSU will have Vector Multiplication Units (VMUs). Each VMU will take a row of the weight matrix and the input vector as its input. It will then compute the MAC operation on these two vectors and add the bias.

$$a_k^L = MAC(R_{\mathrm{k}}, A) + \ b_k^L \tag{6}$$

3.4.2 RELU

This unit does a simple operation. It takes as input, a value, and determines if the value is greater than 0. If it is greater than 0, it returns the value as its output. Otherwise, it returns 0. This can be described as a simple comparator in the FPGA design.

3.4.3 Stochastic Rounding Unit

The stochastic rounding unit will take as input, a fixed-point number of high precision and rounds it to a low precision value. In [25] this is achieved by generating a random number of the same bit-width as the Least Significant Bits (LSB) to be rounded off. The randomly generated number is added to the high-precision value and the LSB is dropped off. After this, if the final value is greater than the largest value that can be represented by the target WL, the output will be saturated to the maximum value, and the unnecessary Most Significant Bits (MSB) will be truncated. Similarly, if the value after rounding is less than the least value that can be represented by the target WL, the value will be saturated before truncating the MSB.

On the FPGA, the random number is generated using a pseudo-random number generator (PRNG). In [25] a linear feedback shift register (LFSR) is used as the PRNG. LFSR is easy to implement, requiring few logic resources. It also gives a good degree of randomness. The LFSR starts with a seed, selected by the designer. Each bit of the seed is stored in a different register as shown in Figure 3.4.3. The designer will then choose an arbitrary set of registers to be **ON** while the others remain **OFF**. On every clock cycle three things occur. First, the values in **ON** registers will be summed together and least significant bit will be taken. This value will be denoted as α. Secondly, every bit will be shifted to the register on its right. This will cause the rightmost bit to be dropped off. Lastly, α will be assigned to the leftmost register.

ON (Reg 1)	**OFF (Reg 2)**	**ON (Reg 3)**	**OFF (Reg 4)**	**ON (Reg 5)**
1	0	0	0	0

Figure 3.4.3: Shows how LFSR bit values are stored in a register

After the first few clock cycles, the values produced when the registers are read in parallel will be pseudo-random. This technique will be used for our Stochastic Rounding Unit.

3.4.4 Module Input and Output

The input to this module will be the python object produced as the output of the *Training Environment*. This object will be passed as a parameter to a python function. The python function will inspect the object and assemble a VHDL design that represents the final neural network. It will do this by fetching component templates and transforming them into actual VHDL components that suit the input neural network. The bus sizes as well as the specific internal design of components will be dependent on the final neural network passed to this module. The output of this module will be a set of VHDL files that can be compiled for a target platform using software such as Xilinx’s Vivado.

3.5 Interface Module – Design

The final module of the design focuses on the communication between the FPGA and the microcontroller unit. The aim is for the FPGA to act as a co-processor for the microcontroller unit, allowing each system to focus on the task it specializes in. The microcontroller unit will control the processing of information from sensors and will drive actuators. The running of the neural network will be left to the FPGA. This is illustrated in Figure 3.5.

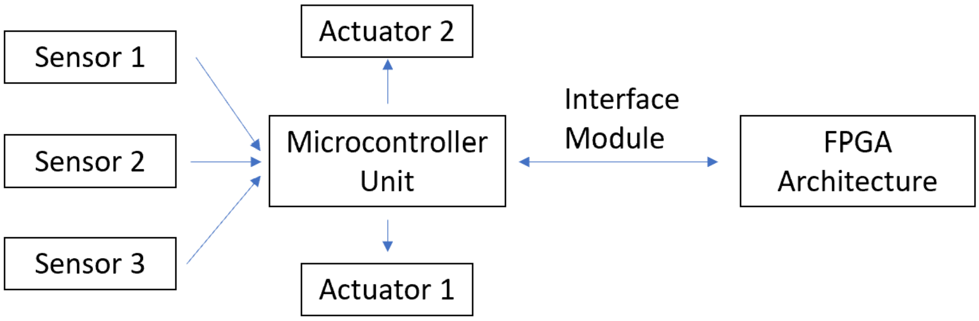

Figure 3.5: Shows how FPGA will be integrated into an embedded system

In practice, however, the input from the sensors will usually be fed to the input layer of the neural network. Therefore, a bridge will have to exist between the microcontroller unit and FPGA to exchange this information. To achieve this, a communication module will have to be designed on the FPGA. Subsequently, libraries will have to be designed for the microcontroller unit to allow easy information exchange. The protocol through which information will be transferred will also have to be considered.

3.5.1 Communication Protocol

A good number of microcontroller units and FPGAs support serial communication protocol. However, serial communication can usually be slow, requiring multiple clock cycles to transfer information. Instead, the General-Purpose Input/Output (GPIO) pins on the microcontroller unit will be utilized for communication. Data can be written and read from these pins in a few clock cycles. Furthermore, information can be written and read from multiple pins concurrently, allowing parallel data transfer.

3.5.2 Module Input and Output

This module will facilitate exchange of information between the two devices. As such it will accept input from the microcontroller unit and send it as output to the FPGA. It will also accept input from the FPGA and send it to the microcontroller unit as input.

3.6 Choice of FPGA

The final accelerator design of this system will be deployed on an FPGA. The primary goal of the design is compatibility; the final design should be compatible with almost any FPGA board. As such, the number of assumptions about the target FPGA board will be kept to a minimum. Below are the assumptions that will be made about the target FPGA board. The following elements should be available:

- Programmable Logic Cells
- Block RAM
- Digital Signal Processing Units

Consequently, any modern FPGA should suffice for testing purposes. However, since the use case is for embedded systems, it is only fitting that the design is tested on a low budget FPGA that will fall within the cost constraints of a small embedded project. For this reason, the Xilinx Basys3 Artix-7 board will be used. This board is described on the Xilinx webpage as an entry-level FPGA development board for student and beginners. This makes it suitable to be used as a baseline for testing. The specifications for the board are listed below.

The Xilinx Basys3 board:

- Has 33,280 logic cells in 5200 slices (each slice contains four 6-input Lookup Tables and 8 flip-flops

- Has 90 DSP slices
- Has internal clock speeds exceeding 450 MHz
- Has 1,800 Kbits of fast block RAM
- Features the Xilinx Artix-7 FPGA: XC7A35T-1CPG236C

3.7 Choice of Microcontroller Unit

Like the FPGA, the choice of a microcontroller unit is arbitrary since the focus is on compatibility. As such, a few assumptions are made about the target microcontroller unit. Below are the assumptions made. It should:

- Have a minimum of 10 GPIO pins
- Support external interrupts on, at least, one of these pins
- Operate between 3.3V and 5V

For this purpose, the Arduino Uno is used. The Arduino Uno is a well-tested and popular microcontroller unit within embedded system communities and academic spaces. The Arduino Uno operates at 16 MHz and supports external interrupt handling on two of its pins. It also operates at 5V. It has 6 analogue and 14 digital GPIO pins.

Chapter 4: Implementation

The previous chapter laid out the design overview of the various modules of the proposed system. The goal of this chapter is to describe and provide details pertaining to the implementation of the designs discussed in Chapter 3. This will include diagrams of constituent components and any necessary algorithms. The final implementation and tests can be found at: https://github.com/Oracking/FPGA_Accelerator.git

4.1 Training Environment – Implementation

The *Training Environment* will leverage the stochastic gradient descent (SGD) algorithm[6] to train the neural network. It will differ slightly from the standard SGD. Unlike the traditional SGD implemented in other toolkits, the *Training Environment* will store parameters – weights and biases – in fixed-point format and will use stochastic rounding to round results of fixed-point operations when necessary. The *Training Environment* will produce as output, the final parameters after training. It will then utilize the templates stored in the *Component Library* to generate a single file that describes top-level architecture of the accelerator. This architecture can then be compiled to a bitstream and uploaded onto an FPGA.

4.2 Component Library – Implementation

This module will be at the heart of the accelerator design process since it contains the digital components necessary to build the accelerator. In the Chapter 3, each layer in the neural network was described as an FPGA component. Components are usually strictly declared, with defined internal architecture and input/output data sizes. However, neural network layers are dynamic. A single neural network can have a fully connected layer with 20 neurons, followed by another fully connected layer with 10 neurons. In digital design, this implies there will be two instances of the MMSU component with different input and output sizes. Furthermore, the organization of logic blocks within each MMSU may also differ. This makes it impractical to have a single strictly defined architecture of each component.

As such, the library will not contain fully implemented components. Rather, it will contain templates of the components. These templates will have configurable parameters that will be supplied from the *Training Environment*. This implies that, in the scenario that was given, a template of an MMSU will be available. During the build process, two separate MMSU architectures will be described from the template, each with different input and output sizes, as well as internal structure. One instance of each architecture will then be created. These generated MMSU instances will then be connected to produce the final design of the accelerator.

In the subsections that follow, the template for each architecture will be described and illustrated. The design of components and templates will be implemented in VHDL.

4.2.1 Matrix Multiplier and Summer

The MMSU, as described in Chapter 3, implements the fully connected layer. It accepts the input from the previous layer, multiplies it with the weight vector and adds the bias vector. Figure 4.2.1 shows the pin configuration of the MMSU.

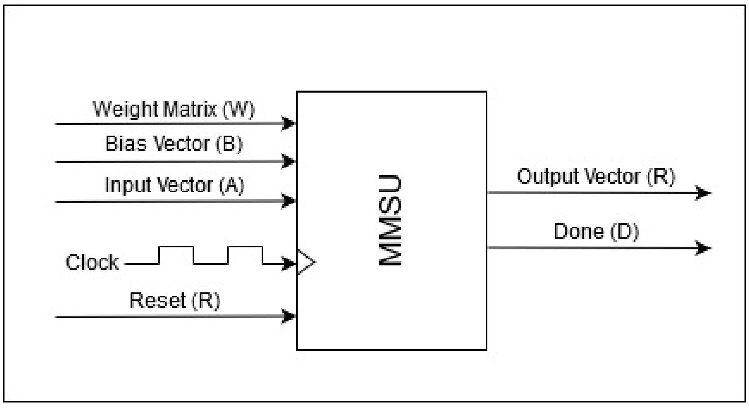

Figure 4.2.1: Pin configuration of MMSU

The functions of the input and output pins are as follows:

- **Weight Matrix**: This holds the weight of all connections from previous layer to current layer.
- **Bias Vector**: This holds the biases associated with each neuron in the current layer
- **Input Vector**: This holds the activations of the neurons from the previous layer
- **Clock**: The input clock signal is generated by the FPGA and is used to drive the operations of the MMSU.
- **Reset**: The reset pin is used to clear the MMSU and prepare it for a new set of inputs. When the reset pin is set to high, it clears any pre-computed values on the **Output Vector** line. It also sets **Done** to low. When it is set to low, the MMSU begins computation.
- **Output Vector**: This gives the output of the layer. However, it makes the output available as it is being computed. This implies that intermediate computational values are made available on this line.
- **Done**: When the MMSU gets new input, it sets this pin to low and begins computation. When it is done computing and the value on the **Output Vector** is final, this pin is set to high.

As described in Chapter 3, the multiplication of the weight by input vector is achieved through parallel VMU computations within the MMSU. Each row of the weight matrix is passed to a VMU, alongside the input vector. Each VMU performs a MAC operation on the two inputs it is given. This is achieved by leveraging the onboard DSP units, which perform fast MAC operations.

The DSP starts by initializing its output to zero. On each clock cycle, the VMU feeds the DSP unit with two values: element k, from the row of the weight matrix it has been passed, and element k, from the input vector. The DSP multiplies these two values and adds result to its current output. The VMU then increments k, to feed the DSP with the next set of values. After a few clock cycles, the VMU would have successfully performed the MAC operation on the row of weight matrix and input vector it was passed.

Extra attention must, however, be paid to the output of the DSP. Since the fixed-point representation is being used, the result of the MAC operation will contain more bits than either operand. The DSP, however, makes it possible to initialize the output to be 48-bit wide, which is much larger than the typical WL to be used for the model. Yet still, this can overflow with enough accumulation.

4.2.2 RELU

RELU represents the ReLU activation in the neural network. It takes a value as input and produces zero as its output if the input value is less than zero. Otherwise, the output value is assigned the input value. Figure 4.2.2 shows the pin configuration of the RELU block.

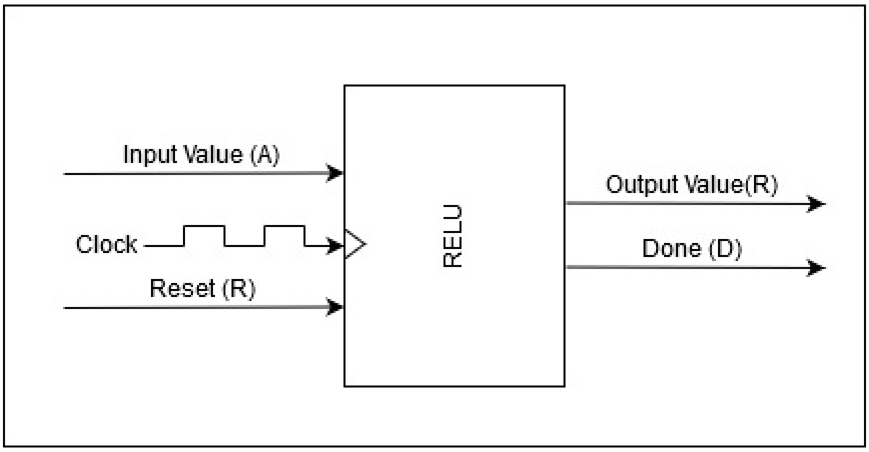

Figure 4.2.2: RELU pin configuration

When the *Reset* pin is set to high, it clears the value on the *Output Value* line, and sets the value of *Done* to low. When the *Reset* pin is set to low, it takes the input value on the *Input Value* line and, after a single clock cycle, returns its output on the *Output Value* line. The value of *Done* is then set to high.

4.2.3 Stochastic Rounding Unit

The stochastic rounding unit ensures that results of computations from any layer is finally rounded to the set WL of the system. As described in Chapter 3, it achieves this by using an LFSR to generate a random number which is added to the LSB of its input value. The pin configuration of the SRU is shown in Figure 4.2.3.

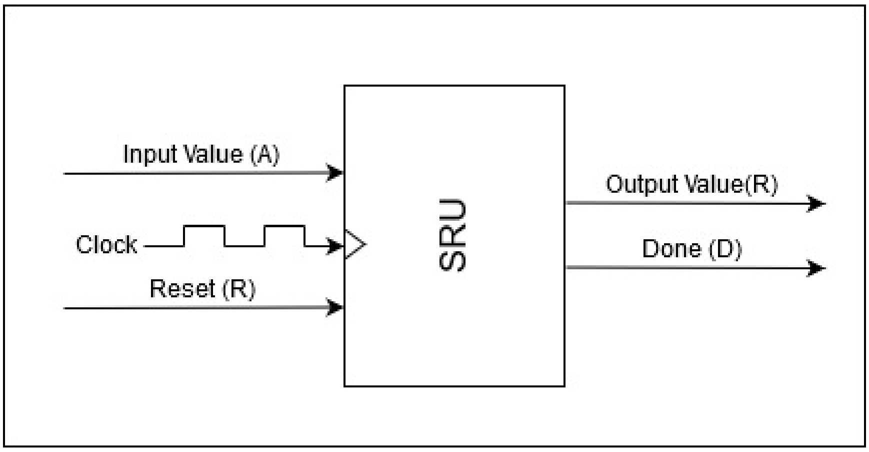

Figure 4.2.3: Pin configuration of stochastic rounding unit

The pins of the SRU perform the same function as the pins on the RELU. The main difference is with the internal logic of the block. The SRU takes a random value generated by the LFSR and uses it to stochastically round the input value. Its output value will have the WL of the system.

4.3 Interface Module

This module is an integral part of the system as it facilitates the exchange of data between the FPGA and the microcontroller unit, allowing the FPGA to function as a coprocessor. As shown in Figure 3.5 the microcontroller unit will focus on collecting and processing data from the sensors. The data will then be sent out to the FPGA as input to the neural network. The FPGA will, subsequently, send the result of the model to the microcontroller. The steps involved are laid out in the algorithm below. Note that the steps in the algorithm are enumerated as they will be referenced in later sections that shed light on the implementation.

```
// ALGORITHM exchange_data()
while true
    1. Microcontroller reads data from sensors
    2. Microcontroller sends data to FPGA as input to model
    3. FPGA performs necessary model computations
    4. FPGA sends results back to microcontroller
```

4.3.1 Pin Configuration and Utilization

To allow for quick exchange of information between the microcontroller unit and the FPGA, the GPIO pins on the microcontroller unit are utilized. Most microcontroller units can set and read the value of multiple GPIO pins in a few clock cycles. The GPIO pins from the microcontroller unit are connected to the FPGA and are used to transfer data bits between the two devices.

The implementation utilizes ten digital pins for information exchange between the two devices. The pin configuration is shown in Figure 4.3.1. Four of these pins hold data to be sent from the microcontroller unit to the FPGA. Two of the pins hold data to be transferred from the FPGA to the microcontroller. The four remaining pins are used to coordinate information transfer.

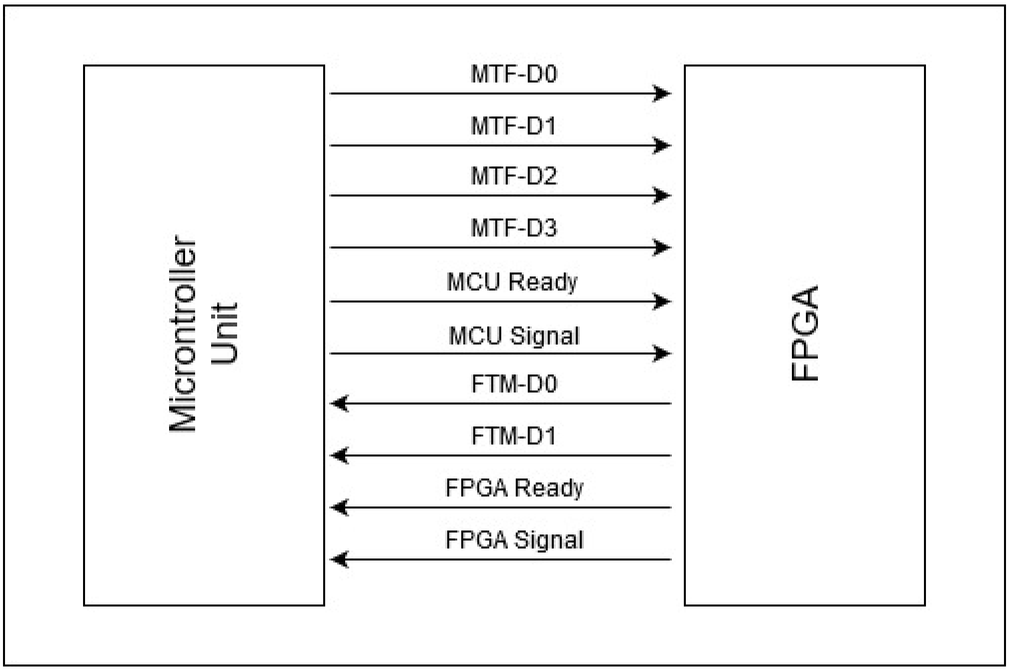

Figure 4.3.1: Shows pin configuration for communication between FPGA and microcontroller

When both devices are connected to each other, the *FPGA Ready* pin is set to be high by the FPGA. Similarly, the *MCU Ready* pin is set to high by the microcontroller unit. This is a preliminary step that indicates to either device that information transfer is ready to occur. The behaviour of the other pins will depend on which step in the algorithm **exchange_data** that we are considering. In each step, the pins are utilized as follows:

Step 1: This step does not require the pins to be set to any specific value. It will be implemented by the libraries associated with the sensors being used.

Step 2: Any information to be sent to the FPGA will be put on the pins *MTF-D0*, *MTF-D1*, *MTF-D2* and *MTF-D3*. These pins are collectively referred to as the *Microcontroller to FPGA Data (MTF-D)* pins. Each of the *MTF-D* pins represents a single bit of information. As such, the *MTF-D* pins, together represent 4 bits of information. Consequently, 4 bits of information can be sent to the FPGA at a go. Hence, all information to be sent will be sent 4 bits at a time.

To transfer data, the *MCU Signal* pin is set to low and the data to be sent is put on the *MTF-D* pins. After, the *MCU Signal* is set to high. The FPGA detects the rising edge on the *MCU Signal* pin and sets *FPGA Signal* pin to low. It then proceeds to read data from the *MTF-D* pins. When it's done, it sets *FPGA Signal* to high. This rising edge on the *FPGA Signal* pin is detected by the microcontroller unit, indicating a successful transfer of information to the FPGA.

Step 3: The data collected will be fed into the first fully connected layer of the network which will proceed to compute and propagate its result through the rest of the network. In this step, the pins will not require any special configuration.

Step 4: When the result of the model is ready, the *FPGA Signal* pin is set to low, and the data is put on the pins: *FTM-D0* and *FTM-D1*. These two pins are responsible for carrying data from the FPGA to the microcontroller. They are referred to, collectively, as *FPGA to Microcontroller Data* (*FTM-D*) pins. After data is put on the *FTM-D* pins, the *FPGA Signal* pin is set to high. The microcontroller detects the rising edge on this pin and sets *MCU Signal* to low. It then proceeds to read the data on the *FTM-D* pins. After reading, the *MCU Signal* pin is set to high. The rising edge on this pin is detected by the FPGA, indicating a successful transfer.

4.3.2 FPGA Component

On the FPGA, an interface module is implemented that allows the pin configuration described to be utilized. Most of the behaviour can translate directly to VHDL code. However, on the Basys3, it is not possible to detect a rising edge on an input pin. As such, determining the rising edge on the *Arduino Signal* pin cannot be achieved directly. Consequently, this behaviour must be simulated. This is done by buffering the values of the *Arduino Signal* pin. On every clock cycle, it analyses the current value of *Arduino Signal* pin and the value stored in the buffer. If the current value of *Arduino Signal* pin is high and the previous value, stored in the buffer, is low, then it identifies it as a rising edge.

4.3.3 Microcontroller Unit

Implementing the microcontroller component of this module involves writing procedures to read and write data bits from the digital pins. It also requires the efficient handling of interrupts. Since our target users are embedded system developers, they are likely to possess the skills required to achieve this on their microcontroller unit. Nevertheless, for this paper, the necessary procedures will be implemented for the Arduino Uno.

Chapter 5: Testing and Results

This chapter focuses on testing the proposed system by comparing a system that utilizes the FPGA accelerator against one that doesn’t. To test the efficiency of our accelerator, a neural network will be designed and deployed on two embedded systems. One embedded system will run the neural network on a microcontroller unit without support from an accelerator. The second system will run the same neural network but will use the FPGA as an accelerator for the microcontroller unit. As already shown in [25], networks trained with fixed-point and stochastic rounding achieve similar accuracy as those trained on floating-point. As such, the focus of this test will be on the speedup provided by the accelerator.

5.1 Neural Network Design

In the test scenarios, the neural network designed will be used to help a rover navigate an environment. The rover has 3 ultrasonic sensors pointing outwards. The values from the ultrasonic sensors are read and passed to the network. The network then decides whether the rover should turn right, turn left or continue moving forward in order to avoid obstacles. The test will focus on how fast the network makes a prediction.

For this problem, the neural network to be deployed will have an input layer with 3 neurons. This layer will be followed by a fully connected layer with 16 neurons. The neurons in the fully connected layer will use the ReLU activation function. The fully connected layer will then be followed by an output layer with three neurons.

5.2 System I – System without Accelerator

The final neural network design will be deployed on an Arduino Uno without an accelerator. This system will be the baseline for testing. The parameters of the neural network will be stored in floating-point format for this system. To compute the output of the fully connected layer, equation (5) will be utilized. The weight matrix, input vector and bias vectors in the equation will be represented as arrays. The weight matrix will be represented as a 2-dimensional array whereas the input and bias vectors will be represented as 1-dimensional arrays. Nested for loops will be used to compute the product of the weight matrix and input vector.

The time taken for the neural network to evaluate an output once it has the input to the network will be recorded over multiple tests and averaged. Specifically, the program will start timing when data is ready to be fed into the network and stop when the output of the network is produced. This will then be compared to the system that utilizes the accelerator.

5.3 System II – System with Accelerator

The neural network designed will also be deployed on another system. This system will utilize both a microcontroller unit and an FPGA accelerator. The neural network will be described in the *Training Environment*, which will train the network on some data and produce the final parameters of the network. The training will be done in 16-bit fixed-point format. 8 bits will be used for the IL and 8 bits will be used for the FL. The *Training Environment* will generate the VHDL design files that describe the architecture of the accelerator. This architecture description will be compiled and downloaded onto the Xilinx Basys3 board via Vivado. The Basys3 will run the neural network and interface with the Arduino Uno to read information from the sensors. The Basys3 will return to the Arduino, the index of the whichever neuron in the output layer has the greatest activation.

The time taken for the neural network to evaluate its output once it has the sensor readings will be recorded. More specifically, the program will start timing when the microcontroller is about to send the input data to the FPGA. It will finish timing when the results from the FPGA are received by the Arduino Uno.

5.3.1 Generated Architecture

Figure 5.3.1 demonstrates the final FPGA architecture generated by the *Training Environment*. The architecture consists of two MMSUs, one RELU and two SRUs. The first MMSU computes the result of the fully connected layer with 16 neurons. It then feeds its output to the RELU, which then passes the activations through the ReLU function. The output of the RELU is forwarded to the SRU which rounds the results to 16-bit fixed-point.

The output of the first SRU then goes into the second MMSU, which computes the value of the output layer. Naturally, the output of this MMSU will be fed into an SRU to round the results to a 16-bit fixed-point format.

Finally, the communication layer will examine all neurons in the output layer and send the index of the neuron with the highest activation to the Arduino Uno

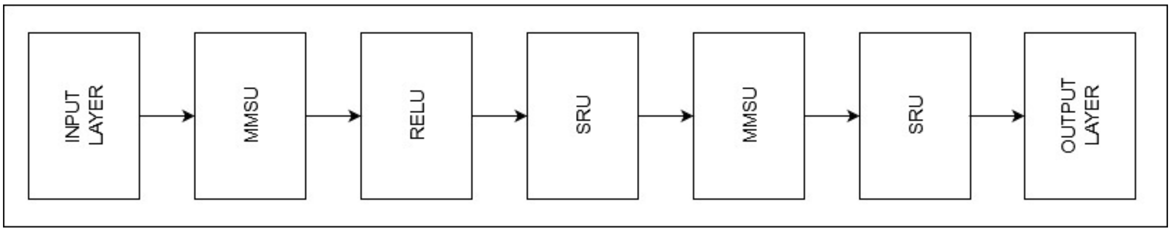

Figure 5.3.1: Generated Architecture of neural network

5.4 Results & Discussion

The performance of System I and System II are both evaluated. On average, System I uses 586 microseconds to evaluate the output of the network. System II uses 250 microseconds to compute the results of the network. In other words, System II runs at more than twice the speed of System I. As such, the FPGA accelerator does provide reasonable performance gains.

Interestingly, in System II, most of the time is spent on exchanging information between the FPGA and microcontroller unit. The computation by the FPGA is relatively fast. Therefore, for applications that have many inputs, this can prove to be a bottleneck. The full test can be found by navigating to the test folder of the following repository: https://github.com/Oracking/FPGA_Accelerator.git

Chapter 6: Conclusion

The previous chapters laid out the problem and developed a solution that addresses it. For the neural network tested, the system that used the FPGA as a coprocessor achieved more than twice the speed of the system that did not use the FPGA. The focus of this chapter will be to shed light on the some of the limitations that were encountered in developing and testing the solution. It will also highlight some prospective areas of improvement that can be explored by future researchers.

6.1 Limitations

In the process of developing and testing the proposed system, some limitations were encountered that influenced the choices made. Listed below are some of the limitations faced:

- One of the limitations was the size of the neural network that could be tested. With the proposed solution, the size of the network that could be deployed on the FPGA was limited by the number of DSP tiles available on the board. In the case of the Basys3 this was 90. As such, a neural network that has many neurons in one or more of its layers will be unable to fit on the Basys3.
- Another limitation was communication bottleneck which limited the speeds that could be attained with the FPGA and microcontroller.

6.2 Future Works

This paper sets out to address a problem. And through the chapters, a feasible solution is developed that addresses the requirements of the problem. However, as is usually the case with scientific development, the initial solution is rarely the final one. Therefore, even though the solution presented is feasible, there are still ways through which it can be improved. Below are suggested improvements that can be further explored:

- Currently, the accelerator only supports networks that utilize fully connected layers with ReLU activation functions. In the future, the development framework can be extended to include support for other types of layers and activation functions.
- The utilization of DSP tiles on the FPGA is not the most efficient since each Vector Multiplier unit requires its own DSP. This can be improved by allowing different layers to share DSP tiles via time-multiplexing.
- The FPGA accelerator can only be used to make predictions. It cannot be used to train the network. As such, the accelerator design can be further developed to support learning. This will allow learning to occur in a real-time environment.

References

[1] M. I. Jordan and T. M. Mitchell, “Machine learning: Trends, perspectives, and prospects,” *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[2] P. Domingos, “A few useful things to know about machine learning,” *Commun. ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[3] T. O. Ayodele, “Types of machine learning algorithms,” *New Adv. Mach. Learn.*, pp. 19–48, 2010.

[4] I. G. Maglogiannis, *Emerging artificial intelligence applications in computer engineering: real word ai systems with applications in ehealth, hci, information retrieval and pervasive technologies*, vol. 160. Ios Press, 2007.

[5] J. Zou, Y. Han, and S.-S. So, “Overview of artificial neural networks,” in *Artificial Neural Networks*, Springer, 2008, pp. 14–22.

[6] M. A. Nielsen, *Neural networks and deep learning*, vol. 2018. Determination press San Francisco, CA, USA:, 2015.

[7] L. Zhang, S. Wang, and B. Liu, “Deep learning for sentiment analysis: A survey,” *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, vol. 8, no. 4, p. e1253, 2018.

[8] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural Netw.*, vol. 61, pp. 85–117, 2015.

[9] P. D. McNelis, *Neural networks in finance: gaining predictive edge in the market*. Academic Press, 2005.

[10] P. J. Lisboa, E. C. Ifeachor, and P. S. Szczepaniak, *Artificial neural networks in biomedicine*. Springer Science & Business Media, 2000.

[11] P. W. Mirowski, Y. LeCun, D. Madhavan, and R. Kuzniecky, “Comparing SVM and convolutional networks for epileptic seizure prediction from intracranial EEG,” in *2008 IEEE workshop on machine learning for signal processing*, 2008, pp. 244–249.

[12] M. Barr, *Programming embedded systems in C and C++*. O’Reilly Media, Inc., 1999.

[13] D. A. Pomerleau, “Knowledge-based training of artificial neural networks for autonomous robot driving,” in *Robot learning*, Springer, 1993, pp. 19–43.

[14] A. Badlani and S. Bhanot, “Smart home system design based on artificial neural networks,” in *Proceedings of the World Congress on Engineering and Computer Science*, 2011, vol. 1, pp. 19–21.

[15] F. Yang and M. Paindavoine, “Implementation of an RBF neural network on embedded systems: real-time face tracking and identity verification,” *IEEE Trans. Neural Netw.*, vol. 14, no. 5, pp. 1162–1175, 2003.

[16] P. R. Gankidi, “FPGA accelerator architecture for Q-learning and its applications in space exploration rovers,” PhD Thesis, Arizona State University, 2016.

[17] J. Zhu and P. Sutton, “FPGA implementations of neural networks–a survey of a decade of progress,” in *International Conference on Field Programmable Logic and Applications*, 2003, pp. 1062–1066.

[18] G. Lacey, G. W. Taylor, and S. Areibi, “Deep learning on fpgas: Past, present, and future,” *ArXiv Prepr. ArXiv160204283*, 2016.

[19] A. R. Omondi and J. C. Rajapakse, *FPGA implementations of neural networks*, vol. 365. Springer, 2006.

[20] A. Dertat, “Applied Deep Learning - Part 1: Artificial Neural Networks.” https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6.

[21] G. Sanderson, “But what is a Neural Network? | Deep learning, chapter 1.” https://www.youtube.com/watch?v=aircAruvnKk.

[22] U. Farooq, Z. Marrakchi, and H. Mehrez, “FPGA architectures: An overview,” in *Tree-based heterogeneous FPGA architectures*, Springer, 2012, pp. 7–48.

[23] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, “DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family,” in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[24] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, “A 240 g-ops/s mobile coprocessor for deep neural networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2014, pp. 682–687.

[25] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *International Conference on Machine Learning*, 2015, pp. 1737–1746.