PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER ENGINEERING

AN FPGA IMPLEMENTATION FOR CONVOLUTIONAL NEURAL NETWORK

GUILHERME DOS SANTOS KOROL

End-of-term work submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fullfillment of the requirements for the degree of Bachelor in Computer Engineering.

Advisor: Prof. Dr. Fernando Gehm Moraes

Porto Alegre
2019

ACKNOWLEDGMENTS

I want to express my gratitude to all those that made this work possible.

First, to my family that has always supported me. This was only possible because you were there. To Iris, Ricardo, and Leo my most sincere thank you. To Joice, who read this document with me so many times, thank you for your love and support.

Second, to all my colleagues from GAPH (*Grupo de Apoio ao Projeto de Hardware*) that offered suggestions and expertise. There were times that your help was crucial to the progress of the project. To professor Marcon, Cataldo, and Ramon to whom with I had my first research experience, thank you. To the two that were by my side (literally, as their desks are next to mine), Leonardo and Tanauan, our technical discussions deserve my greatest appreciation.

Last, to my advisor that, untiringly, engaged in endless meetings. Your bits of advice, I will take with me. Thank you.

UMA IMPLEMENTAÇÃO EM FPGA PARA REDES NEURAIS CONVOLUCIONAIS

RESUMO

Avanços recentes em plataformas de *hardware* impulsionam o uso de Redes Neurais Convolucionais na resolução de problemas nas mais diversas áreas, tais como Processamento de Linguagem Natural e Visão Computacional. Com os melhoramentos nos algoritmos envolvidos na aprendizagem e inferência de Redes Neurais Convolucionais, uma grande quantidade de arquiteturas dedicadas em *hardware* foram propostas para prover maior desempenho com custos reduzidos em área e consumo energético. Em especial, os altos níveis de exigência em termos de largura de banda e poder de processamento desafiam os projetistas a criarem estruturas de Redes Neurais Convolucionais eficientes, e passíveis de serem implementadas em ASICs e FPGAs. Esse trabalho tem por objetivo implementar, especificamente em plataformas reconfiguráveis (dispositivos FPGA), um estudo de caso de Rede Neural Convolucional, a Alexnet. Adicionalmente, o trabalho propõe a avaliação frente a uma versão em *software* simulada para um ambiente baseado em processador ARM.

Palavras-Chave: FPGA, Redes Neurais Convolucionais, Alexnet.

AN FPGA IMPLEMENTATION FOR CONVOLUTIONAL NEURAL NETWORK

ABSTRACT

Recent advances in hardware platforms boosted the use of Convolutional Neural Networks to solve problems in several fields such as Computer Vision and Natural Language Processing. With the improvements of algorithms involved in learning and inferencing for Convolutional Neural Networks, a huge amount of dedicated hardware architectures have been proposed to provide high performance at low energy and area costs. Notably, requirements in bandwidth and processing power have challenged architects to find structures that allow modern Convolutional Neural Networks to be embedded into ASICs and FPGAs. This work aims to implement a Convolutional Neural Network case study, the Alexnet, targeting configurable (FPGA devices). Additionally, it proposes an evaluation against an ARM-simulated software version.

Keywords: FPGA, Convolutional Neural Networks, Alexnet.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ANN – *Artificial Neural Network*

AI – *Artificial Intelligence*

ASIC – *Application-Specific Integrated Circuit*

CLB – *Configurable Logic Blocks*

CNN – *Convolutional Neural Network*

DNN – *Deep Neural Network*

DSP – *Digital Signal Processing*

FC – *Fully Connected*

FPGA – *Field Programmable Gate Array*

GPU – *Graphics Processing Unit*

LRN – *Local Response Normalization*

LUT – *Look-up Table*

MAC – *Multiply and Accumulate*

MLP – *Multilayer Perceptron Network*

OVP – *Open Virtual Platform*

RELU – *Rectified Linear Unit*

RTL – *Register-Transfer Level*

USB – *Universal Serial Bus*

VHDL – *VHSIC Hardware Description Language*

VLSI – *Very Large Scale Integration*

CONTENTS

1. INTRODUCTION

In artificial intelligence, deep learning is getting much attention from industry and academia. The attention comes mainly from recent advances in hardware that provided room for complex algorithms that run in acceptable time, making it possible to recover meaning from the enormous quantity of data made available every day. Many problems regarding the hardware architectures that are being developed for this kind of use remain open.

Nowadays, the number of applications using CNNs is countless. The two most common fields of application are Natural Language Processing and Computer Vision. Convolutional Neural Networks have shown a great deal of improvement on audio [Hershey et al., 2017], video [Karpathy et al., 2014], and image classification [Russakovsky et al., 2015], as well as segmentation [Girshick et al., 2014], scene labeling [Pinheiro and Collobert, 2014], and face recognition [Lawrence et al., 1997]. There are even networks that achieve better than human-level performance in some tasks such as the ResNet that in 2015 achieved error rates lower than human experts in image classification [He et al., 2016]. Unfortunately, this performance comes at a cost. The algorithms employed by the most modern neural networks require high memory bandwidth and extensive use of computational resources [Sze et al., 2017].

To mitigate the memory and performance bottlenecks created by recent CNNs, a tremendous quantity of hardware architectures have been proposed by industry and research communities. For example, Microsoft created the Project Catapult, developing FPGAs accelerators for its cloud servers in [Ovtcharov et al., 2015, Putnam et al., 2016, Caulfield et al., 2016]. Google invested in the development of a series of ASICs for deep learning processing known as the Tensor Processing Unit [Jouppi et al., 2017]. ARM presented the ARM Machine Learning Processor and the Object Detection Processor [Zhu et al., 2018]. Interestingly, Movidius, an Intel company, has embedded a Deep Neural Network (DNN) prototyping tool into a USB stick for rapid development [Intel Corp., 2015]. Moreover, many startups appeared. To mention a few, Graphcore developed the Intelligence Processing Unit [Stokel-Walker, 2018], NovuMind has shown NovuTensor and NovuBrain chips for deep learning [NovuMind, 2018], Ceva introduced the Intelligent Vision Processor [Ceva, 2015].

Memory is the first challenge to address when developing hardware for deep learning [Sze et al., 2017, Wang et al., 2018]. Fundamentally, a neural network consists of a set of neurons organized in layers. Each layer can be viewed as a series of multiply and accumulate operations between the input data and some parameters. It turns out that the number of layers is getting larger with each new network is made public. From a hardware perspective, the growth in the number of layers means that the amount of data to be transferred from memory to the processing unit and then saved back to memory is also increasing. For

instance, in 2012 Alexnet won the Imagenet competition with a network of 60 million parameters to process a single image [Krizhevsky et al., 2012]. Two years later in 2014, VGGNet won the same contest, but instead, this network has seven times more parameters to load [Simonyan and Zisserman, 2014a]. Hence, it is expected that the architectures make good use (and reuse) of data to minimize the time consumed by memory transfers.

On the other hand, neural networks require a considerable amount of processing as well. For instance, a convolution slides a window over the inputs, computing a sum of products several times. Furthermore, the second most common kind of layer in CNNs, the fully connected, can be viewed as a sequence of matrix multiplications, which can also be computationally challenging for any physical implementation. Taking the networks previously cited, Alexnet has five convolutional layers and three fully connected layers while VGGNet consists of 16 convolutional layers and three fully connected layers. This amount of operations poses a problem even for architectures that are composed of many Graphics Processing Units (GPUs) arranged on a computer. Specifically, when it is proposed to embed these networks on FPGAs or dedicated ASICs, this volume of operations requires well-thought schemes for sharing arithmetic resources and, consequently, saving area.

There are three primary design paths that one can follow when developing hardware architectures for deep learning algorithms. The first one is adopting the use of GPUs, where developers implement a network by programming these and taking advantage of the high level of parallelism provided. As for the training phase, GPUs are the number one choice for machine learning scientists, as they are mainly focused on achieving peak performance once training will not be executed regularly. Besides, during this phase, high precision floating-point operations (also provided by GPUs) are required by algorithms like backpropagation. The other two common paths include FPGAs or ASICs. The former is one of the most flexible choices, and the latter achieves the highest performance [Kuon and Rose, 2007]. The design flow for an ASIC is expensive and involves many steps from initial specification to final tape-out and fabrication, unlike FPGAs that can deliver fast time-to-market and possible upgrades to the architecture after delivery. Fundamentally, FPGA and ASIC have attracted numerous designers to implement inference algorithms, due to: (*i*) the possibility of using numeric formats of any precision [Jiao et al., 2017, Courbariaux et al., 2015]; (*ii*) the possibility of energy saving compared to GPUs [Nurvitadhi et al., 2017, Gupta et al., 2015]; (*iii*) and the overall performance and potential for fine-tuning of the architecture [Hailesellasie et al., 2018].

1.1 Objectives

The goal of this end-of-term work is to implement a hardware architecture for Convolutional Neural Networks, specifically the Alexnet Convolutional Neural Network. The hardware implementation is compared to an Alexnet software version.

In a broader context, the goal is the understanding of the issues concerning the hardware implementation of a deep artificial neural network in general. As demonstrated in Section 2.2.4, despite the specific choice of Alexnet network as the case study, its architecture modeling includes mechanisms like convolution and fully connected layers, repeatedly used in other DNNs.

1.2 Document Structure

This document is structured in seven chapters. Chapter 1 contextualizes this work, presenting the central issues involving the hardware development for deep learning networks and some examples from industry. In Chapter 2, Field Programmable Gate Arrays and Machine Learning concepts relevant to this work are given. Next, the reference CNN is detailed in Chapter 3. In addition, a evaluation of two software implementations is presented. Then, Chapter 4 presents the hardware modules that supported this work; and Chapter 5 details the implemented architecture. Concluding the text, Chapter 6 traces an evaluation of the architecture, and Chapter 7 concludes the document.

2. THEORETICAL REFERENCE

This work embraces two distinct fields: hardware design and machine learning (convolutional neural networks). Thus, this Chapter introduces basic concepts for both fields, making the text self-contained, easing its reading for the general public. Section 2.1 introduces concepts related to FPGAs and Section 2.2 principles of machine learning.

2.1 Field Programmable Gate Arrays

Very Large Scale Integration (VLSI) systems enabled designers to develop complex digital circuits. Those circuits, or chips, designed using full-custom or standard cell methods, can integrate more than a billion of transistors nowadays. However, the cost and the design time of such chips requires a high volume of units to be sold. Field Programmable Gate Arrays (FPGAs) have emerged providing a faster time-to-market development and rapid prototyping [Stephen D. Brown et al., 1992].

The principle behind FPGAs is programmability. Its generic circuitry can be configured to execute any function for any specific application (digital functions). To achieve its programmability, the FPGA distributes several logic blocks across a programmable interconnection fabric surrounded by input and output blocks for interfacing the FPGA core with outside devices. Also, FPGA architectures offer dedicated clock distribution and control to synchronize all the blocks inside it, as well as dedicated blocks for memory and arithmetic functions, for example. Figure 2.1 shows the arrangement of the basic blocks inside an FPGA.

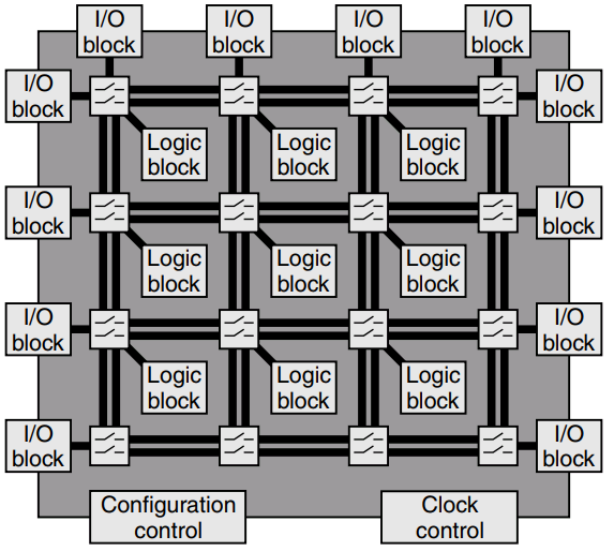

Figure 2.1 – The basic architecture of an FPGA. Adapted from [Bailey, 2011].

FPGAs may differ in the technology they are built. There are two main configuration technologies for FPGAs: antifuse and RAM-based [Bailey, 2007]. Devices using antifuse technology are configured once, but the performance is higher than memory-based FPGAs. On the other side, external devices (SRAM, EEPROM, Flash memories) stores the configuration of memory-based FPGAs. The advantage of memory-based FPGAs is clear: the easiness to modify and update the hardware.

The most frequent technology employed by FPGAs is the SRAM memory-based. In SRAM-based FPGAs, the logic function of a block or the state of an interconnection is controlled by programmed SRAM cells [Bailey, 2007]. The next subsections detail the logic, memory, and arithmetic blocks in an FPGA.

2.1.1 Function Generators

The function generators are the core of any FPGA. They are responsible for performing the logic functions programmed by the user. In modern FPGAs, functions are implemented using look-up tables (LUT).

Figure 2.2 presents a 3-input LUT. The LUT function is stored in a register during the device configuration. The register is a collection of memory cells containing all possible values (truth table) for a given function. Hence, making the LUT able to implement any function that fits its number of inputs. Once the device is configured, the LUT output is selected according to the input values (the inputs are the multiplexer selector). Modern FPGAs contain 6-input LUT, using 64-bit registers [Xilinx Inc, 2016a].

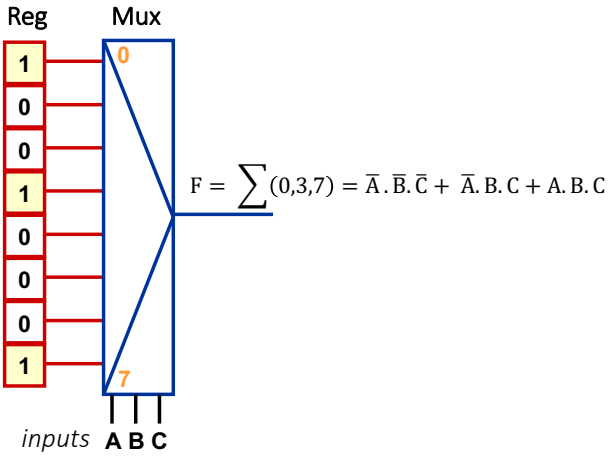

Figure 2.2 – Example of a 3-input Look-Up-Table (LUT3).

In current FPGAs, several LUTs are grouped into larger modules called Configurable Logic Blocks (CLBs). These modules provide faster internal connections between

LUTs than those provided by connections across the FPGA network. Also, they may house flip-flops, shift registers, distributed RAMs, multiplexers, and arithmetic operators (Figure 2.3).

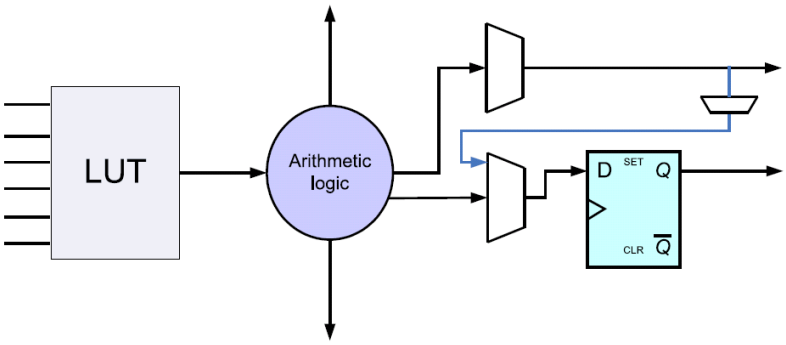

Figure 2.3 – A basic CLB from Xilinx FPGAs [Bailey, 2007].

2.1.2 Block RAM Memories

Block Ram memories (BRAMs) are FPGA internal modules, that provide fast data access. In modern architectures they offer flexibility regarding word width, reading/writing modes, and multi-clock capabilities. BRAMs can be inferred by synthesis tools or be instantiated by the user in the design capture phase. Additionally, most FPGAs provide *First In, First Out* (FIFO) modules that can be instantiate as macros by users. A schematic diagram of a FIFO module is shown in Figure 2.4.

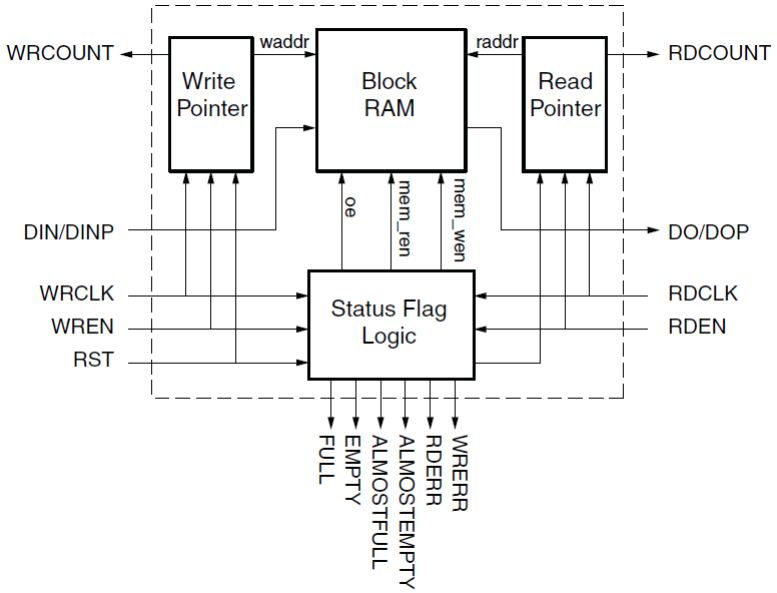

Figure 2.4 – A Xilinx FIFO implementation [Xilinx Inc, 2016b].

An important BRAM feature explored in this work is the double port access, i.e., it is possible to have two memory accesses simultaneously. This feature enables data prefetching or simultaneous read and write accesses.

2.1.3 DSP Blocks

FPGA programmability makes them suitable for applications requiring parallel processing. Examples of applications include Finite Impulse Response filters, video and voice processing, Fast Fourier Transforms, and remote sensing. These applications induced the introduction of Digital Signal Processing (DSP) blocks in modern FPGAs, enabling binary multiplication and accumulation, without using the basic blocks (LUTs).

In a DSP block, there may be several configurable functional blocks. For instance, the example in Figure 2.5, there is a 25 x 18 two's-complement multiplier, a 48-bit accumulator (that can also be used as up and down counter), a pre-adder, single-instruction multiple-data arithmetic unit (dual 24-bit or quad 12-bit add, subtract, or accumulate), optional logic unit (bitwise operations like AND, OR, NOT, AND, NAND, NOR, XOR, and XNOR), pattern detector (overflow and underflow), and a configurable pipeline [Xilinx Inc, 2018].

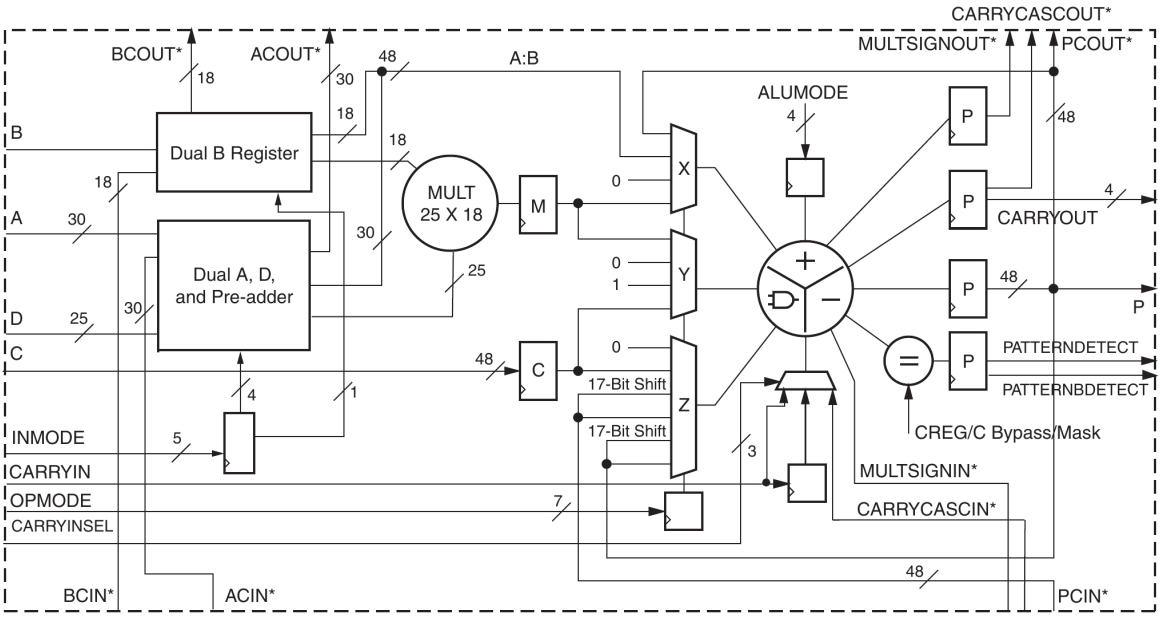

Figure 2.5 – The DSP48E1 top view [Xilinx Inc, 2018].

2.1.4 The Virtex 7 FPGA

This work adopts the XC7VX690T Virtex 7 FPGA Xilinx device. The adoption of this device is due to its availability in the laboratory and a large number of internal resources for prototyping [Xilinx Inc, 2016c]. Table 2.1 details some of this device available resources.

Table 2.1 – FPGA resources.

	CLBs	DSP Slices	36 Kb BRAM Blocks	Total BRAMs (Kb)	PCIe
XC7VX690T	693,120	3,600	1,470	52,920	3

These resources correspond to the area restriction the work must follow. For each layer of the CNN, it is evaluated the number of required resources in such a way to make possible to prototype the design in the target device.

2.2 Machine Learning

Machine learning is an area in Artificial Intelligence (AI) responsible for algorithms that can perform a task with no explicit programming to do so. The principle is to enable computers to learn from experience to solve problems that would be otherwise difficult (or impossible) to be formally described by people.

A learning algorithm is an algorithm that is capable of learning from data. Formally, Mitchell [Mitchell, 1997] states that "A computer program is said to learn from experience (E) with respect to some class of tasks (T) and performance measure (P), if its performance at tasks in T, as measured by P, improves with experience E" . A task T may be a description of how the algorithm should treat an input example, for instance, the problem of image classification (described in subsection 2.2.1), is a machine learning task. The experience E is the set of examples fed to the algorithm (a set of images in an image classification problem), while P is some form of quantitative performance measurement. It can be accuracy, error rate, or any other measure that is function of the application.

2.2.1 Image Classification

One of the core problems in Computer Vision is the task of classifying an image. It consists in assigning an image into a class label from a fixed set of classes. The problem has application in areas ranging from medical [Qing Li et al., 2014], defense [Yoshihisa Hara et al., 1994] and volcanology [Gabor Kereszturi et al., 2018].

Usually, a classification algorithm is split into two phases. The first phase is training. This phase is responsible for learning the model set of parameters, enabling the model to map a certain input x to a class label k. Formally, a set of inputs in $\mathbf{R}^n$ (a tensor, such as images) is used to induce parameter values so that a model f, which relates the input examples to the set of k classes. Thus, $f : \mathbf{R}^n \rightarrow \{1, ..., k\}$. Once training is complete, the model is put to infer any new, to the set of the known classes, example x. In other words, the model is ready to classify new inputs.

2.2.2 Deep Learning

Similar to what happens in any area of Computer Science, part of the performance of algorithms rely on how the input data is organized. In machine learning it is not different, the way the information is represented is paramount. Each piece of information consumed by a learning algorithm is called *feature*. Many tasks rely on hand-picked features to execute. Hence, prior to the execution of a learning algorithm, a set of features is extracted from the data and supplied to the algorithm.

The approach of handcrafted features may work for some simpler tasks. However, the search for features can be difficult, or even impossible, depending on the problem complexity. Suppose an image classification algorithm. In order to classify a single class, a cat, for example, it should be known all the features that make an image of a cat different from any other image. It would be possible to use as features the size and color of eyes, the fur, the tail, and so on. Nevertheless, each feature can appear in numerous variations of relative size, orientation, angle, light, etc. Hence, it is impractical to code every feature of every class in problems of this size.

Deep learning models solve this problem by representing features internally. Those models may describe a cat in terms of curves, shapes, and corners. According to Goodfellow et al., it extracts complex concepts in a hierarchical manner from less complex, raw data [Ian Goodfellow and Courville, 2016]. Moreover, in their words:

> *Machine learning is the only viable approach to building AI systems that can operate in complicated, real-world environments. Deep learning is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones.*

2.2.3 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a subset of deep learning models. ANNs provide a general model that can learn its parameters from data in a process robust to errors in the input data [Mitchell, 1997]. Besides the original inspiration on the biological brain, modern ANNs do not attempt to imitate the biological brain in every aspect. The human brain has approximately 10^{11} neurons, each connected on average to 10^4 others [Mitchell, 1997]. This enormously dense network in our brain gives more processing power than any computer ever built.

The fundamental unit in an ANN is the neuron (2.6). As described in [Haykin and Haykin, 2009], the neuron model is composed of three basic elements: *(i)* a set of synapses, where each synapse j, multiplied by a particular weight w_{kj}, is connected to a neuron k; *(ii)* an adder to sum up all the synaptic input values (and an additional bias); *(iii)* an activation function that limits the value produced by the adder.

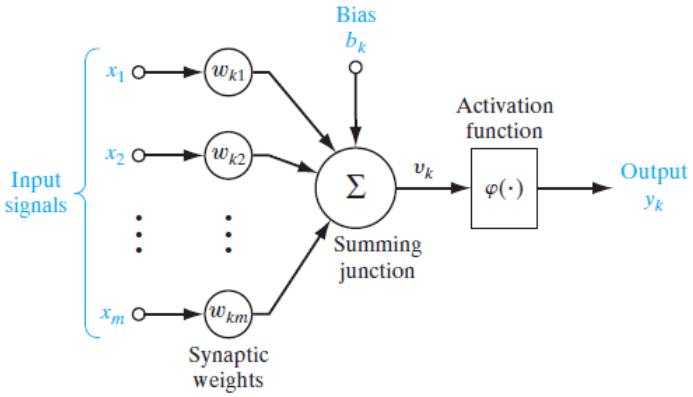

Figure 2.6 – The neuron model [Haykin and Haykin, 2009].

Neurons can be connected in a variety of ways (or architectures). The most common architectures are feedforward. For a network that is said to be feedforward, the data flows strictly in one direction, from input to output. It can also have one or multiple layers; they distinguish in the presence of hidden layers. A hidden layer is any layer between the first input layer of neurons and last output layer. An example of such network is shown in 2.7.

Despite the historical importance of Rosenblatt's model, the fact that the classes classified by the Perceptron needed to be linearly separable posed a limitation in applicability to the model. To solve this problem, the Multilayer Perceptron Network (MLP) was proposed. The MLP is a multilayer feedforward network, which means that it includes one or more hidden layers.

Haykin refers to three main characteristics of MLPs [Haykin and Haykin, 2009]: *(i)* the nonlinearity and differentiability of the activation functions presented on the neuron

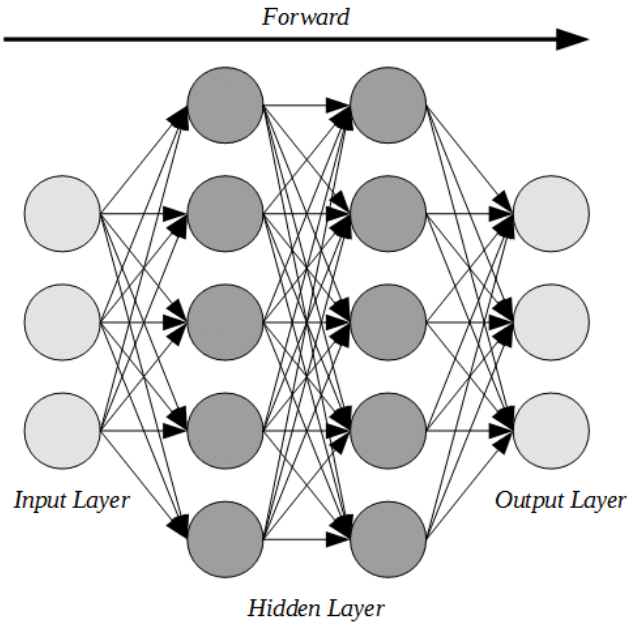

Figure 2.7 – A fully-connected Artificial Neural Network.

model; *(ii)* the hidden layers; *(iii)* the high degree of connectivity. Those characteristics give the model a great deal of generalization and performance for tasks like classification but make the learning (or training) process challenging.

The most popular choice for training such networks is the backpropagation algorithm. In general terms, it occurs in two phases. The **forward** phase when the input data is provided to the input layer, and the data flow layer by layer through the network up to the final, output, layer. The **backward** phase, calculates the error by comparing the result produced by the first phase with the expected output. Then, the error propagates through the network, layer by layer, in the opposite direction of the first phase. During the passage through the layers, the weights are adjusted accordingly to the error of each specific layer. As the focus of this work is on the first phase, also called inference, aspects related to the training of ANNs are out-of-the-scope of this work.

2.2.4 Convolutional Neural Networks

The idea of an artificial neural network aimed for visual pattern recognition came in 1983 by Fukushima, Miyake, and Takayukiito [Kunihiko Fukushima and Ito, 1983]. On the article entitled "*Neocognitron: A neural network model for a mechanism of visual pattern recognition*" the authors were the first to introduce the architecture that became known as Convolutional Neural Network (CNN). Their architecture was capable of classifying digits from handwritten numerals. Later, in 1998, Lecun et al. were able to train a network of convolutional neurons (Neocognitron) using the backpropagation algorithm [Yann LeCun et al.,

1989]. Since then, CNNs have grown in depth and size. Nowadays, it is possible to find networks with hundreds of layers [He et al., 2016]. Fortunately, the building blocks (convolutional and fully-connected layers) remain the same. Here, only the layers concerning Alexnet will be detailed. However, most of them are still used in modern networks.

A convolution layer can be understood as a fully-connected layer which employs the mathematical operation of convolution instead of the usual matrix multiplication. Lecun and Bengio pointed out the three main ideas that explain why convolutions are exceptional in dealing with translated and locally distorted inputs (such as images): *(i)* **local receptive fields**; *(ii)* **shared weights**; *(iii)* and, optional **subsampling** [LeCun and Bengio, 1998].

The use of local receptive fields means that a convolutional neuron takes its input from a reduced local of the input instead of the whole input as in a regular ANN. It also means that the filter (or set of weights) belonging to a set of neurons is trained to extract features that can appear in any location in the input. Additionally, it saves memory space once the same set of weights is shared across the input. There is, also, the subsampling that reduces dimensionality to higher layers. The process of subsampling comes from the idea of complex cells feeding information to simpler cells in the visual cortex as described in [Kunihiko Fukushima and Ito, 1983].

The process in a CNN convolutional layer can be described as follows: it accepts an Input Feature Map tensor (each map may also be called channel) and another tensor of filters [Sze et al., 2017]. Then, each filter window is convolved with (and slid across) its respective input channel forming a new set of feature maps. Next, a vector of bias can be added to the feature maps, generating the final Output Feature Map tensor. The process can be more readily understood by looking at Figure 2.8.

Formally, Equations 2.1 and 2.2 describe the convolution and fully-connected layers, respectively.

$$\mathbf{O}[c][x][y] = \mathbf{B}[c] + \sum_{k=0}^{C-1} \sum_{i=0}^{Width-1} \sum_{j=0}^{Height-1} (\mathbf{I}[k][Sx+i][Sy+j] * \mathbf{W}[c][i][j]) \tag{2.1}$$

where: c, x and y are the current output channel, the horizontal and the vertical position respectively; also, C is the total number of input and filter channels, Width and Height corresponds to the filter size; S is the stride[1], and $\mathbf{O}$ is the output, $\mathbf{I}$ the input, and $\mathbf{W}$ the filter tensors and $\mathbf{B}$ the bias vector.

$$\mathbf{O}[y] = \mathbf{B}[y] + \sum_{k=0}^{C-1} \sum_{i=0}^{Width-1} \sum_{j=0}^{Height-1} (\mathbf{I}[k][i][j] * \mathbf{W}[k][i][j]) \tag{2.2}$$

[1]The value of stride gives the number of positions that the filter slides over to the next window.

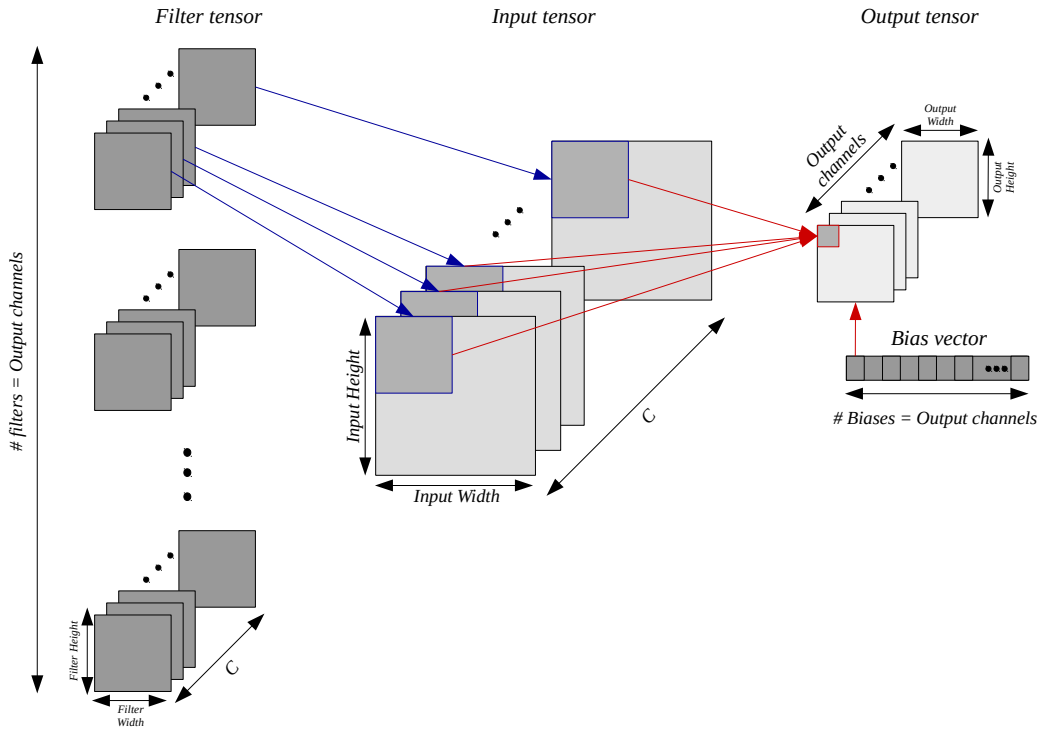

Figure 2.8 – The convolution operation in CNNs.

where: **O** is the output vector plus **I** and **W** which are the input and weight tensors, as **B** is the bias vector. The index *y* gives the current output neuron, *C* is the number of input channels, while *Width* and *Height* are the input width and height respectively.

Among convolutional and fully-connected layers, there can be other layers, such as pooling, activation, and normalization.

Pooling layers are applied to reduce dimensionality and avoid overfitting [Krizhevsky et al., 2012]. Typically, pooling is applied to each channel separately. The most two common forms of pooling are mean and max-pooling, the latter is used by Alexnet. A max-pool consists in a window of reduced size that passes over the input channel, in an overlapping fashion in the case of Alexnet, returning the maximum value. Besides, this value is written to the output channel. More precisely, Alexnet uses windows of size 3 x 3 spaced by two positions each in all of its pooling layers. There are three pooling layers in the Alexnet; they follow the convolutional layers number one, two and five.

Activation is an important operation that is, in fact, part of the neuron model. Activation functions add non-linearity to the network. There exist several functions that can be used: sigmoid, hyperbolic, exponential and Rectified Linear Unit (ReLU) are some of the functions found in the literature. Alexnet uses the ReLU activation function [Nair and Hinton, 2010]. ReLU differs from other functions mainly because of its non-saturating (to values greater tha zero) characteristic that makes the network training possible [Krizhevsky et al., 2012]. Equation 2.3 presents an example of the ReLU activation function.

$$ReLU(x) = max(x, 0) \tag{2.3}$$

where: x is the value of each element in an Output Feature Map.

Normalization layers are applied (usually following activation layers) to control the distribution in feature map values and, hence, they help to speed up learning and increase accuracy [Krizhevsky et al., 2012]. The normalization used across Alexnet layers is the Local Response Normalization (LRN). The authors in Alexnet explain that LRN aids the learning by increasing generalization, reducing top-1 and top-5[2] errors rates by 1.4% and 1.2% respectively [Krizhevsky et al., 2012]. Krizhevsky et al. introduced LRN as in expression 2.4.

$$b^i_{x,y} = \frac{a^i_{x,y}}{(k + \alpha * \sum_{j=max(0,i-\frac{n}{2})}^{min(N-1,i+\frac{n}{2})} (a^j_{x,y})^2)^\beta} \tag{2.4}$$

where: $a^i_{x,y}$ denotes the activated neuron output of filter *i* at position (x, y) and $b^i_{x,y}$ the normalized output at the same position. The idea is that a neuron gets its value by "averaging" over the N neighboring neurons. The other variables k, α, and β are hyper-parameters set by the network designer.

Nonetheless, the LRN normalization has not been used in most modern deep learning architectures. It is observed the adoption of other normalization functions instead of the LRN, such as the Batch Normalization [Ioffe and Szegedy, 2015]. During the development of this work, it was decided that the hardware implementation of an LRN layer would not be carried on. Also, the LRN layer was omitted from software simulations to enable a compatible verification against the hardware version.

The last operation, **padding**, is not a layer, but it is used in most networks. This operation is useful to match the input size that feeds a layer with specific configurations. In convolutional layers, for example, sometimes it is desirable to use a particular stride and filter size that will not match the input feature map dimensions. Hence, padding shall be applied. In the case of Alexnet, it uses zero-padding. It consists in adding rows and columns of zeros around the original input map, creating a new map whose dimensions are acceptable by the next layer (map size matches stride and filter size).

2.2.5 CNN Architectures

The literature is rich in CNNs solving problems in several areas. However, it is possible to trace a historical trend of the development of CNN architectures by presenting the first trainable CNN, LeNet, and the CNNs that recently scored best in the Imagenet

[2] In image classification applications, it is common to use the terminology of top-1 and 5. Top-N takes into account the N classes that scored the highest probabilities among all classes in the output vector.

contest. This Section does not aim to present the architectures in full detail but introduces general concepts related to the state-of-the-art in CNNs.

The first work to employ a trainable Convolutional Neural Network was **LeNet** [Yann LeCun et al., 1989]. In 1998, the architecture proposed by LeCun et al. was a seven-layers deep architecture that was able to recognize hand-written digits using 60 thousand parameters.

Later, in 2012, the **Alexnet** won the Imagenet contest [Krizhevsky et al., 2012]. The architecture is deeper and has more parameters than LeNet. The Alexnet consists of five convolutional layers of 11x11, 5x5, and 3x3 filters. With its 60 million parameters, Alexnet achieved a Top-5 error rate of 15.3% in the Imagenet dataset. The next year, the **ZFNet** won the same contest [Zeiler and Fergus, 2014]. The architecture leveraged the Alexnet architecture, and by tuning of hyper-parameters, the ZFNet achieved a Top-5 error rate of 14.8%.

Then, in 2014, **GoogLeNet** [Szegedy et al., 2015] achieved the surprisingly 6.67% Top-5 error rate. It was close to a human-level performance in image classification. The GoogLeNet reduced, by using small filters, the number of parameters from the 60 million in Alexnet to 4 million. However, it introduced new elements such as the Inception module and an architecture of 22 layers deep. Also in 2014, the **VGGNet** attained 7.3% Top-5 error rate in the Imagenet dataset [Simonyan and Zisserman, 2014b]. Its architecture is more regular that GoogLeNet, however, its 16 layers use 138 million parameters.

At last, in 2015 a CNN called **ResNet** beat human-level performance at the dataset [He et al., 2016]. The ResNet achieved 3.57% Top-5 error rate. The architecture is composed of 152 layers, with 60.2 million parameters. Also, it makes use of batch normalization and other new techniques.

Among all cited CNNs, the architecture chosen to be implemented in this work is the Alexnet [Krizhevsky et al., 2012]. There are two main reasons for this choice. First, the network is a state-of-the-art large-scale CNN [Li et al., 2016]. Alexnet enables to exercise all the crucial operations of any modern CNN such as multiple layers of cascaded convolutions, fully connected layers, activation, and max-pooling layers. Second, the Alexnet is a popular choice of network to many authors who are working on dedicated hardware architectures, which makes it possible to run a future comparison against other implementations.

3. ALEXNET

Alexnet classifies images using an eight-layer deep architecture, five of which are convolutional, and three of which are fully-connected. The input is a 227 x 227 pixels RGB image. Further, the output is a column vector containing the probability of each of the 1000 classes from the ImageNet dataset [Russakovsky et al., 2015]. The overall network architecture is depicted in Figure 3.1.

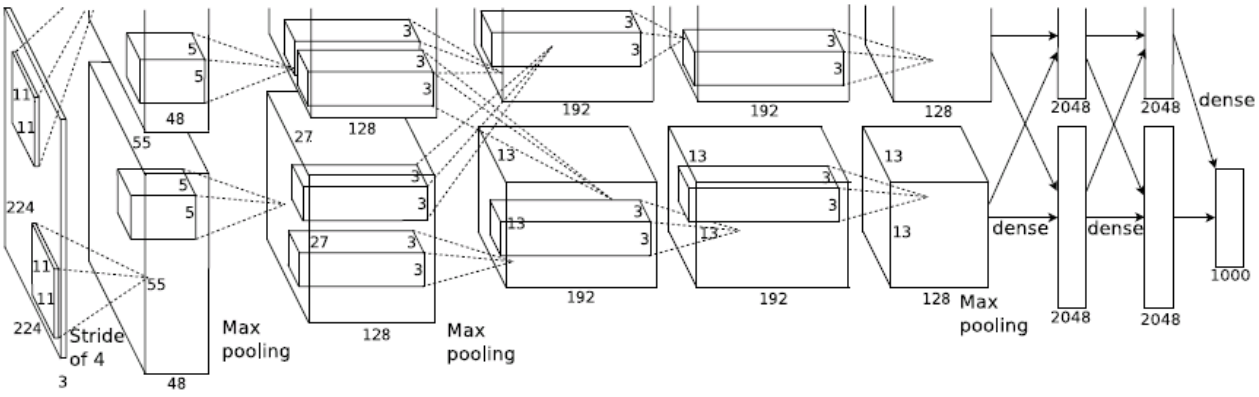

Figure 3.1 – The Alexnet CNN [Krizhevsky et al., 2012].

As Figure 3.1 shows, the network architecture cascades five convolutional (CONV) layers ending with three fully-connected (FC) layers. From left to right, there is the input layer, the subsequent hidden layers (four convolutional and two full-connected), and the final fully-connected output layer. The parameters of each layer are detailed in Table 3.1.

Table 3.1 – Shapes for the Alexnet CNN.

Layer	Operation	Input Size (padded)	Weight/Filter Size	Output Size
1	CONV	227x227x3	11x11x3 (x96)	55x55x96
1	MAX-POOL	55x55x96	3x3	27x27x96
2	CONV	31x31x96	5x5x96 (x256)	27x27x256
2	MAX-POOL	27x27x256	3x3	13x13x256
3	CONV	15x15x256	3x3x256 (x384)	13x13x384
4	CONV	15x15x384	3x3x384 (x384)	13x13x384
5	CONV	15x15x384	3x3x384 (x256)	13x13x256
5	MAX-POOL	13x13x256	3x3	6x6x256
6	FC	6x6x256	6x6x256x4096	4096
7	FC	4096	4096x4096	4096
8	FC	4096	4096x1000	1000

3.1 Alexnet Resource Estimation

The parameters (Table 3.1) can be further explored to estimate the memory and computational resources needed to run Alexnet. The amount of memory required is depicted in Table 3.2. Memory estimates were obtained using Equations 3.1 for convolutional, and 3.2 for fully connected layers.

$$conv_{bits} = (F * F * C * K + B) * W \tag{3.1}$$

where: F is the filter size, C the number of channels (depth), K the number of filters and B the number of biases for a convolutional layer. The word size assumed for the calculations is 32 bits and is represented by W.

$$fc_{bits} = (W_{in} * H_{in} * C_{in} * N + B) * W \tag{3.2}$$

where: W_{in} and H_{in} are the input width and height respectively; C_{in} is the number of input channels, and N is the number of neurons for the layer being evaluated. The number of biases for the layer is B, and the word size is W.

Table 3.2 – Memory required by each CONV and FC layer (assuming a 32-bit numeric format).

Layer	Input size (bits)	Weights/filter plus bias size (bits)
1 - CONV	4,949,856	1,118,208
2 - CONV	9,300,992	19,668,992
3 - CONV	5,984,256	28,323,840
4 - CONV	2,088,960	42,479,616
5 - CONV	2,084,864	28,319,744
6 - FC	294,912	1,208,090,624
7 - FC	131,072	268,566,528
8 - FC	131,072	65,568,000
Total	**24,866,912**	**1,662,135,552**

Additionally, Table 3.2 offers another interesting distinction between convolutional and fully-connected layers that is the number of required parameters. The saving in parameter space is one of the attractive characteristics of convolutions. Because of the small size sliding window, convolutions do not require large filters – as in the fully-connected layers that require orders of magnitude more memory than convolutions. However, the tensors being moved across the CNN convolutional layers namely input and output feature maps, consume most of the memory space. In other words, convolutional layers need more buffers for storing feature maps while fully-connected layers need larger buffers to store parameters.

Similarly, it is possible to estimate the number of multiply operations for convolutional and fully-connected layers by the summations in Equations 2.1 and 2.2. For the sake of brevity, only the number multiplications at each layer are shown here (Equation 3.3).

$$conv_{mults} = (F * F * C_{in}) * C_{out} * H_{out} * W_{out} \tag{3.3}$$

where: $conv_{mults}$ denotes the total number of multiply operations in a convolutional layer, where F is the filter size, C_{in} the input channels, C_{out} the output channels, and H_{out} and W_{out} are the height and width of the layer output.

On the other hand, for a fully-connected layer, the number of multiplications is given by Equation 3.4

$$fc_{mults} = (C_{in} * H_{in} * W_{in}) * N \tag{3.4}$$

where: H_{in}, W_{in}, and C_{in} denote the input feature map size (height, width, and depth), and N is the the number of neurons in the layer.

From Equations 3.3 and 3.4, Table 3.3 presents the number of multiply operations for each layer in the Alexnet. It demonstrates the computationally-intensive aspect of a modern CNN. With the Alexnet example, another interesting point can also be made about the two types of layer. When analyzing the number of multiply operations required by each layer, it is clear that convolutional layers tend to demand more computation than fully-connected layers.

Table 3.3 – Alexnet total number of multiply operations.

Layer	Multiply Ops
1 - CONV	105,415,200
2 - CONV	447,897,600
3 - CONV	149,520,384
4 - CONV	224,280,576
5 - CONV	149,520,384
6 - FC	37,748,736
7 - FC	8,388,608
8 - FC	2,048,000
Total =	**1,124,819,488**

3.2 Software Evaluation

The first step towards a more significant understanding of the particularities in the implementation of CNNs was to execute and evaluate a publicly distributed version of Alexnet in software. The second step was to implement a simpler version in C language where it

would be possible to get familiar with the data structures involved in a CNN forward execution.

3.2.1 Public Software Distribution

Besides the recent improvements in the available hardware as seen in the form of new GPU, ASIC, or FPGA platforms [Nvidia, 2018, Jouppi et al., 2017, Ovtcharov et al., 2015], the observed increase in software frameworks has also influenced the development of new deep learning networks. There is a multitude of publicly available, open-source, frameworks tailored to the development of artificial neural networks. They facilitate the portability of models across developers and add the transparency necessary to experiment with different hardware platforms [Sze et al., 2017].

For example, The Caffe framework was made public in 2014 by the University of California, Berkeley [Jia et al., 2014]. In 2015, Google launched the Tensorflow framework where it is possible for the developer to smoothly run the CNN either in GPUs or conventional CPU architectures just by editing a few lines of its Python script [Abadi et al., 2016]. Facebook also has its open-source framework, called Torch [Collobert et al., 2002]. Despite all the cited frameworks being capable of running the Alexnet, a portable framework with no particular library dependencies was needed to enable simulation on an embedded environment. Consequently, the Tiny-Dnn library was used to evaluate the Alexnet in an embedded environment [Nomi, 2018].

The simulation was carried out on the Open Virtual Platform (OVP) [Imperas, 2018]. OVP provides a platform for embedded systems development, supplying models for a variety of processors such as PowerPC, RISC-V, MIPS, and ARM. Next, it was possible to cross-compile the Alexnet C++ implementation, provided along with the Tiny-Dnn, targeting the ARMv8 instruction set. Thus, the executable file can be loaded and run over the simulated ARM Cortex-A57 single core processor. Additionally, the simulated system was required to embed the Linux kernel (version 4.3.0). The operating system was necessary to provide some basic system calls as the Tiny-Dnn network implementation is not “bare-metal.”

The first evaluation regards the number of ARM ISA instructions used during classification of a single image on the Alexnet. To do so, it was essential to separate the instructions executed during the Linux system boot from the instructions that were employed during classification. The applied method consists in running several executions of two different Linux initialization scripts. The first one simply waited for the system to boot and halted. The second script, however, performed a classification after the boot is completed and then halted the system.

Further, after several executions of both scripts, the average number of instructions can be calculated for booting the system and for booting plus a single Alexnet execution. The

difference in the number of instructions gives the average instructions count for classifying an image on the Alexnet Convolutional Neural Network. Table 3.4 presents the instruction count for five executions of both scripts. The simulation was executed multiple times with the same input because it was not possible, to the author knowledge, to stop the instruction counting precisely after the halting of the operating system.

Table 3.4 – Average instruction count ($*10^9$) by single image classification, obtained from OVP reports.

Execution	Linux boot plus Alexnet	Only Linux boot
1	739.4	430.5
2	802.4	510.5
3	655.5	437.0
4	743.6	538.0
5	746.3	445.2
AVG =	737.5	472.2
Average instructions by classification:		**265.2**

After evaluating the total number of instructions, the Alexnet execution time was measured using the Linux command *time*. The result is given in Table 3.5.

Table 3.5 – Time for the classification of a single image with the Tiny-Dnn library, measured from the OVP simulation.

Alexnet execution time (in seconds)		
real	user	sys
34.24	32.57	1.67

The second evaluation proposed for the Tiny-Dnn Alexnet implementation is memory consumption. Here, the Valgrind tool was used to perform a memory consumption evaluation [Nethercote and Seward, 2007]. The analysis was performed on the host machine, once the simulated environment does not provide all the necessary Linux packages for the Valgrind installation. Regardless, the same source code was profiled which gives roughly the same number of memory allocations. In Figure 3.2, the memory heap consumption is portrayed. The peak memory usage reaches over 520 MB.

3.2.2 Implemented Software

Next, the CNN was implemented in the C language. First, this was mainly motivated by the initial modeling of the problem. Second, to facilitate the latter hardware verification (providing verification "checkpoints" in the CNN data flow). Lastly, the software written in C can be compiled to run in bare-metal platforms enabling more accurate statistics from the OVP simulator that serve as a reference in performance for the hardware. Appendix A shows the code for the Alexnet CNN written in C.

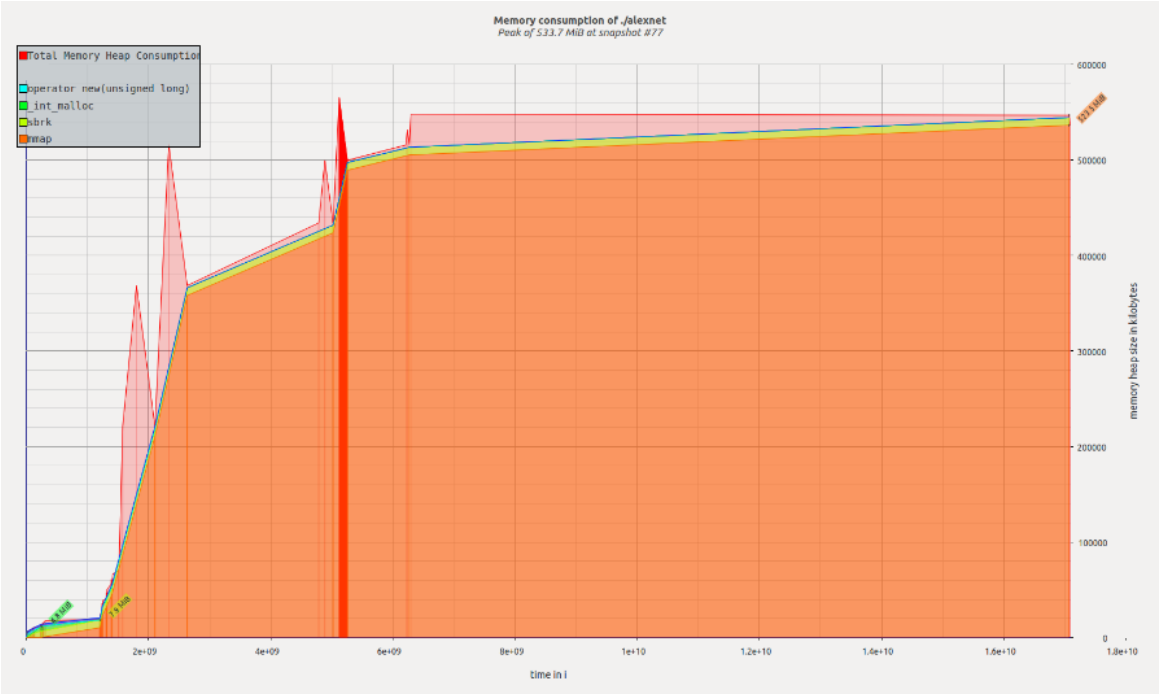

Figure 3.2 – Memory heap consumption for the C++ Alexnet inference.

The implemented software was validated using the Tiny-Dnn distributed version. To perform the validation, the output vector of both implementations were compared after the classification of 11 different images. After the executions, the same Top-5 classes were retrieved by the two output vectors on all images. However, when comparing the output vectors from the last fully-connected layer, a small error was found (Table 3.6) which can be explained by differences in the code of arithmetic functions provided by C++ libraries, used by Tiny-Dnn, and the software written in C. The error was measured as the element-wise absolute difference, the maximum difference found after the classification of 11 images is presented in the Table.

Table 3.6 – The absolute maximum error between implemented and distributed software.

Image	Max Absolute Error
amplifier	0,000499815
ball	0,000995398
cat	0,001348913
cellphone	0,002319574
electricGuitar	0,000403762
inkjetPrinter	0,000041589
laserPrinter	0,005688460
monitor	0,000040695
phone	0,000801794
sink	0,000101125
slippers	0,001310173
Avg Max Absolute Error:	**0,001231936**

Simulated on the ARM Cortex-A5, the bare-metal Alexnet classified an image executing **116.1x10^9** ARMv7 ISA instructions (OVP report). Moreover, the total simulated time was **1,161.13** seconds and the memory usage peak, profiled by the Valgrind tool, achieved 250.5 MB (Figure 3.3). Note that using a simpler embedded processor (ARM Cortex-A5) than ARM Cortex-A57 the number of instructions reduced by 50% because there is no system calls neither the complexity of object-oriented structures. On the other side, the execution time increases 37 times.

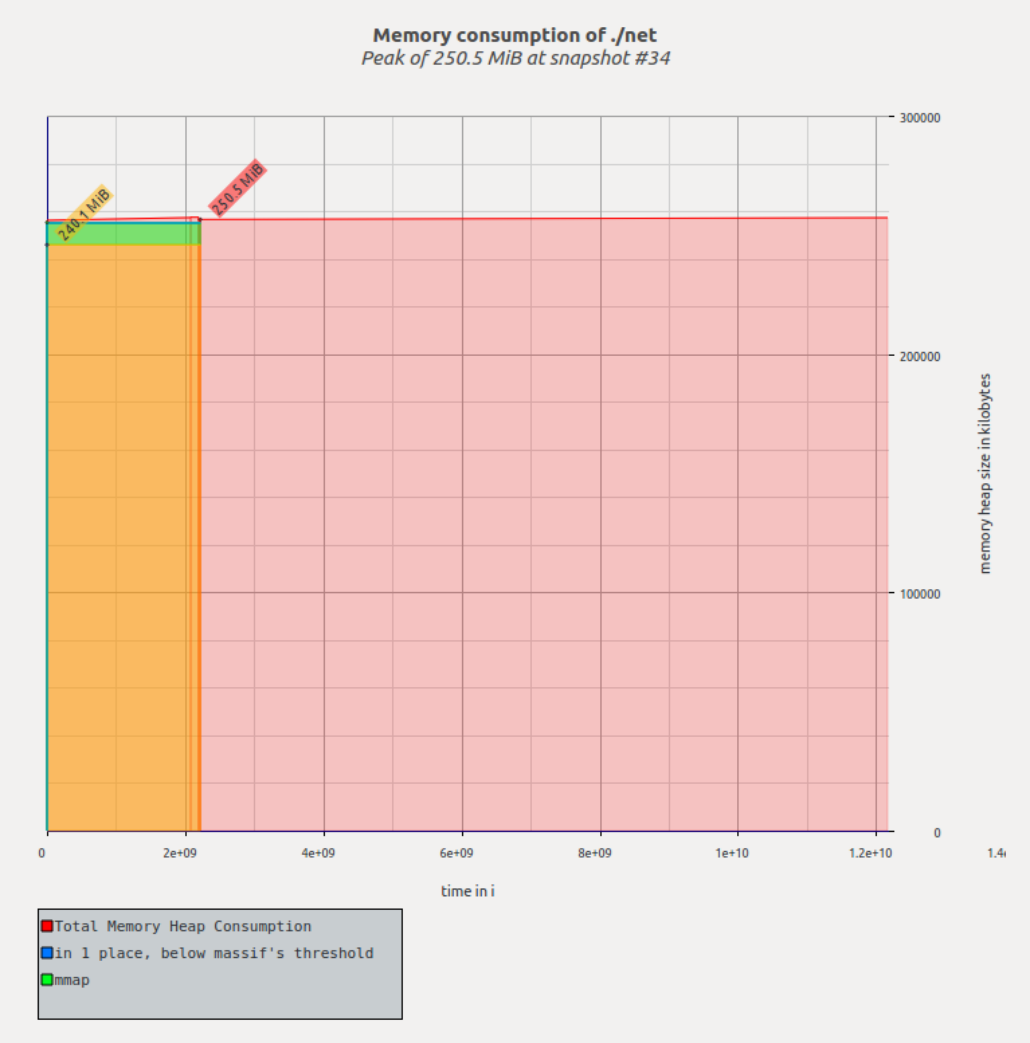

Figure 3.3 – Memory heap consumption for the C Alexnet inference.

Executing the software on the host machine, an Intel Core i7 at 3.4GHz with 8G bytes of memory, the classification took only **3.234** seconds. The result is impressive, and future works aim to explain the optimizations made by the compiler and the architecture to execute all arithmetic and memory accesses required during an image classification.

Concluding, the implemented software was validated using the Tiny-Dnn (C++) framework and simulated on a virtual ARM platform. Also, it offers means for performing an incremental verification of the hardware by producing a set of files that contain the values of specific points in the architecture. Additionally, it makes possible to use the simulated environment as a baseline for performance analysis.

4. HARDWARE INFRASTRUCTURE

This Chapter presents the modules used in the hardware implementation. Also, it details the memory architecture, the floating-point format, and the arithmetic operators.

4.1 Generic Memory Module

One of the main goals behind this project is to make the hardware modules as generic as possible. It enables, via off-line configuration, the design to implement a CNN with any set of parameters and size.

The first issue addressing the generality of the design is the memory. Throughout the design, the same memory module is instantiated for both input and output feature maps and weight values. Hence, the first configuration parameter of this memory is **depth**, which allows configuring the amount of BRAMs for each instance. Another parameter is **width**. This second parameter is important so the memory will not pose a problem for changes in numeric format. In other words, it is possible to tweak the numeric precision with no need to rewrite memory modules.

Specifically, the memory is configured with *B* blocks and each block has *W* BRAMS (of 1 bit width each), where *W* is the word-length. Figure 4.1 gives the schematic of a generic memory instance configured with 3 blocks and 29-bit word (width of the floating-point number used in this work and described in the next Section). Besides, the total BRAM size being equal to 36 Kb, the effective data storage is 32 Kb since for every 8 bits there is a parity bit. In Figure 4.1, the total number of addressable 29-bit words is 96,000. This Figure shows the schematics for the memory module implemented for this work. The module called **GenMem** is configurable in depth to store the set of weights and feature maps that vary in size across layers.

Any convolutional layer works with at least three data structures: filter, input, and output feature map. For instance, the Alexnet layer one input is a 227x227x3 tensor (Table 3.1), which corresponds to 154,587 29-bit floating-point words. Dividing it by 32Kb, this corresponds to 5 blocks (***B***), or 145 BRAMs. To store the output feature map produced by the max-pool in layer one, according to Table 3.1, 69,984 words are required, resulting in 3 blocks, or 87 BRAMs. Finally, the layer one filter and bias take 34,944 words, resulting in 2 blocks, or 58 BRAMs. Thus, for the input, output, and filter, 10 blocks of generic memory are used in layer one. This represents 290 BRAMs, corresponding to 20% of the available resources in the target FPGA.

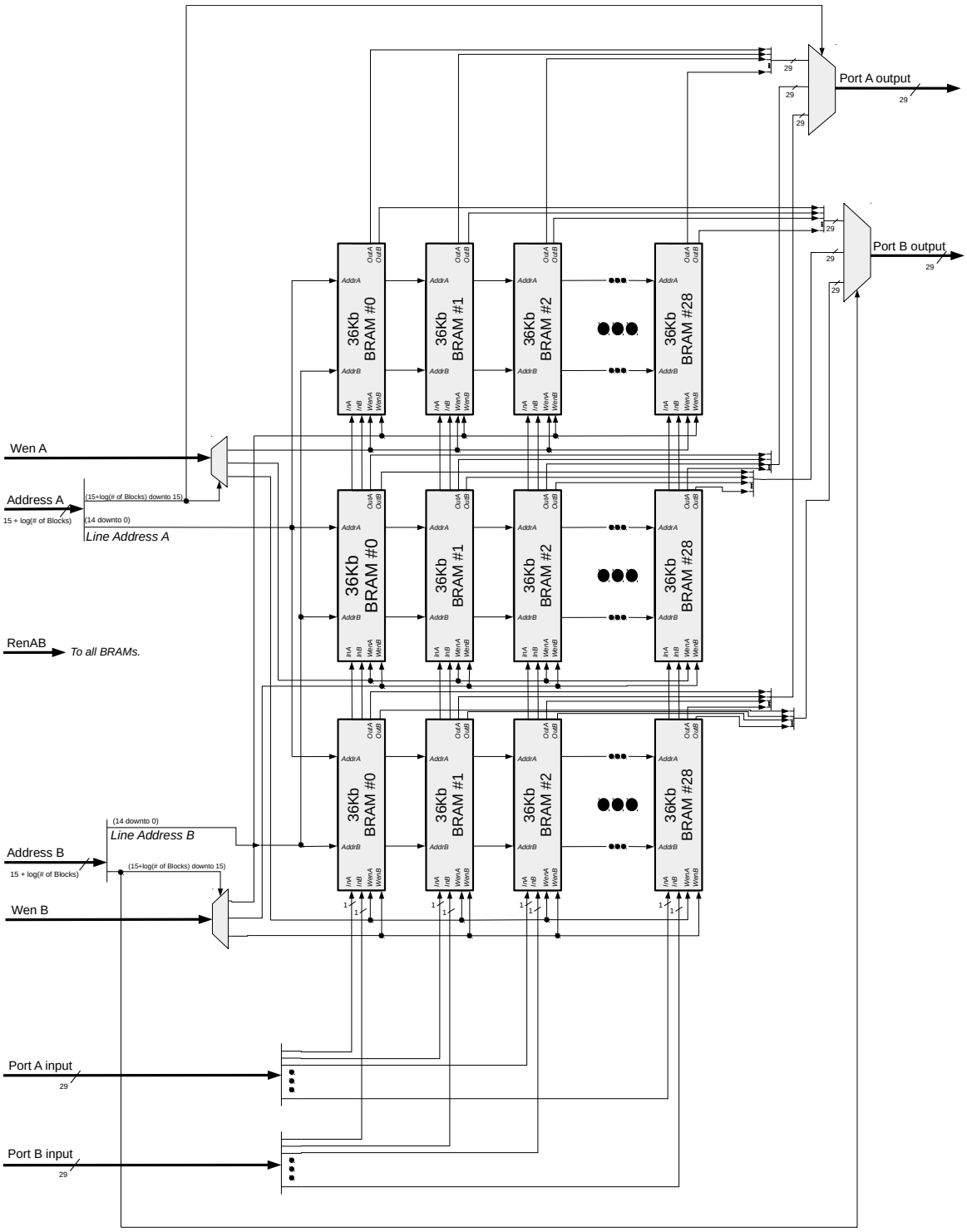

Figure 4.1 – Schematic of the memory module (with a 3 blocks and 29-bit wide configuration).

4.2 Floating-Point Format

Despite existing studies pointing out that the use of fixed-point has a little impact on the performance of deep learning accelerators [Jiao et al., 2017, Courbariaux et al.,

2015, Gupta et al., 2015], this work adopted floating-point representation to minimize the accumulated error between layers and to make it easy the verification against the software version. Our work adopted the Flopoco framework for floating-point arithmetic [de Dinechin and Pasca, 2011]. The framework provides a list of configurable arithmetic operators tailored for FPGAs, along with its proper floating-point format.

The format used across all modules of this work is the Flopoco floating-point configured to an 8-bit exponent and 18-bit mantissa (Figure 4.2). The exponent width was chosen based on the IEEE-754 standard that uses an exponent of same size [IEEE, 2008]. However, the mantissa was set to the DSP block input width of the target FPGA [Xilinx Inc, 2018]. It was necessary to fit the significand part to the DSP block input, so the integer multiplications require a single block to be performed.

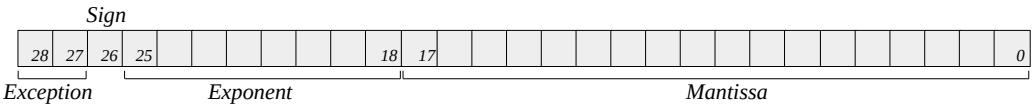

Figure 4.2 – The adopted floating-point format.

4.3 Arithmetic Operators

The Flopoco framework offers an extensive list of operators[1]. This section only presents the ones instantiated in this design.

4.3.1 Multiplier

Multiplication is at the core of convolutional and fully-connected layers. With the Flopoco floating-point multiplier at hand [de Dinechin and Pasca, 2011], it was necessary to modify it, by instantiating the DSP Block. The DSP block, which executes the *Integer Multiplication* (Figure 4.3), was manually inserted in the multiplier VHDL code from Flopoco. Afterwards, a control logic around the multiplier was implemented to control and synchronize the operation with other modules (not depicted in Figure 4.3).

The resulting module is capable of multiplying two floating-point numbers in three clock cycles. Also, it uses 205 LUTs, 166 flip-flops, and a single DSP block.

[1] http://flopoco.gforge.inria.fr/

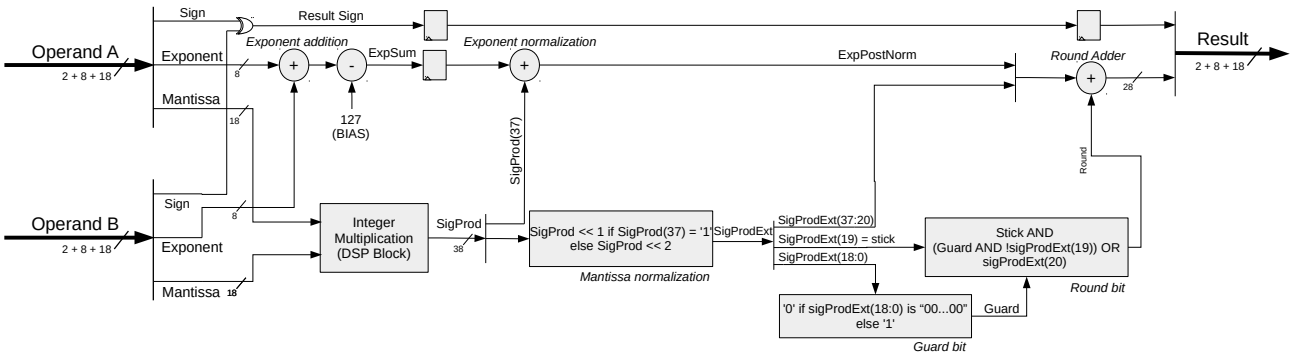

Figure 4.3 – Floating-point multiplier top view.

4.3.2 Adder

Another important operation in CNNs is addition. Most modern accelerators and hardware implementations adopt multiply and accumulate (MAC) modules [Sze et al., 2018, Sze et al., 2017, Venkataramani et al., 2017]. The obvious advantage on using MACs is precision, because the accumulator comes right after the multiplication module, and the result of the first operation is not rounded. However, in our case, multiplication and addition could not be "glued" together due the the adopted architecture, which will be presented in the next Chapter.

Conceptually, the floating-point addition takes more area and time than multiplication to execute. Flopoco implements both single and dual path floating-point adders [de Dinechin and Pasca, 2011]. Figure 4.4 shows the dual path floating-point adder available in the Flopoco framework. According to Muller in [Muller et al., 2010], the dual path architecture is optimal for FPGA designs once it has little area overhead compared to other single path implementations and the shifter amortizes the delay in the leading-zero counting. In a dual path adder, first, the exponents are compared and the operands possibly swapped. Next, the operands may be shifted to align the mantissas and one of them complemented in case of subtraction. Then, based on the difference of the exponents, the result of the close or far path is selected. The close path is chosen when the case is a subtraction of inputs with exponents that differ by at most one bit. One the other hand, the far path is chosen for adding or subtracting numbers with distant exponents (at least two bits).

The Flopoco dual path adder tailored to this work performs a floating-point addition in six clock cycles, taking 324 LUTs, 18 shift-registers (SRL) and 208 flip-flops.

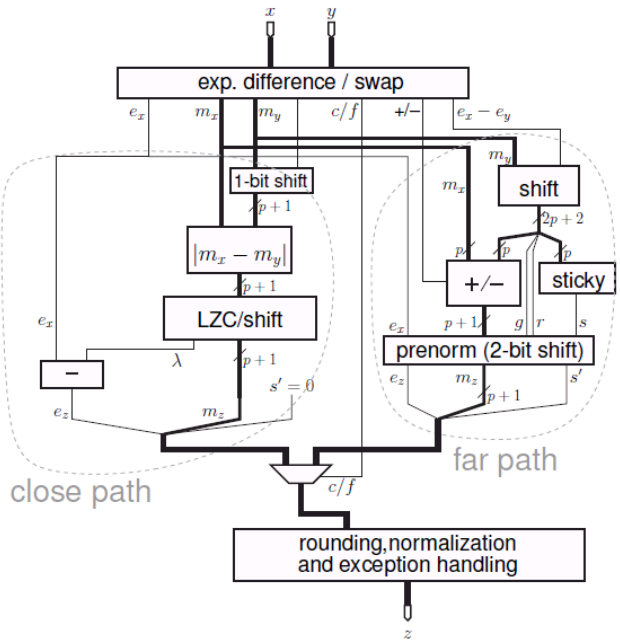

Figure 4.4 – The dual path floating-point adder architecture [Muller et al., 2010].

4.3.3 Fixed-Point Converter

The last Flopoco operator used is a converter to fixed-point format. Conversion was necessary to perform comparisons in the max-pool layer. The layer, explored in more detail in the next Chapter, first converts all the input values to fixed-point, then finds the maximum value among them by comparing their fixed-point representation. Basically, the conversion consists of a shifter, by the amount in the exponent, and a possible truncation. The framework also offers conversion to fixed-point with rounding; however the additional hardware was not seen as necessary to perform the max-pool layer.

The choice for the fixed-point size required attention to avoid overflow and underflow. The fixed-point width was set to 32 bits, where the lower 15 bits are the fractional part, the next 16 bits represent the integer part, and the most significant bit gives the sign. This setting gives a range from -65536 (2^{16}) to 65535.9999695 ($2^{16} - 2^{-15}$), which is enough to perform comparison operations like *greater than* in the max-pool layer.

5. HARDWARE IMPLEMENTATION

Following a top-down approach, this Chapter presents the modules that compose the implemented architecture. First, a top view of the architecture is provided. The next sections detail the modules responsible for performing the convolution and max-pool layers, accessing weight and feature map memories, as well as the *multilayer* module, which can execute an unlimited number of convolutional layers sequentially. Concluding the Chapter, a brief discussion of the fully connected layers is offered since this layer was not implemented due to the lack of time to design it.

Figure 5.1 presents the proposed architecture to implement each layer of the CNN. The left memory (*A)* contains the pre-processed input image and memory *B* contains the set of weights for the first layer. The first layer includes the convolutional and max-pool modules that fill the output feature map memory (*C*). Next, the second layer reads memory *C*, executing the functions related to the second layer. Once layer two is completed, the *Multilayer* starts. The *multilayer* executes sequentially layers three, four, and five. Note that this *multilayer* structure enables to implement an arbitrary number of layers sequentially. The result of the fifth layer is written into the output feature map, in memory *G*. *This architecture is, in the Authors opinion, a relevant contribution, since it allows to implement CNNs with an arbitrary number of layers.*

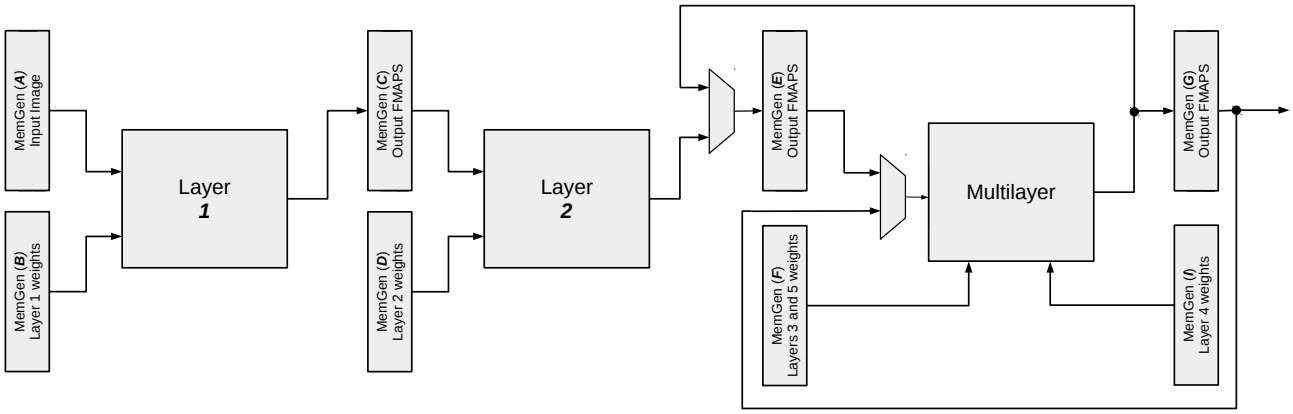

Figure 5.1 – Architecture top view.

The "Layer" encapsulates the convolutional and max-pool modules, interconnected in a "ping-pong" arrangement (Figure 5.2). While the producer (convolutional layer) writes on the upper buffer, the consumer (max-pool layer) reads from the lower buffer (*a*). In *b*, producer and consumer switch buffers and the consumer starts reading from the just filled buffer, as the producer writes over the already consumed data in the lower buffer. The first advantage in using "ping-pong" buffers is memory saving, once it requires less memory space since only two channels need to be stored instead of the entire feature map. For example, the first layer channel size is 55x55 while the entire feature map contains 96 channels. The second advantage is the latency reduction since the max-pool layer starts its

computations after each channel is done, not waiting for the complete feature map be ready for consumption.

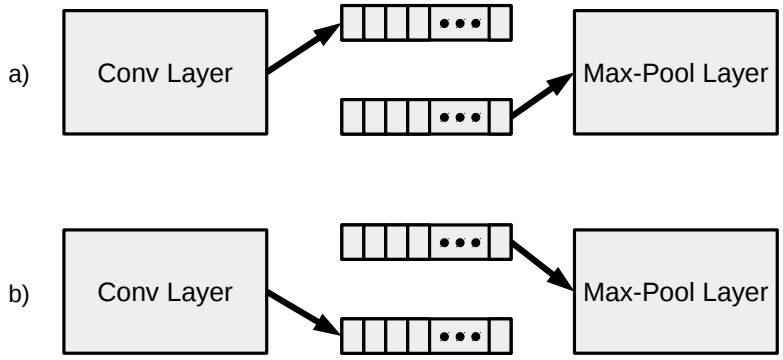

Figure 5.2 – Ping-pong scheme for interfacing Convolutional and Max-Pool layers.

5.1 Convolutional Implementation

A key module in the architecture is the one responsible to execute the convolution operation. It may be divided into six functional blocks, detailed in Figure 5.3.

a) address generator;

b) input buffers;

c) a pair of Finite State Machines (FSMs);

d) convolutional tree

e) neuron Adder;

f) ReLU activation.

Additionally, it is important to note that all internal modules are configurable and are instantiated using a set of user-defined VHDL *generics*, enabling the same VHDL entity to be used across all layers.

The **address generator** block (**A** in Figure 5.3) is responsible for filling the stride buffers with data from the input feature map (IFMAP) and filter memories. This module generates three addresses: to read from the IFMAP memory it uses the two reading ports, A and B, to fill the current and stride buffers; and to read from the filter weights memory, it generates only one address.

The **input buffers** (**B** in Figure 5.3) are a pair of shift registers that store one window of the feature map and data from the next stride positions (*stride buffer*), being their size equal to the convolution filter length. As detailed in 5.1.1, data is read serially from the input

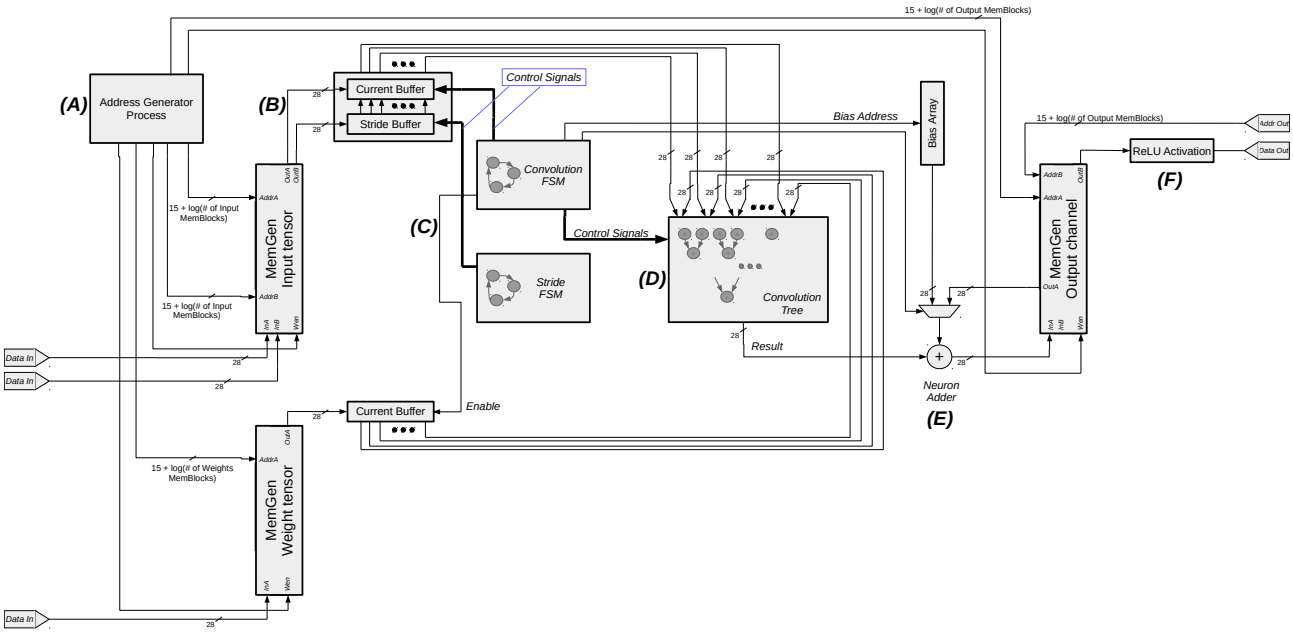

Figure 5.3 – Convolutional layer schematic.

memories. There is also another buffer, responsible for storing the weights of the current filter.

A **pair of FSMs** (**C** in Figure 5.3) control the operation of the convolutional tree and manage the write and read timing between the input buffers and the address generator. The FSMs need to be tightly synchronized to produce the right set of output values. This synchronization is important because the number of clock cycles taken by the convolutional tree, loading the current buffer, loading the stride buffer, and memory latency are different. In other words, the time spent by the convolution differs accordingly to the filter and stride size.

The **convolutional tree** (**D** in Figure 5.3) computes a window of convolution. At the leaves, it multiplies the IFMAP with the weight array. Then, it adds the multiplication results in a tree of adders. This module is also portable to any filter size. Afterwards, the convolutional tree produces a value, which is passed to the **neuron adder**. The neuron adder is in charge of summing up the results from the input channels belonging to the same output neuron. Additionally, this final adder adds the bias. To put it differently, when the result window of the first input channel is produced, there is no value to increment, so the bias is added to it and the neuron adder output is written to the OFMAP memory. Then, during the convolutions in the next channels, the result window is accumulated together with the value already stored in the OFMAP. This process avoids the extra step of the bias addition after the OFMAP completion.

Only when the next layer reads from the output feature map (OFMAP), the **activation** (**E** in Figure 5.3) function is executed. The address generated by the next layer is read from the memory, then, it passes through the activation module that performs the **ReLU**

operation (**F** in Figure 5.3). In other words, it only outputs values greater than zero from the OFMAP memory.

5.1.1 Buffers Operation

This section details the input buffers – **B** in Figure 5.3. As explained in Section 2.2.4, the convolution slides a window (the filter) through the IFMAP. This window reads a certain number of positions to execute the computation of the next neuron. Normally, the window size is greater than the stride, which makes part of the current window useful to the next convolutional window. This property was used to decrease the time spent in loading values from the IFMAP in the convolutional module.

For instance, in the Alexnet first layer, the window size is 11x11 with a stride of 4. It means that 121 values need to be loaded from the IFMAP memory at first. For the next window, the addresses increment four positions to the right. Thus, instead of reading all 121 values, only 44 new values are read (4 * 11). For instance, consider the buffers configured to a window size of 3x3 and stride of 2, as shown in Figure 5.4.

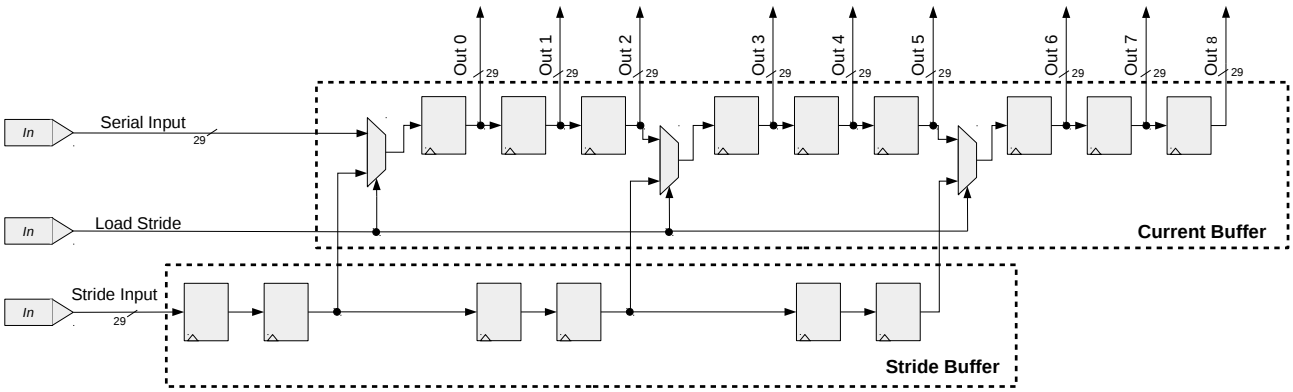

Figure 5.4 – Current and stride buffers.

At the beginning of an operation (a new window or a new IFMAP), both buffers are empty. Then, the *current buffer* is loaded with a window according to the addresses generated by the "address generator" block. After filling the current buffer, the FSM controlling the convolutional tree starts the computation. During the time spent by the tree, the second FSM (Stride FSM at Figure 5.3) switches the IFMAP memory port to the *stride buffer* and the address generator starts to request the next two addresses for each line in the window. Then, the *stride buffer* is ready to be loaded into the *current buffer*, which is done when the Convolution FSM pulses the *Load Stride* signal. Finally, the *current buffer* has the contents for the next window, and a new convolution can be computed.

This process optimizes the loading from IFMAP by inserting *cutoff paths* in the shift registers. This is possible by using a memory port that is idle during the long arithmetical

operations. This process reduces the memory access by keeping some values and loading only the stride to the input buffer. In the case of the Alexnet layer one, this reduction represents 36.3% less memory reads in subsequent windows (44 instead of the 121 reads).

5.1.2 Convolutional Tree

Most of the literature in Deep Learning accelerators adopts systolic arrays to implement convolutions. Works [Chen et al., 2016] and [Jouppi et al., 2017] employ a set of multiply-accumulate (MAC) operators disposed in a grid-like structure. The approach has the advantage of a high level of parallelism. However, it requires a considerable amount of resources. First, each node in the grid consists in a pair of multiplier and accumulator. Second, the grid requires a network to interconnect all MACs to pass the output values to other neighboring MACs. Last, besides storing the feature maps and filters, there is the need for a large number of input buffers spread around the grid to input the feature maps and filters.

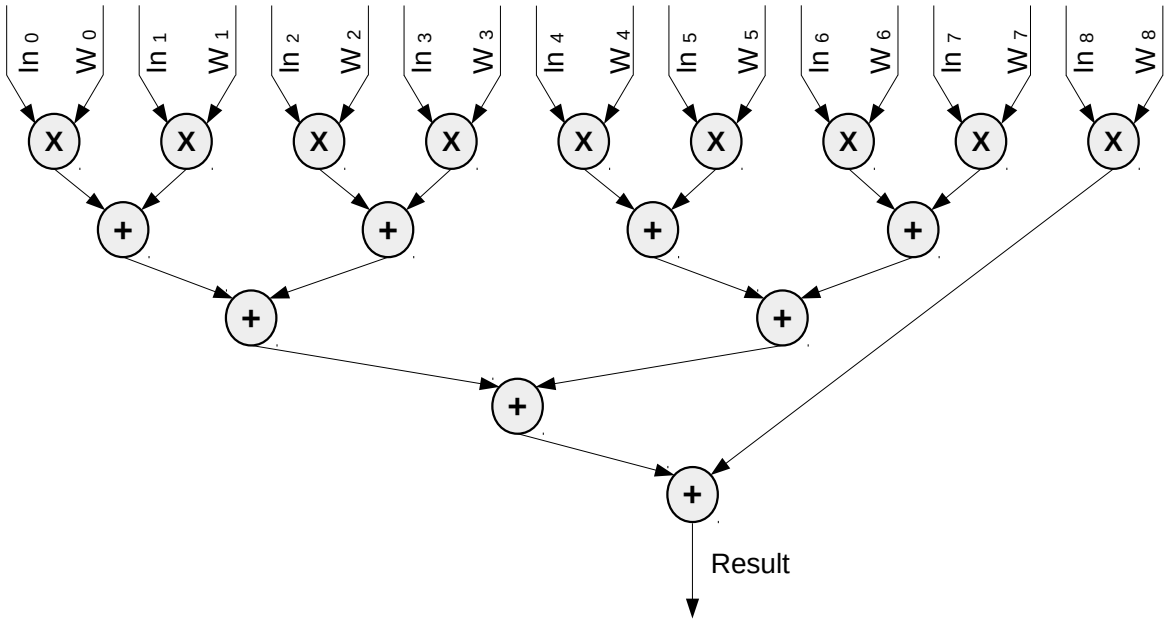

Figure 5.5 – Example of a 3x3 multiply-add tree.

An architectural option to systolic arrays is the multiply-add trees – Figure 5.5 and **D** in Figure 5.3. This approach uses a line of multipliers and an adder tree [Zhang et al., 2015]. The inputs of the multiply-add tree are the *current buffer* and the weight array. Next, the result produced by each pair of multipliers are added in the next level. The result of the adders in the first level travels up the tree, until the root, which produces the final result. The module implemented in this work is capable of implementing convolutions of any size. The size of the adder tree is configured through VHDL generic parameters.

The multiply-add tree is pipelined. The multiplication takes 3 clock cycles and each addition 5 clock cycles. Thus, for layer one there are 121 multipliers in parallel (plus 7 dummy

modules to obtain a full binary tree), and 8 adder levels ($log_2 n + 1$). The time to compute one 11x11 convolution in the current implementation is 43 clock cycles.

5.2 Max-Pool Layer

Simpler than the Convolutional Layer, the Max-pool Layer makes use of a single FSM that controls a shift register input buffer and the max-pool tree. A process generates the address of each window loading the input buffer. When the window is ready (buffer full), the FSM starts the max-pool tree. After the maximum value in the window is found, it is written to the OFMAP memory.

To illustrate a 3x3 max-pool tree, Figure 5.6 shows that the input values are converted to a fixed-point format (as presented in Section 4.3.3). Then, the tree of comparators evaluates all values. It is important to note that the fixed-point numbers are not re-converted to floating-point. Instead, they only serve to perform the comparisons that will decode the output value. Hence, no precision is lost in the conversion from and to fixed-point, and the comparator can be simplified to work with integer operands.

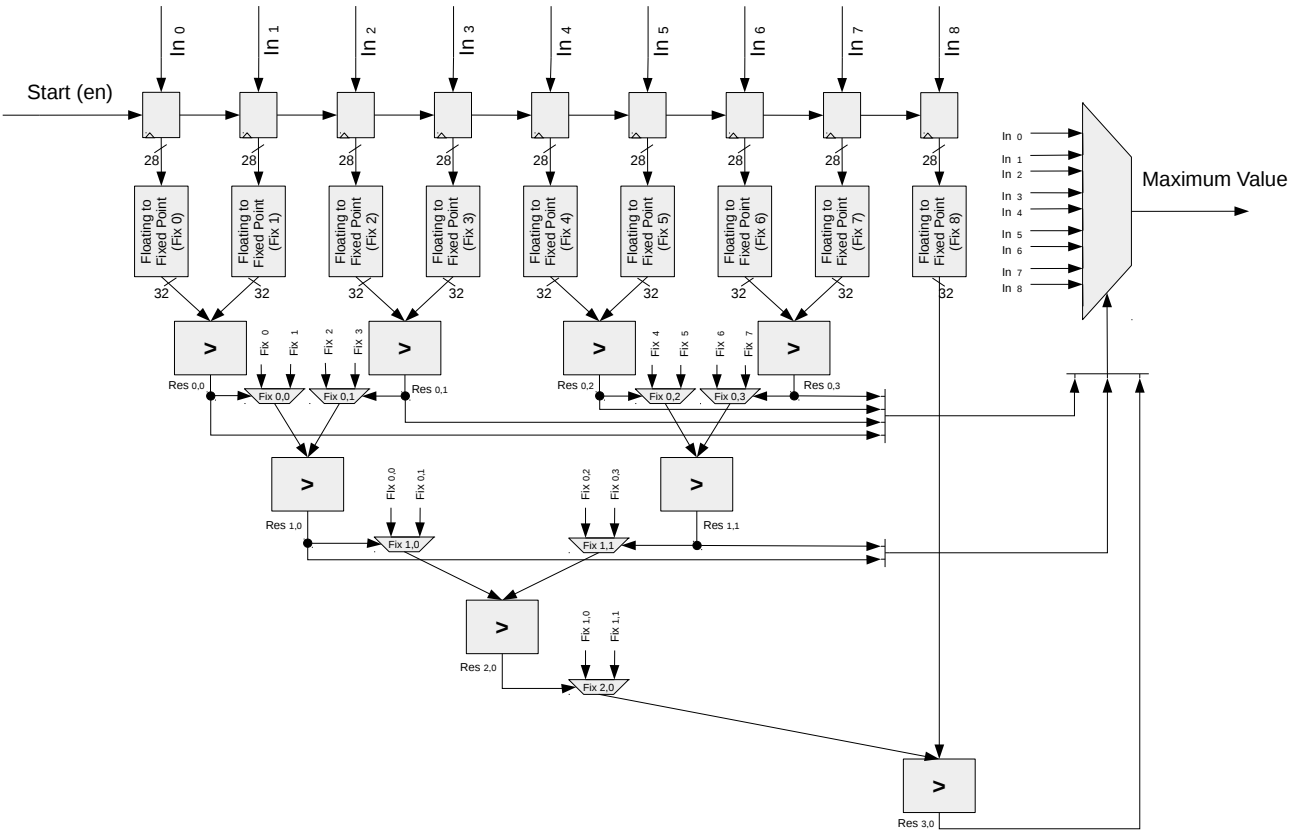

Figure 5.6 – The max-pool tree.

5.3 Multilayer

Following the first two Alexnet layers, the Multilayer module implements the Alexnet layers three, four, and five. As shown in the top view (Figure 5.1), the Multilayer feedbacks itself creating a loop that could implement any number of subsequent convolutional layers of the same filter size. Even that the Multilayer could be used to implement all five convolutional layers on Alexnet, the architecture dedicates two Layer modules for the first two convolutional layers, which have filter sizes of 11x11 and 5x5, while the remaining layers, executed by the multilayer, have filters of size 3x3. The hardware implementation as it is, depends on the off-line configuration to instantiate the convolutional tree. An improvement could be made to re-configure at run-time the convolutional tree, so it would be possible to execute convolutions of different filter sizes. Another important aspect of the Multilayer module is that it saves FPGA resources since only one module is capable of performing multiple CNN layers.

The Multilayer encapsulates a convolutional module and the ping-pong buffer that interfaces with a max-pool or a module that, simply, writes the data from the ping-pong buffer to the OFMAP (when no max-pool is performed such as in the Alexnet layers three and four). Additionally, there is a control FSM that coordinates the two multiplexers seen in top view. To keep the diagrams in Figures 5.1 and 5.7 simplified, they do not display the control connections from the FSM in the Multilayer. However, it is important to mention its role in the operation of the Multilayer module.

In the case of the Alexnet CNN, during the execution of layer two, the multiplexers are first controlled to pass the OFMAP from layer number two to the memory *E* (Figure 5.1). Next, when Multilayer starts to perform the computations of layer three, the FSM sets the multiplexers in a way that the Multilayer reads the IFMAP from memory *E* and writes the OFMAP to memory *G*. As soon as layer three is done, the FSM re-configure both multiplexers, so the Multilayer starts to perform the convolutional layer four, reading the IFMAP from memory *G* and writing the resulting OFMAP to memory *E*. Finally, layer five can start its execution. The FSM, then, returns to the initial configuration, where the IFMAP comes from memory *E* as the OFMAP goes to memory *G*. Similarly, the FSM controls from which memory (*F* or *I*) the Multilayer reads the set of filter weights.

5.4 Fully-Connected Layer Exploration

Unfortunately, due to the limited time to develop this project, the fully-connected layers were not implemented. Some exploration was done to access the viability of embedding these layers into an FPGA. As demonstrated in Section 6, the convolutional layers are consuming most of the available FPGA already. Hence, it is necessary to explore techniques

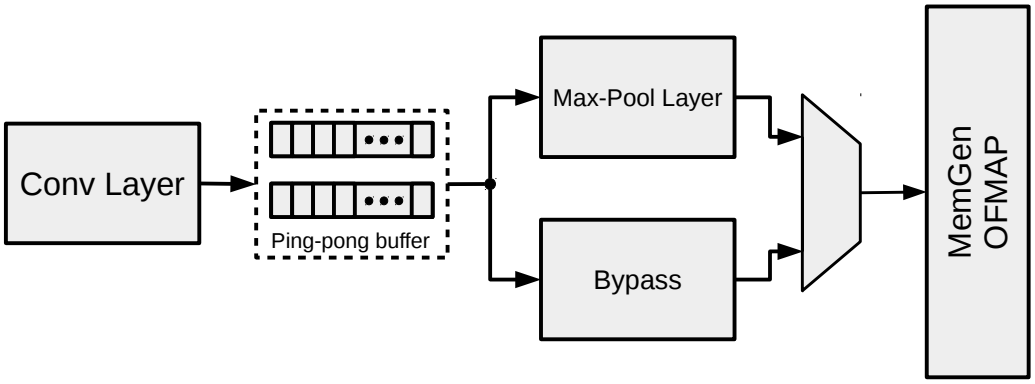

Figure 5.7 – The Multilayer module.

to decrease the amount of memory needed to store the fully-connected weights. Then, enabling the instantiation of all layers in the FPGA. An initial approach was taken to reduce the memory requirements, which involved the discretization of weights. A clustering algorithm was used to discretize the weight values. However, the error observed via software executions made it clear that a more sophisticated approach to the problem is required.

6. RESULTS

This Chapter presents the results related to the proposed architecture. First, it is given an overall performance analysis of the architecture. Next, the FPGA resource utilization is provided. Then, a section details the validation process. Finally, some limitations found during this work are exposed as well as possible solutions.

6.1 Performance Analysis

This section first estimates the CNN performance according to the hardware architecture. The second subsection presents results related to the VHDL simulation.

6.1.1 Estimated Performance

Before presenting RTL simulation results, the time consumed by the convolutional layers can be extracted analytically. The total time can be divided into three parts, the time consumed by arithmetical operations, by loading the current buffer, and by loading the stride buffer between adjacent windows.

First, there is the time consumed by arithmetical operations, i.e., the convolutional tree, is given in Equation 6.1.

$$t_{arith} = (H_{out} * W_{out} * C_{out} * C_{in}) * t_{tree} \tag{6.1}$$

where: H_{out}, W_{out}, and C_{out} are the OFMAP height, width and depth (number of channels) respectively; C_{in} the IFMAP depth, and t_{tree} is the convolutional tree delay observed in the simulation. As the tree varies in size according to the convolutional filter sizes, the time taken by the tree to complete the convolution is particular to each layer. Alexnet has filters of sizes 11x11, 5x5, and 3x3 (Table 3.1). For these configurations, t_{arith} corresponds to 43, 35, and 30 clock cycles.

Second, there is the time taken loading the current buffer (Figure 5.4). When it is not possible to use the stride buffer for fast access (windows not adjacent in the IFMAP), the current buffer needs to be completely loaded before starting the convolutional tree. It occurs in every new line of the IFMAP. In other words, when a filter window strides end, it is not possible to use the stride buffer to load the current buffer. Equation 6.2 presents the time consumed by the multiple loads of the current buffer for a given layer.

$$t_{inBuffer} = (H_{in} * C_{in} * C_{out}) * t_{loadIn} \tag{6.2}$$

where: H_{in} and C_{in} are the IFMAP height and depth, C_{out} is the OFMAP depth, and t_{loadIn} is the time taken by loading the input buffer. The buffer size changes from layer to layer. For layers one and two, $t_{inBuffer}$ is 121 and 26 cycles, respectively. Meanwhile, for layers three, four, and five (3x3 filters) the time for loading the input buffer is 10 cycles.

Finally, for adjacent windows, there is no need of loading the current buffer completely. Hence, only the time for transferring the contents in the stride buffer to the current buffer is accounted for. Equation 6.3 gives the time for the stride buffer.

$$t_{strideBuffer} = (H_{out} * W_{out} - H_{in}) * C_{in} * C_{out} * t_{transfer} \tag{6.3}$$

where: H_{out}, W_{out}, and C_{out} are the OFMAP height, width and depth, respectively. H_{in} and C_{in} are the IFMAP height and depth, as $t_{transfer}$ gives the time consumed in transferring the values from stride to input buffer.

Therefore, the estimated time for a given layer is obtained by adding up the three components:

$$t_{conv} = t_{arith} + t_{inBuffer} + t_{strideBuffer} \tag{6.4}$$

Other operations consume clock cycles during a convolution, like loading the stride buffer, accumulating the results from input channels of the same filter, and adding bias. However, all the operations not exploited in the calculations of t_{conv} happen in parallel with one or more of the three operations presented above. To estimate the number of cycles taken by each convolutional layer in the Alexnet (Table 6.1), Equations 6.1, 6.2, 6.3 were used.

Table 6.1 – Estimated number of clock cycles by layer.

Layer	t_{arith}	$t_{inBuffer}$	$t_{strideBuffer}$	t_{conv}
1	37,461,600	7,910,496	3,223,296	**48,595,392**
2	627,056,640	19,808,256	99,041,280	**745,906,176**
3	498,401,280	14,745,600	88,473,600	**601,620,480**
4	747,601,920	11,059,200	132,710,400	**891,371,520**
5	498,401,280	14,745,600	88,473,600	**601,620,480**

The estimated number of clock cycles to execute the first five layers is 2,889,114,048, corresponding to **5.78** seconds at 500MHz. *Observe that largest parcel in the execution time is due to the arithmetic operations.*

6.1.2 Simulated Performance

The hardware description was simulated to assess the overall performance (Table 6.2). Assuming a frequency of 500 MHz, the architecture executes all Alexnet convolutional layers in approximately **5.67** seconds (this small difference from the estimated time comes from optimizations made in the hardware implementation). In contrast, the ARM Cortex-A5 processor (an ARMv7 architecture) when simulated with the OVP tool takes **1,118.66** seconds to run the five Alexnet convolutional layers.

Results displayed in Table 6.2 are approximated because a five seconds simulation is unfeasible to run with a discrete event simulator. The approximations were based on simulations of the Alexnet with the convolutions having a smaller number of channel (the simulation of the first layer used six channels instead of 96 for example). The time taken by each layer is extrapolated to the nominal parameters.

Table 6.2 – Number of clock cycles for a forward execution (convolutional layers only).

	Operation	**Cycles**
1:	Loading Input image	154,587
2:	Layer 1	47,136,384
3:	Channel 96 Max-pool (Layer 1)	8,028
4:	Layer 2	730,619,904
5:	Channel 256 Max-pool (Layer 2)	1,868
6:	Layer 3	588,054,528
7:	Layer 4	881.639.424
8:	Layer 5	587.759.616
9:	Channel 256 Max-pool (Layer 5)	395
	Total:	**2,835,374,734**

Additionally, a usual metric for deep learning dedicated hardware is Floating-Point Operations per Second (FLOPS). Considering the results above mentioned (5.67 seconds at an operating frequency of 500 MHz) and the total of floating-point operations required by the Alexnet layers one to five ($2.15 * 10^9$), the implemented architecture achieves **380 MFLOPS** executing the Alexnet CNN.

6.2 Resource Utilization

The design was synthesized using the Vivado tool. Table 6.3 gives the resource utilization for the complete design. Note that the critical component is the memory. The architecture, which is not implementing the fully-connected layers, is already requiring 96.67% of the available BRAMs. However, the DSP blocks consumption is less than 5%. The three modules are using only 155 DSP blocks, 121 of which are in layer one convolution, 25 of

which in layer two, and the remaining 9 are used by the Multilayer (Figure 6.1). The utilization of DSP blocks points out that there is room for future improvements concerning more parallelization (multiple convolutional trees executing in one layer).

Table 6.3 – Resource utilization for the complete design.

Resource	Used	Available	Utilization (%)
Clock	1	32	3.13
Slice LUTs	103848	433200	23.97
Slice Registers	75902	866400	8.76
BRAM	1421	1470	96.67
DSPs	155	3600	4.31

The BRAM and DSP utilization of each module are given in Figure 6.1. As explained in Section 4.1, a parameter defines the number of memory blocks (set of 29 BRAMs) when instantiating each memory module. The number of blocks is based on the Alexnet parameters (Table 3.1). It is important to mention that the target FPGA does not offer enough BRAMs to accommodate all Alexnet weights. Hence, the weights of layers three, four, and five need to be loaded from the external memory at run-time. Unfortunately, this feature could not be implemented and is left as future work.

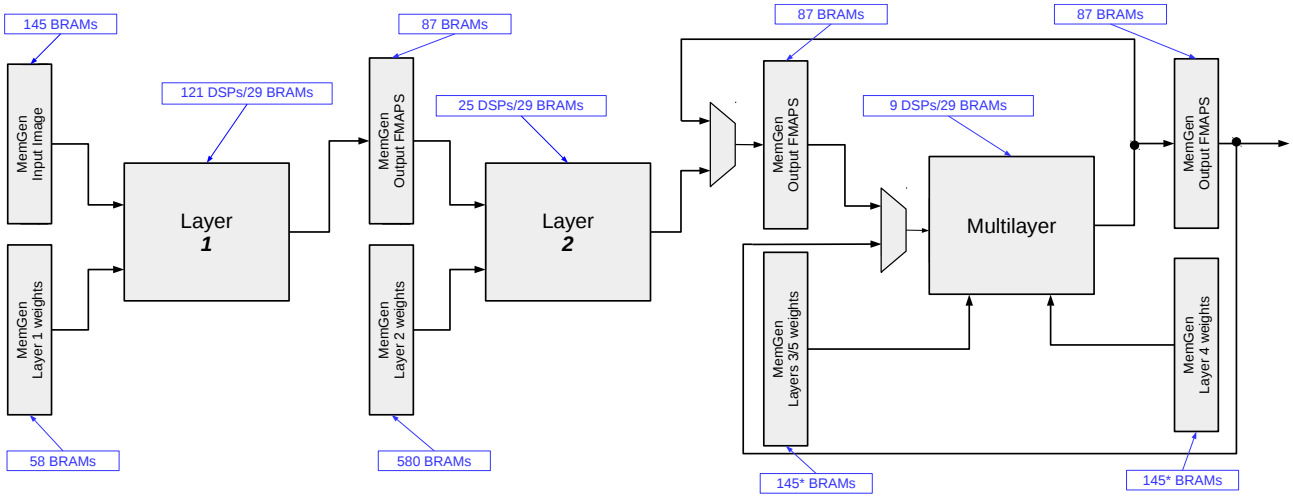

Figure 6.1 – Top view with resource utilization.

6.3 Accuracy of the Hardware Implementation

As pointed out in Section 3.2.2, the software implementation was used to generate the feature maps that were compared against dump files from the hardware simulation. This method enabled to validate the implementation incrementally. To compare both implementations, first, the dump from the hardware simulation is processed, converting the 29-bit sequences of floating-point numbers into their decimal representations. Then, it is

selected in the C source code at which point in the CNN forward execution data is written to a file. Table 6.4 presents the comparison of the first 20 positions of the layer one max-pool OFMAP produced by software and hardware. A small error between the software and the hardware implementation is observed. It comes from the numeric representation (C: float single-precision, hardware: Flopoco 29-bit floating-point format). Such validations were used across all layers during development, enabling to validate the hardware implementation.

Table 6.4 – *Diff* between software and hardware implementation for part of the Alexnet layer one max-pool OFMAP.

Software	**Hardware**	**Relative Error (%)**
44.91939	44.91931	0.000178097
21.23786	21.23785	4.70857E-05
8.752445	8.752502	0.000651242
19.13532	19.13525	0.000365817
19.13532	19.13525	0.000365817
10.94287	10.94293	0.000548299
13.93836	13.93839	0.000215233
18.45827	18.45837	0.00054176
18.45827	18.45837	0.00054176
14.70498	14.70511	0.000884046
14.70498	14.70511	0.000884046
8.460338	8.46048	0.001678392
10.93966	10.93976	0.000914097
18.31306	18.31305	5.46059E-05
12.3388	12.33887	0.000567313
12.3388	12.33887	0.000567313
27.41918	27.41919	3.64708E-05
22.66814	22.66809	0.000220574
12.72069	12.7207	7.8612E-05
28.2347	28.23456	0.000495846
19.10462	19.10461	5.23434E-05

6.4 Final Remarks

The development of this first CNN implementation brings important lessons:

Positive outcomes:

a) the arithmetic operations are not the bottleneck, and it is possible to explore a massive parallelism in the convolutional trees;

b) FPGAs offer enough memory resources to implement the feature maps;

c) the multilayer approach enables the implementation of other CNNs, with a larger number of hidden layers;

d) the spatial parallelism to execute the tasks (e.g., convolutional trees and max-pool) speed-up the performance of the implementation.

Negative outcomes:

a) the weight parameters must be stored in external memories;

b) the fully connected layer is challenging due to the large number of weights;

c) the performance in a PC is a result that requires further study, since optimizations made by compilers can also be used in hardware.

7. CONCLUSIONS AND FUTURE WORK

The work described in this document produced a parametrizable architecture capable of executing multiple CNN convolutional layers. The case study CNN was Alexnet. Nonetheless, it is important to mention that the hardware can be configured to other Convolutional Neural Networks by modifying the set of parameters. The implementation of the first two layers showed that a set of layers could be pipelined while there are available resources on the target FPGA. On the other hand, the multilayer module enables the designer to implement an arbitrary number of convolutional layers with a limited amount of resources.

This work consisted in a set of steps, starting from the study of Convolutional Neural Networks to the hardware validation. Objectively, the following goals were achieved:

- Understanding the design flow of an FPGA project;
- Dominium of the involved algorithms in the inference of Convolutional Neural Networks;
- Implementation of a baremetal software version of the Alexnet CNN;
- Performance analysis of the Alexnet on a virtual ARM platform;
- Implementation of several parametrizable RTL modules, including:
  - Interleaved buffers
  - Convolutional tree
  - Max-pool tree
  - Address generators
  - Generic memory

The main contribution of this work is creating an environment that can be used for further exploration, which may concern arithmetical operators or numeric representations. A next step to improve the architecture performance is the parallelization of the convolutional trees. It is possible to keep a single input buffer feeding the parallel trees if multiple filters are divided into multiple memories of smaller sizes. Thus, the convolutional layer would perform convolutions from different filters on the same input concurrently. Another future work is the implementation of the fully-connected layers. Moreover, as the amount of memory used by these layers and the number of necessary multiplications form an architectural challenge, a study of techniques that can alleviate the high requirements in the bandwidth of fully-connected layers could take place to initiate its implementation.

REFERENCES

[Abadi et al., 2016] Abadi, M. et al. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*, abs/1603.04467:19.

[Bailey, 2007] Bailey, D. G. (2007). *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. Springer.

[Bailey, 2011] Bailey, D. G. (2011). *Design for embedded image processing on FPGAs*. Wiley-IEEE Press.

[Caulfield et al., 2016] Caulfield, A. M. et al. (2016). A cloud-scale acceleration architecture. In *IEEE/ACM International Symposium on Microarchitecture, MICRO*, pages 7:1–7:13.

[Ceva, 2015] Ceva (2015). Intelligent Vision Processor. https://www.ceva-dsp.com/product/ceva-xm4/.

[Chen et al., 2016] Chen, Y., Emer, J. S., and Sze, V. (2016). Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *International Symposium on Computer Architecture, ISCA*, pages 367–379.

[Collobert et al., 2002] Collobert, R., Bengio, S., and Marithoz, J. (2002). Torch: A modular machine learning software library. Technical report, Idiap Research Institute. http://publications.idiap.ch/downloads/reports/2002/rr02-46.pdf.

[Courbariaux et al., 2015] Courbariaux, M., Bengio, Y., and David, J.-P. (2015). Training deep neural networks with low precision multiplications. In *International Conference on Learning Representations, ICLR*, pages 1–10.

[de Dinechin and Pasca, 2011] de Dinechin, F. and Pasca, B. (2011). Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28:18–27.

[Gabor Kereszturi et al., 2018] Gabor Kereszturi, L. N. S. et al. (2018). Integrating airborne hyperspectral imagery and lidar for volcano mapping and monitoring through image classification. *Int. J. Applied Earth Observation and Geoinformation*, 73:323–339.

[Girshick et al., 2014] Girshick, R. B., Donahue, J., Darrell, T., and Malik, J. (2014). Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 580–587.

[Gupta et al., 2015] Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep Learning with Limited Numerical Precision. In *International Conference on Machine Learning, ICML*, pages 1737–1746.

[Hailesellasie et al., 2018] Hailesellasie, M., Hasan, S. R., Khalid, F., Wad, F. A., and Shafique, M. (2018). FPGA-Based Convolutional Neural Network Architecture with Reduced Parameter Requirements. In *IEEE International Symposium on Circuits and Systems, ISCAS*, pages 1–5.

[Haykin and Haykin, 2009] Haykin, S. and Haykin, S. S. (2009). *Neural Networks and Learning Machines*. Prentice Hall.

[He et al., 2016] He, K., Zhang, X., et al. (2016). Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 770–778.

[Hershey et al., 2017] Hershey, S. et al. (2017). CNN architectures for large-scale audio classification. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP*, pages 131–135.

[Ian Goodfellow and Courville, 2016] Ian Goodfellow, Y. B. and Courville, A. (2016). *Deep Learning*. MIT Press.

[IEEE, 2008] IEEE (2008). IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70.

[Imperas, 2018] Imperas (2018). Open Virtual Platforms. http://www.ovpworld.org/.

[Intel Corp., 2015] Intel Corp. (2015). Intel Movidius Neural Compute Stick. https://www.movidius.com/.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning, ICML*, pages 448–456.

[Jia et al., 2014] Jia, Y. et al. (2014). Caffe: Convolutional Architecture for Fast Feature Embedding. In *International Conference on Machine Learning, ICML*, pages 675–678.

[Jiao et al., 2017] Jiao, L., Luo, C., Cao, W., Zhou, X., and Wang, L. (2017). Accelerating low bit-width convolutional neural networks with embedded FPGA. In *International Conference on Field Programmable Logic and Applications, FPL*, pages 1–4.

[Jouppi et al., 2017] Jouppi, N. P. et al. (2017). In-Datacenter Performance Analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture, ISCA*, pages 1–12.

[Karpathy et al., 2014] Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., and Li, F. (2014). Large-scale video classification with convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 1725–1732.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *Conference on Neural Information Processing Systems, NIPS*, pages 1106–1114.

[Kunihiko Fukushima and Ito, 1983] Kunihiko Fukushima, S. M. and Ito, T. (1983). Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Trans. Systems, Man, and Cybernetics*, 13:826–834.

[Kuon and Rose, 2007] Kuon, I. and Rose, J. (2007). Measuring the gap between fpgas and asics. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(2):203–215.

[Lawrence et al., 1997] Lawrence, S., Giles, C. L., Tsoi, A. C., and Back, A. D. (1997). Face recognition: a convolutional neural-network approach. *IEEE Trans. Neural Networks*, 8(1):98–113.

[LeCun and Bengio, 1998] LeCun, Y. and Bengio, Y. (1998). *The Handbook of Brain Theory and Neural Networks*. MIT Press.

[Li et al., 2016] Li, H., Fan, X., Jiao, L., Cao, W., Zhou, X., and Wang, L. (2016). A high performance fpga-based accelerator for large-scale convolutional neural networks. In *International Conference on Field Programmable Logic and Applications, FPL*, pages 1–9.

[Mitchell, 1997] Mitchell, T. (1997). *Machine Learning*. McGraw-Hill, Inc., 1 edition.

[Muller et al., 2010] Muller, J., Brisebarre, N., et al. (2010). *Handbook of Floating-Point Arithmetic*. Birkhäuser.

[Nair and Hinton, 2010] Nair, V. and Hinton, G. (2010). Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning, ICML*, pages 807–814.

[Nethercote and Seward, 2007] Nethercote, N. and Seward, J. (2007). Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM Sigplan Notices*, 42(6):89–100.

[Nomi, 2018] Nomi, O. (2018). tiny-dnn Documentation - Release 1.0.0a1. Technical report, Tiny-Dnn. https://media.readthedocs.org/pdf/tiny-dnn/latest/tiny-dnn.pdf.

[NovuMind, 2018] NovuMind (2018). NovuMind Showcases the Most Power-Efficient Chip at CES 2018. https://www.prnewswire.com/news-releases/novumind-showcases-the-most-power-efficient-chip-at-ces-2018-300580611.html.

[Nurvitadhi et al., 2017] Nurvitadhi, E. et al. (2017). Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks? In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA*, pages 5–14.

[Nvidia, 2018] Nvidia (2018). NVIDIA TITAN V. https://www.nvidia.com/en-us/titan/titan-v/.

[Ovtcharov et al., 2015] Ovtcharov, K., Ruwase, O., Kim, J.-Y., Fowers, J., Strauss, K., and Chung, E. (2015). Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. Technical report, Microsoft Research. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/CNN20Whitepaper.pdf.

[Pinheiro and Collobert, 2014] Pinheiro, P. H. O. and Collobert, R. (2014). Recurrent convolutional neural networks for scene labeling. In *International Conference on Machine Learning, ICML*, pages 82–90.

[Putnam et al., 2016] Putnam, A. et al. (2016). A reconfigurable fabric for accelerating large-scale datacenter services. *Commun. ACM*, 59:114–122.

[Qing Li et al., 2014] Qing Li, W. C. et al. (2014). Medical image classification with convolutional neural network. In *International Conference on Control Automation Robotics & Vision, ICARCV*, pages 844–848.

[Russakovsky et al., 2015] Russakovsky, O. et al. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252.

[Simonyan and Zisserman, 2014a] Simonyan, K. and Zisserman, A. (2014a). Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556:1–14.

[Simonyan and Zisserman, 2014b] Simonyan, K. and Zisserman, A. (2014b). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

[Stephen D. Brown et al., 1992] Stephen D. Brown, R. J. F. et al. (1992). *Field-Programmable Gate Arrays*. Springer.

[Stokel-Walker, 2018] Stokel-Walker, C. (2018). Move over CPUs and GPUs, the Intelligence Processing Unit is the super-smart chip of the future. https://www.wired.co.uk/article/graphcore-ai-ipu-chip-nigel-toon.

[Sze et al., 2018] Sze, V., Chen, Y., et al. (2018). Hardware for machine learning: Challenges and opportunities. In *IEEE Custom Integrated Circuits Conference, CICC*, pages 1–8.

[Sze et al., 2017] Sze, V., Chen, Y., Yang, T., and Emer, J. S. (2017). Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329.

[Szegedy et al., 2015] Szegedy, C., Liu, W., et al. (2015). Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 1–9.

[Venkataramani et al., 2017] Venkataramani, S., Ranjan, A., et al. (2017). Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *International Symposium on Computer Architecture, ISCA*, pages 13–26.

[Wang et al., 2018] Wang, J., Lin, J., and Wang, Z. (2018). Efficient Hardware Architectures for Deep Convolutional Neural Network. *IEEE Trans. on Circuits and Systems*, 65-I(6):1941–1953.

[Xilinx Inc, 2016a] Xilinx Inc (2016a). 7 Series FPGAs Configurable Logic Block. https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[Xilinx Inc, 2016b] Xilinx Inc (2016b). 7 Series FPGAs Memory Resources. https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[Xilinx Inc, 2016c] Xilinx Inc (2016c). 7 Series FPGAs Overview. https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[Xilinx Inc, 2018] Xilinx Inc (2018). 7 Series DSP48E1 Slice. https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf.

[Yann LeCun et al., 1989] Yann LeCun, B. B. et al. (1989). Handwritten Digit Recognition with a Back-Propagation Network. In *Conference on Neural Information Processing Systems, NIPS*, pages 396–404.

[Yoshihisa Hara et al., 1994] Yoshihisa Hara, R. G. A. et al. (1994). Application of neural networks to radar image classification. *IEEE Trans. Geoscience and Remote Sensing*, 32:100–109.

[Zeiler and Fergus, 2014] Zeiler, M. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European Conference on Computer Vision ECCV*, pages 818–833.

[Zhang et al., 2015] Zhang, C., Li, P., et al. (2015). Optimizing fpga-based accelerator design for deep convolutional neural networks. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA*, pages 161–170.

[Zhu et al., 2018] Zhu, Y., Mattina, M., and Whatmough, P. N. (2018). Mobile Machine Learning Hardware at ARM: A Systems-on-Chip (SoC) Perspective. *Computing Research Repository*.

APPENDIX A – ALEXNET C SOURCE CODE

This appendix shows the source code in C that was used to carried out simulations on the ARM platform as well as provided means for the hardware verification.

Listing A.1 – The Alexnet C implementation

```
// Alexnet CNN

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "param_headers/1_in.h"
#include "param_headers/1_weight.h"
#include "param_headers/1_bias.h"

#include "param_headers/2_weight.h"
#include "param_headers/2_bias.h"

#include "param_headers/3_weight.h"
#include "param_headers/3_bias.h"

#include "param_headers/4_weight.h"
#include "param_headers/4_bias.h"

#include "param_headers/5_weight.h"
#include "param_headers/5_bias.h"

#include "fc_include.h"

/******* Layer Conv 1 *******/
#define FILTER_HEIGHT_1      11
#define FILTER_WIDTH_1       11
#define IN_HEIGHT_1          227
#define IN_WIDTH_1           227
#define IN_DEPTH_1           3
#define OUT_CONV_HEIGHT_1    55
#define OUT_CONV_WIDTH_1     55
#define OUT_HEIGHT_1         27
#define OUT_WIDTH_1          27
#define OUT_DEPTH_1          96
#define STRIDE_CONV_1        4
```

```
#define STRIDE_MAX_1          2
#define POOL_SIZE_1           3
/* ***************************** */

/* ******* Layer Conv 2 ******** */
#define PAD_IN_2              2
#define FILTER_HEIGHT_2       5
#define FILTER_WIDTH_2        5
#define IN_HEIGHT_2           31
#define IN_WIDTH_2            31
#define IN_DEPTH_2            96
#define OUT_CONV_HEIGHT_2     27
#define OUT_CONV_WIDTH_2      27
#define OUT_HEIGHT_2          13
#define OUT_WIDTH_2           13
#define OUT_DEPTH_2           256
#define STRIDE_CONV_2         1
#define STRIDE_MAX_2          2
#define POOL_SIZE_2           3
/* ***************************** */

/* ******* Layer Conv 3 ******** */
#define PAD_IN_3              1
#define FILTER_HEIGHT_3       3
#define FILTER_WIDTH_3        3
#define IN_HEIGHT_3           15
#define IN_WIDTH_3            15
#define IN_DEPTH_3            256
#define OUT_HEIGHT_3          13
#define OUT_WIDTH_3           13
#define OUT_DEPTH_3           384
#define STRIDE_CONV_3         1
/* ***************************** */

/* ******* Layer Conv 4 ******** */
#define PAD_IN_4              1
#define FILTER_HEIGHT_4       3
#define FILTER_WIDTH_4        3
#define IN_HEIGHT_4           15
#define IN_WIDTH_4            15
#define IN_DEPTH_4            384
#define OUT_HEIGHT_4          13
```

```
#define OUT_WIDTH_4       13
#define OUT_DEPTH_4       384
#define STRIDE_CONV_4     1
/* ************************** */

/* ******* Layer Conv 5 ******** */
#define PAD_IN_5              1
#define FILTER_HEIGHT_5       3
#define FILTER_WIDTH_5        3
#define IN_HEIGHT_5           15
#define IN_WIDTH_5            15
#define IN_DEPTH_5            384
#define OUT_CONV_HEIGHT_5     13
#define OUT_CONV_WIDTH_5      13
#define OUT_HEIGHT_5          6
#define OUT_WIDTH_5           6
#define OUT_DEPTH_5           256
#define STRIDE_MAX_5          2
#define STRIDE_CONV_5         1
#define POOL_SIZE_5           3
/* ************************** */

/* ******* Layer FC 6 ******** */
#define IN_HEIGHT_6   6
#define IN_WIDTH_6    6
#define IN_DEPTH_6    256
#define OUT_HEIGHT_6  4096
#define OUT_WIDTH_6   1
#define OUT_DEPTH_6   1
/* ************************** */

/* ******* Layer FC 7 ******** */
#define IN_HEIGHT_7   4096
#define IN_WIDTH_7    1
#define IN_DEPTH_7    1
#define OUT_HEIGHT_7  4096
#define OUT_WIDTH_7   1
#define OUT_DEPTH_7   1
/* ************************** */

/* ******* Layer FC 8 ******** */
#define IN_HEIGHT_8   4096
```

```
#define IN_WIDTH_8    1
#define IN_DEPTH_8    1
#define OUT_HEIGHT_8 1000
#define OUT_WIDTH_8   1
#define OUT_DEPTH_8   1
/*****************************/

void conv(const float in[], const float weights[], const float bias[], float out[]
  const unsigned kh, const unsigned kw, const unsigned ih, const unsigned iw,
  const unsigned id, const unsigned oh, const unsigned ow, const unsigned od,
  const unsigned s, char layer){

  unsigned o_s;
  unsigned i_s;
  unsigned o_x;
  unsigned o_y;
  unsigned f_x;
  unsigned f_y;

  float sum;
  float * in_el;
  float * f_el;
  float * in_slice;
  float * filter_slice;
  float * in_line;
  float * o_line;
  float * out_slice;

  unsigned ls = s * ih;

  for ( o_s = 0; o_s < od; o_s++) {
    out_slice = &out[ o_s * oh * ow ];
    for ( i_s = 0; i_s < id ; i_s++) {
      filter_slice  = &weights[ (o_s * kh * kw * id) + (i_s * kh * kw) ];
      in_slice = &in[ i_s * ih * iw ];
      o_line = out_slice;
      for ( o_y = 0; o_y < oh; o_y++) {
        in_line = in_slice;
        for ( o_x = 0; o_x < ow; o_x++) {
          in_el = in_line;
          f_el = filter_slice;
          sum = 0.0;
```

```
          for ( f_y = 0; f_y < kh; f_y++) {
            for ( f_x = 0; f_x < kw; f_x++) {
              sum += f_el[f_x] * in_el[f_x];
            }
            f_el += kw;
            in_el += iw;
          }
          o_line[o_x] += sum;
          in_line += s;
        }
        o_line += ow;
        in_slice += ls;
      }
    }

    for(o_x = 0; o_x < oh*ow; o_x++) {
      out_slice[o_x] += bias[o_s];
    }
  }
}

void relu(float in[], const unsigned ih, const unsigned iw, const unsigned id) {
  unsigned size = ih * iw * id;
  unsigned i;

  for( i=0; i < size; ++i) {
    if(in[i] < 0.0){
      in[i] = 0.0;
    }
  }
}

void softmax(float in[], float out [],
  const unsigned ih, const unsigned iw, const unsigned id) {

  unsigned size = ih * iw * id;
  unsigned i;
  float denominator = 0.0;

  float alpha = in[0];
  for( i=1; i < size; ++i) {
    if(in[i] > alpha)
```

```
      alpha = in[i];
  }

  for( i=0; i < size; ++i) {
    out[i] = exp(in[i] - alpha);
    denominator += out[i];
  }

  for( i=0; i < size; ++i) {
    out[i] /= denominator;
  }
}

void pad(float in[], float out[], const unsigned ih, const unsigned iw,
         const unsigned id, const unsigned pad) {
  unsigned oh = ih + 2*pad;
  unsigned ow = iw + 2*pad;
  unsigned d, x, y, o_idx, i_idx;

  for( d = 0; d < id; ++d ){
    for( y = 0; y < oh; ++y ){
      for( x = 0; x < ow; ++x ){

        o_idx = (oh*d + y) * ow + x;
        i_idx = 0;
        if( y < pad || y > (ih+pad-1) || x < pad || x > (iw+pad-1) ){
          out[o_idx] = 0.0;
        } else {
          i_idx = (ih*d + y - pad) * iw + x - pad;
          out[o_idx] = in[i_idx];
        }

      }
    }
  }

}

void add_square_sum(float in[], float out[], const unsigned size) {
  unsigned i;
  for (i = 0; i < size; i++) out[i] += in[i] * in[i];
}
```

```
void sub_square_sum(float in[], float out[], const unsigned size) {
  unsigned i;
  for (i = 0; i < size; i++) out[i] -= in[i] * in[i];
}

void lrn(float in[], float out[],
  const unsigned ih, const unsigned iw, const unsigned id) {

  unsigned size_ = 5;              // Alexnet defined
  float alpha_ = 0.000100;         // Alexnet defined
  float beta_  = 0.750000;         // Alexnet defined

  unsigned i,j;
  unsigned wxh = ih * iw;
  unsigned head = size_ / 2;
  long tail = (long)head - (long)size_;
  float alpha_div_size = alpha_ / size_;

  float * dst;
  float * src;
  float * in_square_ = malloc( wxh * sizeof(float));

  for (i = 0; i < size_ / 2; i++) {
    add_square_sum(&(in[i*wxh]), in_square_, wxh);
  }

  for ( i = 0; i < id; i++, head++, tail++) {
    if (head < id)
      add_square_sum(&in[head * wxh], in_square_, wxh);

    if (tail >= 0)
      sub_square_sum(&in[tail * wxh], in_square_, wxh);

    dst = &out[i * wxh];
    src = &in[i * wxh];
    for (j = 0; j < wxh; j++)
      dst[j] = src[j] * pow(1.0 + alpha_div_size * in_square_[j], -beta_);
  }

  free(in_square_);
}
```

```
void maxpool(float in[], float out[], const unsigned ih, const unsigned iw,
  const unsigned id, const unsigned oh, const unsigned ow, const unsigned od,
  const unsigned stride, const unsigned ps, char layer) {

  unsigned x_o, y_o, k, wx, wy;
  unsigned in_idx, out_idx;
  float max;

  for( k = 0; k < od; ++k) {
    for( y_o = 0; y_o < oh; y_o++ ) {
      for( x_o = 0; x_o < ow; x_o++ ) {
        max = 0.0;
        for( wy = 0; wy < ps; wy++ ) {
          for( wx = 0; wx < ps; wx++ ) {
            in_idx = (ih*k + (y_o*stride+wy)) * iw + (x_o*stride+wx);
            if(in[in_idx] > max) {
              max = in[in_idx];
            }
          }
        }
        out_idx = (oh*k + y_o) * ow + x_o;
        out[out_idx] = max;
      }
    }
  }

}

void fc(float in[], float weights[], float bias[], float out[],
   const unsigned ih, const unsigned iw, const unsigned id,
   const unsigned oh, const unsigned ow, const unsigned od ){

   unsigned o_y, i_s;
   unsigned in_size = ih*iw*id;

   for ( o_y = 0; o_y < oh; o_y++) {
     out[o_y] = 0.0;
     for ( i_s = 0; i_s < in_size; i_s++) {
       out[o_y] += in[i_s] * weights[i_s * oh + o_y];
     }
     out[o_y] += bias[o_y];
```

```
    }

}

int main(int argc, char** argv) {

  size_t alloc_size;

  unsigned i;

  /******************************* Layer 1 *********************************/
  alloc_size = OUT_CONV_WIDTH_1*OUT_CONV_HEIGHT_1*OUT_DEPTH_1;
  float * out_1 = malloc(alloc_size * sizeof(float));
  if (!out_1) { perror("malloc failed"); exit(EXIT_FAILURE); };
  // Inicializa out_1
  for( i = 0; i < OUT_CONV_WIDTH_1*OUT_CONV_HEIGHT_1*OUT_DEPTH_1; ++i) {
    out_1[i] = 0.0;
  }

  conv(in_1, weight_1, bias_1, out_1,
    FILTER_HEIGHT_1, FILTER_WIDTH_1,
    IN_HEIGHT_1, IN_WIDTH_1, IN_DEPTH_1,
    OUT_CONV_HEIGHT_1, OUT_CONV_WIDTH_1, OUT_DEPTH_1,
    STRIDE_CONV_1, 1);

  relu(out_1, OUT_CONV_HEIGHT_1, OUT_CONV_WIDTH_1, OUT_DEPTH_1);

  alloc_size = OUT_CONV_WIDTH_1*OUT_CONV_HEIGHT_1*OUT_DEPTH_1;
  float * out_lrn_1 = malloc(alloc_size * sizeof(float));
  if (!out_lrn_1) { perror("malloc failed"); exit(EXIT_FAILURE); };

  lrn(out_1, out_lrn_1, OUT_CONV_HEIGHT_1, OUT_CONV_WIDTH_1, OUT_DEPTH_1);

  free(out_1);

  alloc_size = OUT_HEIGHT_1*OUT_WIDTH_1*OUT_DEPTH_1;
  float * out_pool_1 = malloc(alloc_size * sizeof(float));
  if (!out_pool_1) { perror("malloc failed"); exit(EXIT_FAILURE); };

  maxpool(out_lrn_1, out_pool_1, OUT_CONV_HEIGHT_1, OUT_CONV_WIDTH_1, OUT_DEPTH_1,
          OUT_HEIGHT_1, OUT_WIDTH_1, OUT_DEPTH_1,
          STRIDE_MAX_1, POOL_SIZE_1, 1);
```

```
free(out_lrn_1);

/******************************* Layer 2 *********************************/
alloc_size = IN_HEIGHT_2*IN_WIDTH_2*IN_DEPTH_2;
float * in_2 = malloc(alloc_size * sizeof(float));
if (!in_2) { perror("malloc failed"); exit(EXIT_FAILURE); };

pad(out_pool_1, in_2, OUT_HEIGHT_1, OUT_WIDTH_1, OUT_DEPTH_1, PAD_IN_2);

free(out_pool_1);

alloc_size = OUT_CONV_WIDTH_2*OUT_CONV_HEIGHT_2*OUT_DEPTH_2;
float * out_2 = malloc(alloc_size * sizeof(float));
if (!out_2) { perror("malloc failed"); exit(EXIT_FAILURE); };

for( i = 0; i < OUT_CONV_WIDTH_2*OUT_CONV_HEIGHT_2*OUT_DEPTH_2; ++i) {
  out_2[i] = 0.0;
}

conv(in_2, weight_2, bias_2, out_2,
  FILTER_HEIGHT_2, FILTER_WIDTH_2,
  IN_HEIGHT_2, IN_WIDTH_2, IN_DEPTH_2,
  OUT_CONV_HEIGHT_2, OUT_CONV_WIDTH_2, OUT_DEPTH_2,
  STRIDE_CONV_2, 2);

free(in_2);

relu(out_2, OUT_CONV_HEIGHT_2, OUT_CONV_WIDTH_2, OUT_DEPTH_2);

alloc_size = OUT_CONV_WIDTH_2*OUT_CONV_HEIGHT_2*OUT_DEPTH_2;
float * out_lrn_2 = malloc(alloc_size * sizeof(float));
if (!out_lrn_2) { perror("malloc failed"); exit(EXIT_FAILURE); };

lrn(out_2, out_lrn_2, OUT_CONV_HEIGHT_2, OUT_CONV_WIDTH_2, OUT_DEPTH_2);

free(out_2);

alloc_size = OUT_HEIGHT_2*OUT_WIDTH_2*OUT_DEPTH_2;
float * out_pool_2 = malloc(alloc_size * sizeof(float));
if (!out_pool_2) { perror("malloc failed"); exit(EXIT_FAILURE); };
```

```
maxpool(out_lrn_2, out_pool_2, OUT_CONV_HEIGHT_2, OUT_CONV_WIDTH_2, OUT_DEPTH_2,
        OUT_HEIGHT_2, OUT_WIDTH_2, OUT_DEPTH_2,
        STRIDE_MAX_2, POOL_SIZE_2, 2);

free(out_lrn_2);

/******************************** Layer 3 ********************************/
alloc_size = IN_HEIGHT_3*IN_WIDTH_3*IN_DEPTH_3;
float * in_3 = malloc(alloc_size * sizeof(float));
if (!in_3) { perror("malloc failed"); exit(EXIT_FAILURE); };

pad(out_pool_2, in_3, OUT_HEIGHT_2, OUT_WIDTH_2, OUT_DEPTH_2, PAD_IN_3);

free(out_pool_2);

alloc_size = OUT_WIDTH_3*OUT_HEIGHT_3*OUT_DEPTH_3;
float * out_3 = malloc(alloc_size * sizeof(float));
if (!out_3) { perror("malloc failed"); exit(EXIT_FAILURE); };
for( i = 0; i < OUT_WIDTH_3*OUT_HEIGHT_3*OUT_DEPTH_3; ++i) {
  out_3[i] = 0.0;
}

conv(in_3, weight_3, bias_3, out_3,
  FILTER_HEIGHT_3, FILTER_WIDTH_3,
  IN_HEIGHT_3, IN_WIDTH_3, IN_DEPTH_3,
  OUT_HEIGHT_3, OUT_WIDTH_3, OUT_DEPTH_3,
  STRIDE_CONV_3, 3);

free(in_3);

relu(out_3, OUT_HEIGHT_3, OUT_WIDTH_3, OUT_DEPTH_3);

/******************************** Layer 4 ********************************/
alloc_size = IN_HEIGHT_4*IN_WIDTH_4*IN_DEPTH_4;
float * in_4 = malloc(alloc_size * sizeof(float));
if (!in_4) { perror("malloc failed"); exit(EXIT_FAILURE); };

pad(out_3, in_4, OUT_HEIGHT_3, OUT_WIDTH_3, OUT_DEPTH_3, PAD_IN_4);

free(out_3);

alloc_size = OUT_WIDTH_4*OUT_HEIGHT_4*OUT_DEPTH_4;
```

```
float * out_4 = malloc(alloc_size * sizeof(float));
if (!out_4) { perror("malloc failed"); exit(EXIT_FAILURE); };
for( i = 0; i < OUT_WIDTH_4*OUT_HEIGHT_4*OUT_DEPTH_4; ++i) {
  out_4[i] = 0.0;
}

conv(in_4, weight_4, bias_4, out_4,
  FILTER_HEIGHT_4, FILTER_WIDTH_4,
  IN_HEIGHT_4, IN_WIDTH_4, IN_DEPTH_4,
  OUT_HEIGHT_4, OUT_WIDTH_4, OUT_DEPTH_4,
  STRIDE_CONV_4, 4);

free(in_4);

relu(out_4, OUT_HEIGHT_4, OUT_WIDTH_4, OUT_DEPTH_4);

/******************************** Layer 5 ********************************/
alloc_size = IN_HEIGHT_5*IN_WIDTH_5*IN_DEPTH_5;
float * in_5 = malloc(alloc_size * sizeof(float));
if (!in_5) { perror("malloc failed"); exit(EXIT_FAILURE); };

pad(out_4, in_5, OUT_HEIGHT_4, OUT_WIDTH_4, OUT_DEPTH_4, PAD_IN_5);

free(out_4);

alloc_size = OUT_CONV_WIDTH_5*OUT_CONV_HEIGHT_5*OUT_DEPTH_5;
float * out_5 = malloc(alloc_size * sizeof(float));
if (!out_5) { perror("malloc failed"); exit(EXIT_FAILURE); };
for( i = 0; i < OUT_CONV_WIDTH_5*OUT_CONV_HEIGHT_5*OUT_DEPTH_5; ++i) {
  out_5[i] = 0.0;
}

conv(in_5, weight_5, bias_5, out_5,
  FILTER_HEIGHT_5, FILTER_WIDTH_5,
  IN_HEIGHT_5, IN_WIDTH_5, IN_DEPTH_5,
  OUT_CONV_HEIGHT_5, OUT_CONV_WIDTH_5, OUT_DEPTH_5,
  STRIDE_CONV_5, 5);

free(in_5);

relu(out_5, OUT_CONV_HEIGHT_5, OUT_CONV_WIDTH_5, OUT_DEPTH_5);
```

```
alloc_size = OUT_HEIGHT_5*OUT_WIDTH_5*OUT_DEPTH_5;
float * out_pool_5 = malloc(alloc_size * sizeof(float));
if (!out_pool_5) { perror("malloc␣failed"); exit(EXIT_FAILURE); };

maxpool(out_5, out_pool_5, OUT_CONV_HEIGHT_5, OUT_CONV_WIDTH_5, OUT_DEPTH_5,
        OUT_HEIGHT_5, OUT_WIDTH_5, OUT_DEPTH_5,
        STRIDE_MAX_5, POOL_SIZE_5, 5);

free(out_5);

/******************************** Layer 6 *********************************/
alloc_size = OUT_HEIGHT_6*OUT_WIDTH_6*OUT_DEPTH_6;
float * out_6 = malloc(alloc_size * sizeof(float));
if (!out_6) { perror("malloc␣failed"); exit(EXIT_FAILURE); };

fc(out_pool_5, weight_6, bias_6, out_6,
              IN_HEIGHT_6, IN_WIDTH_6, IN_DEPTH_6,
              OUT_HEIGHT_6, OUT_WIDTH_6, OUT_DEPTH_6);

free(out_pool_5);

relu(out_6, OUT_HEIGHT_6, OUT_WIDTH_6, OUT_DEPTH_6);

/******************************** Layer 7 *********************************/
alloc_size = OUT_HEIGHT_7*OUT_WIDTH_7*OUT_DEPTH_7;
float * out_7 = malloc(alloc_size * sizeof(float));
if (!out_7) { perror("malloc␣failed"); exit(EXIT_FAILURE); };

      fc(out_6, weight_7, bias_7, out_7,
              IN_HEIGHT_7, IN_WIDTH_7, IN_DEPTH_7,
              OUT_HEIGHT_7, OUT_WIDTH_7, OUT_DEPTH_7);

free(out_6);

relu(out_7, OUT_HEIGHT_7, OUT_WIDTH_7, OUT_DEPTH_7);
/******************************** Layer 8 *********************************/
alloc_size = OUT_HEIGHT_8*OUT_WIDTH_8*OUT_DEPTH_8;
float * out_8 = malloc(alloc_size * sizeof(float));
if (!out_8) { perror("malloc␣failed"); exit(EXIT_FAILURE); };

      fc(out_7, weight_8, bias_8, out_8,
              IN_HEIGHT_8, IN_WIDTH_8, IN_DEPTH_8,
```

```
                    OUT_HEIGHT_8, OUT_WIDTH_8, OUT_DEPTH_8);

  free(out_7);

  alloc_size = OUT_HEIGHT_8*OUT_WIDTH_8*OUT_DEPTH_8;
  float * soft_8 = malloc(alloc_size * sizeof(float));
  if (!soft_8) { perror("malloc failed"); exit(EXIT_FAILURE); };

  softmax(out_8, soft_8, OUT_HEIGHT_8, OUT_WIDTH_8, OUT_DEPTH_8);

  free(out_8);

  return 0;
}
```