TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Analysis and Design Methodology of Convolutional Neural Networks mapping on Reconfigurable Logic

---

Author:
Tzanis FOTAKIS

Thesis Committee:
Prof. Apostolos DOLLAS
Prof. Michail LAGOUDAKIS
Prof. Sotirios IOANNIDIS

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

October 1, 2020

Abstract

Diploma Thesis

Analysis and Design Methodology of Convolutional Neural Networks mapping on Reconfigurable Logic

by Tzanis FOTAKIS

Over the last few years, Convolutional Neural Networks have proved their abilities in several fields of study, with the research community continuing to surprise the world with new and paradoxical use cases, and even more exciting results. The rise of neural networks in general, and especially CNNs, creates a necessity for hardware acceleration of such computationally complex applications to achieve high-performance and energy-efficiency. Due to the fact that neural networks are highly parallelizable, they can exploit FPGA's hardware flexibility. This study presents a hardware platform targeted for FPGA devices for easy and structured implementation of neural network inference accelerators. It is designed with flexibility and versatility in mind, capable of being transferred to various FPGA devices. Furthermore, it is scalable for multi-FPGA implementations, using platforms such as the FORTH QFDB, a custom four-FPGA platform. In addition, it is extendable to enable for easy adding of new layer types and new layer accelerators. Moreover, it can run various CNN models' inference, but most importantly, it provides easy experimentation and development of neural networks hardware accelerator architectures. The proposed platform is implemented for accelerating AlexNet's inference, an award-winning CNN whose robustness analysis is carried out to investigate the FPGA's strengths and weaknesses, studying the computational workloads, memory access patterns, memory and bandwidth reduction, as well as algorithmic optimizations. A comparison in inference performance metrics is presented between the proposed platform, a CPU, a GPU, and other Xilinx developed neural network accelerator platforms. Although there are no performance benefits of using an FPGA over a modern GPU, a potential for performance improvements appears with further development, focusing on the convolution accelerator, which exploits the platform's ease of use, extendability, and expandability.

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Apostolos Dollas, for his invaluable support and guidance throughout both the course of my studies and the work of this thesis, being an inspiration in both professional and personal level, generously providing his expertise and experiences. I would also like to thank him for broadening my horizons and giving me the opportunity to become a part of the research community in the field of Artificial Intelligence and Hardware Architecture design.

Moreover, I would like to thank my thesis committee, Prof. Michail Lagoudakis, and Prof. Sotirios Ioannidis, for evaluating my work.

Furthermore, I would like to thank the CARV team in FORTH for their excellent collaboration and guidance throughout this work, even outside their working hours, including Dr. Christos Kozanitis and Dr. Aggelos Ioannou for their hardware design expertise, and Dr. Gregory Tsagkatakis for his Neural Networks expertise.

In addition, I would like to acknowledge the TUC MHL staff, including but not limited to Mr. Andreas Brokalakis, Mr. Pavlos Malagonakis, and Mr. Markos Kimionis, whose support was essential for the completion of this work.

I would also like to express my thankfulness to my fellow students and colleagues of TUC MHL, George Pitsis, now a Ph.D. candidate, Charisios Loukas, and Maria Argyriou for our collaboration.

Last but not least, I would like to express my deepest gratitude to the people who have always and unquestionably been there for me throughout the years of my studies, my family, Emmanouil Fotakis, Maria Ntavrani and Voula Fotaki, friends and colleagues. My sincerest thanks to all of you.

Tzanis Fotakis,
Chania, 2020

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ALU	**A**rithmetic **L**ogic **U**nit
ANN	**A**rtificial **N**eural **N**etwork
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
AXI	**A**dvanced e**X**tensible **I**nterface
BRAM	**B**lock **R**andom **A**ccess **M**emory
BSP	**B**oard **S**upport **P**ackage
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessor **U**nit
CS	**C**omputer **S**cience
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
cuDNN	**C**UDA **D**eep **N**eural **N**etwork library
DDR4	**D**ouble **D**ata **R**ate type texbf4 memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DNN	**D**eep **N**eural **N**etwork
DPU	**D**eep Learning **P**rocessing **U**nit
DSP	**D**igital **S**ignal **P**rocessor
FC	**F**ully **C**onnected
FF	**F**lip **F**lop
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FORTH	**F**undation of **R**esearch and **T**echnology **H**ellas
FSBL	**F**irst **S**tage **B**oot **L**oader
GDDR6	**G**raphics **D**ouble **D**ata **R**ate type **6** memory
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**ight **P**erformance **C**omputing
ILA	**I**ntegrated **L**ogic **A**nalyzer
ILSVRC	**I**mageNet **L**arge **S**cale **V**isual **R**ecognition **C**hallenge
LUT	**L**ook **U**p **T**able

ML	**M**achine **L**earning
MLP	**M**ulti-**L**ayer **P**erceptron
MMIO	**M**emory-**M**apped **I/O**
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
MXU	**M**atrix Mutliplier **U**nit
PE	**P**rocessing **E**lement
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
QFDB	**Q**uad **F**PGA **D**aughter **B**oard
RAM	**R**andom **A**ccess **M**emory
ReLU	**R**ectified **L**inear **U**nit
SDK	**S**oftware **D**evelopment **K**it
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SM	**S**treaming **M**ultiprocessor
SLC	**S**econd **L**evel **C**odebook
SSE	**S**treaming **S**IMD **E**xtensions
SSD	**S**olid **S**tate **D**rive
TDP	**T**hermal **D**esign **P**ower
TPU	**T**ensor **P**rocessor **U**nit
URAM	**U**ltra **R**andom **A**ccess **M**emory
USD	**U**nited **S**tates **D**ollar

Dedicated to my family and friends…

Chapter 1

Introduction

Since the invention of the first computer, humankind is rapidly solving problems that are intellectually difficult for human beings but relatively easy for computers, as such problems can be described in detail with a formal list of mathematical rules. However, problems that are easy for humans, that are solved intuitively, like distinguishing the difference between a car and a person, or a spoken word and a bird's chirp, is a real challenge for computers and engineers [1]. Those problems cannot be described, at the time of writing, with sharply defined mathematical rules. Artificial Intelligence (AI) and Machine Learning (ML) study those types of problems, with many successes in the cost of highly computationally complex algorithms.

It is estimated that by the year 2025, the total amount of data created worldwide will rise to 163 ZettaBytes, while every minute of the year 2019, Americans used more than 4.4 PetaBytes of data [2]. It is evident that data management systems and knowledge extraction from them, also called Data Analysis, are urgent. Although such problems can be tackled using Artificial Intelligence and Machine Learning, it is extremely computationally intensive, if not even non-feasible, in a reasonable amount of time.

Fortunately, most of the algorithms used to tackle such problems come with high parallelism. Therefore, they can be expanded in the space domain, in other words, they can utilize more hardware resources to cut down on needs from the time domain. Of course, there are many different types of hardware resources, each with their advantages and disadvantages, from parallelism capabilities and energy efficiency to cost of production, reconfigurability, and reusability.

1.1 Motivation

Nowadays, the computational complexity of the aforementioned algorithms makes hardware acceleration a necessity, since running them on Central Processing Units (CPUs) is, while possible, the least efficient and fast solution. Although writing software for CPUs may be fast and easy, its running speed due to low parallelism and high power consumption, as a general propose piece of hardware, are far from ideal. For reference, at the time of writing, a top-grade server CPU, AMD EPYC 7002 Series (Figure 1.1), can provide up to 64 cores and 128 threads, at up to 2.25GHz base clock and 3.4GHz boost clock, with a rated Thermal Design Power (TDP) of 225Watts, and a list price of 4,425 USD [3].

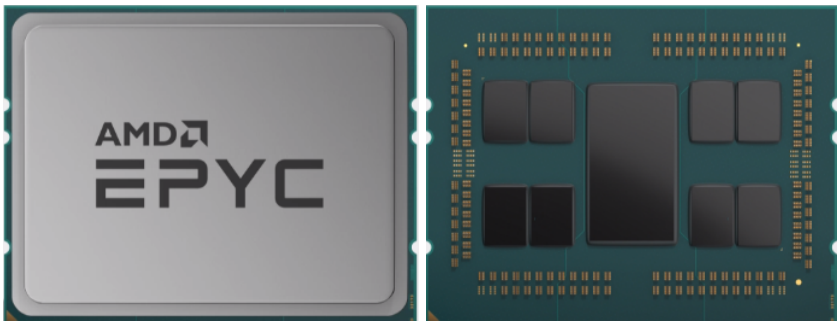

FIGURE 1.1: AMD Epyc 7002 series chip: URL

Graphics Processing Units (GPUs), on the other hand, provide much higher parallelism, while still being relatively easy for their software to be written. However, they can be costly to scale up, and their power consumption can be really high. For reference, at the time of writing, a top-grade GPU for ML, NVIDIA Titan RTX (Figure 1.2), provides up to 72 Streaming Multiprocessors, up to 4,608 CUDA Cores and up to 576 Tensor cores, with a rated base clock of 1,350 MHz and boost clock of 1,770 MHz, 24 GB of Graphics Double Data Rate (GDDR6) Memory and a power consumption of 280 Watts for 2,500 USD [4].

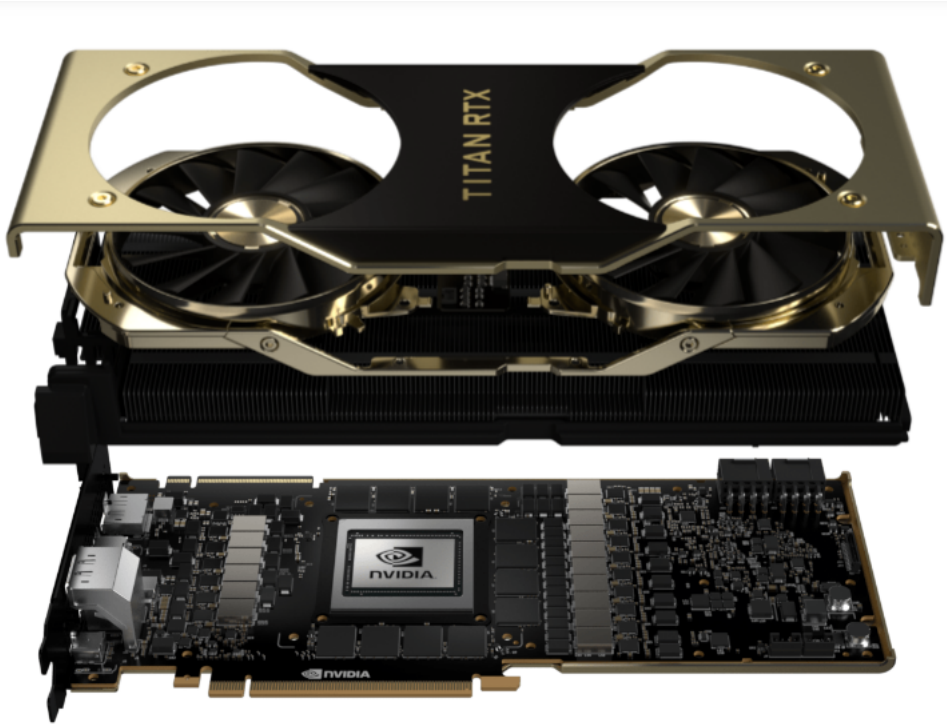

FIGURE 1.2: NVIDIA Titan RTX card: URL

Moreover, there are Application Specific Integrated Circuits (ASICs), which can provide the best parallelism capabilities and the lowest power consumption for a particular application. Unfortunately, they are very expensive to develop and produce, and they can only serve a single purpose, a single application. An example of such an ASIC is the Google Cloud Tensor Processing Unit (TPU) (Figure 1.3), which, for the third version (v3), in a single chip there are two TPU cores, each of which contains two scalar, vector, and matrix units (MXUs), and 16 GB of High Bandwidth Memory (HBM) [5].

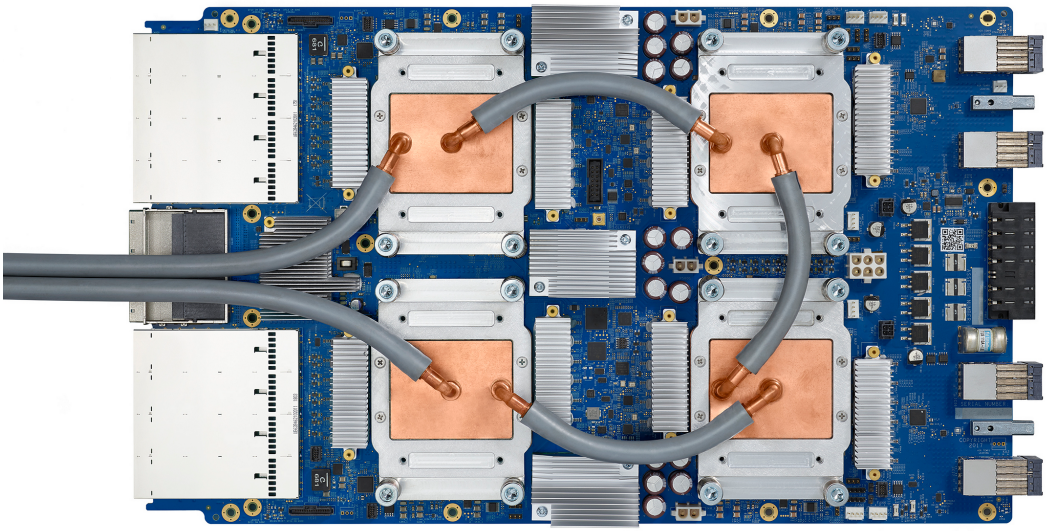

FIGURE 1.3: Google's TPU v3 - 4 chips, 2 cores per chip: URL

On the contrary, Field Programmable Gate Arrays (FPGAs) are bridging the gap between the GPUs' flexibility and the ASICs' performance and power consumption. An example FPGA Hardware targeted for High-Performance Computing (HPC) is the Quad-FPGA Daughter Board (QFDB) (Figure 1.4) [6], developed by the Foundation of Research and Technology Hellas (FORTH) [7], combines four interconnected Xilinx Zynq Ultrascale+ Multi-Processor Systems on Chip (MPSoCs), with 16GB of DDR4 memory and an M.2 Solid State Drive (SSD).

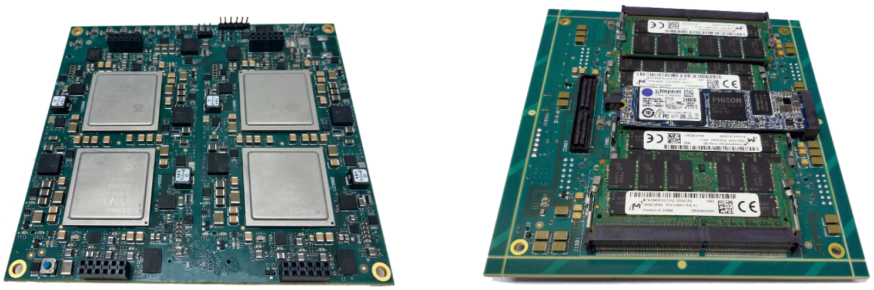

FIGURE 1.4: FORTH QFDB, top-view (left) and bottom-view (right): URL

In this work, the FPGAs' benefits are being utilized to create a hardware accelerator that can speed up the inference of Convolutional Neural Networks (CNNs), a branch of Deep Neural Networks (DNNs), which is a subfield of Machine Learning. FPGA based CNN inference accelerators are not only

targeting High-Performance Computing (HPC) systems like data-centers [8], but also embedded systems [9] like mobile phones and aerospace, where low-power, low-energy and low-latency are of great interest.

1.2 Scientific Goals and Contributions

Initially, this diploma thesis's goal was to design and implement a CNN inference FPGA accelerator, specifically for AlexNet, an award-winning CNN, and create a performance benefit over CPUs and GPUs. However, AlexNet, VGG, and many other CNNs use the same neural network layers, with differences in layer organization and hyper-parameter configuration. Therefore, a generalized CNN inference FPGA accelerator was designed and implemented to run any network that uses convolution, max-pooling, ReLU, and fully-connected layers. AlexNet is used as a reference for CNNs with similar characteristics, and as a benchmark for this thesis proposed platform.

This study presents a hardware platform targeted for FPGA devices for easy and structured implementation of neural network inference accelerators. It is designed with flexibility and versatility in mind, capable of being transferred to various FPGA devices. Furthermore, it is scalable for multi-FPGA implementations, using platforms such as the FORTH QFDB, a custom four-FPGA platform. In addition, it is extendable to enable for easy adding of new layer types and new layer accelerators, but most importantly, it provides easy experimentation and development of neural networks hardware accelerator architectures.

The theoretical modeling and robustness analysis has been conducted using AlexNet as a reference to investigate the FPGA's strengths and weaknesses, study the computational workloads, memory access patterns, and investigate and develop memory and bandwidth reduction techniques, including double, single, and half floating-points, and fixed-points with various bit-widths representations for both activations and parameters, as well as algorithmic optimizations.

A comparison in inference performance metrics is presented between the proposed platform, a CPU, a GPU, and other Xilinx developed neural network accelerator platforms. Although there were no performance benefits

using an FPGA over a modern GPU, a potential for performance improvements appears with further development, focusing on the convolution accelerator, which exploits the platform's ease of use, extendability, and expandability.

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** The theoretical background of Machine Learning, with emphasis on Convolutional Neural Networks, is described.
- **Chapter 3 - Related Work:** The related work on the field of Convolutional Neural Networks and their hardware implementations is described.
- **Chapter 4 - Theoretical Modeling and Robustness Analysis:** AlexNet's characteristics and sensitivity are studied using this work's theoretical modeling on PyTorch, C/C++ and Matlab. Also, techniques on memory footprint and computation complexity reduction are described.
- **Chapter 5 - FPGA Implementation:** This work's CNN inference FPGA platform is being described.
- **Chapter 6 - Results:** Metrics, such as the throughput, latency, power and energy consumption, are compared between the various available technologies and platforms, including an Intel i7 4710MQ CPU, an NVIDIA RTX 2060 Super 8GB GPU, a Xilinx CHaiDNN FPGA accelerator, a Xilinx DPU FPGA accelerator, and this work's platform.
- **Chapter 7 - Conclusions and Future Work:** This work is being concluded, and directions for future work and possible extensions are given.

Chapter 2

Theoretical Background

The theoretical background of Machine Learning and Convolutional Neural Networks is being described below.

2.1 Machine Learning

Machine Learning, the name of which was first proposed in 1959 by Arthur Samuel [10], is a subset of Artificial Intelligence, a Computer Science (CS) field that studies algorithms and statistical models capable of performing specific tasks, such as prediction or decision making, without being explicitly programmed. Instead, sample data are used, also known as "training data", for the machine to "learn" to distinguish useful patterns on the input data capable of creating the needed output, e.g., decision or prediction. There are numerous approaches [11] on the learning algorithms types, as well as on the model types used to get trained.

Such algorithm types, at the time of writing, include, but are not limited to:

- **Supervised Learning:** Algorithms that learn by using "labeled" sample data, " containing both the inputs and their desired outputs for classification and regression.
- **Unsupervised Learning:** In contrast with the Supervised Learning, unlabeled sample data are used to discover structures that could group or cluster them.
- **Reinforcement Learning:** Algorithms responsible for taking actions in an environment, often also described as software agents, maximize a specific metric, many of which use dynamic programming techniques.

- **Feature Learning:** Algorithms that by combining or even discarding features from the input samples, try to create a new, more useful set of features. One of the most popular algorithms of this category is Principal Components Analysis (PCA).
- **Anomaly Detection:** Algorithms that try to identify outlier samples, which are characterized by their significant difference compared to the majority of the data used. Such algorithms are often used in noise reduction, data mining, and even security and defense systems.
- **Association Rule Learning:** Algorithms that aim to discover strong relationships between features.

Such model types, at the time of writing, include, but are not limited to:

- **Artificial Neural Networks (ANN):** Also known as Connectionist Systems, imitate the biological brain's neural networks.
- **Decision Trees:** Make assumptions about the input items' target value (the decision tree's leaves) via its observations (the decision tree's branches). When the target takes continuous values, the Decision Tree is called a Regression Tree.
- **Support Vector Machines (SVM):** Used for classification and regression, mostly famous as non-probabilistic, binary, linear classifiers. They can also be used for non-linear classification using the kernel trick.
- **Bayesian Networks:** Represented as directed acyclic graphs, they can include probabilistic relationships.

Nowadays, most industries have already used Machine Learning in some sort, indicating the significance and variety of its capabilities. It is estimated [12] that by the year 2021, AI and ML spending will reach 57.6 Billion USD. Its applications include but are not limited to [13] [14], web page ranking, image recognition, email filtering, and spam detection, database mining, handwriting recognition, speech recognition, natural language processing, computer vision, image/video/text/speech generation, personalized marketing, traveling, dynamic pricing, healthcare, facial and fingerprint recognition and intrusion detection.

2.2 Artificial Neural Networks

It is widely accepted that the brain's most exceptional ability is pattern recognition, which is used to combine "data" from the organism's senses in a way to better understand its environment. Artificial Neural Networks (ANN), a highly popular sub-field of Machine Learning, try to imitate the brain's structure to solve such problems, a structure that has been developing and proving its capabilities for thousands of years.

While ANNs are inspired by biological neural networks, they are not identical. A neural network is a collection of connected neurons, through which electrical signals from sensor organs or other neurons are passed and processed. A biological neuron comprises four main parts; Dendrites, Cell body, Axon, and Synaptic terminals (Figure 2.1). A Dendrite and its Dendritic branches are used as the neuron's input, where sensors or other neurons get connected. A neuron can have multiple Dendrites. The neuron's cell body collects all the input signals and applies an "activation" function to create the output signal. Afterward, the output signal is transported through the Axon and distributed to the next neurons through the Synaptic terminals. The Synaptic terminals to Dendrites connections are called Synapses.

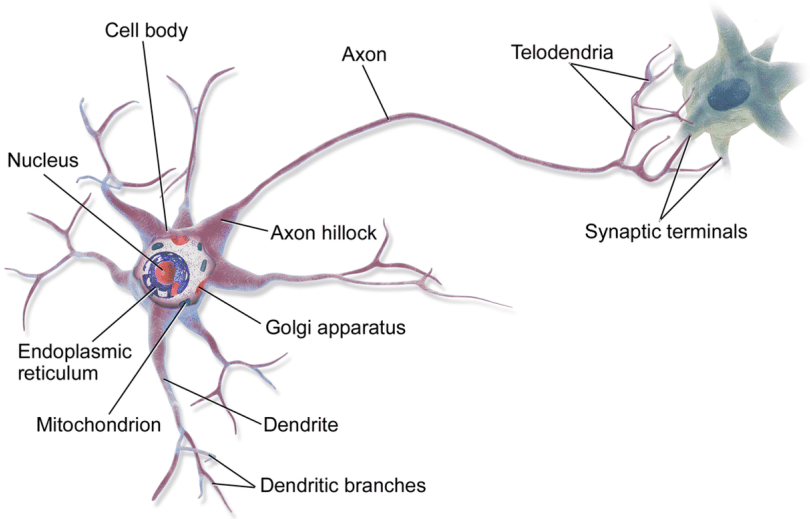

FIGURE 2.1: Standard structure of a biological neuron: URL

2.2.1 ANNs Basic components

Similarly to the biological neural networks, an ANN can be represented as a directed, weighted graph (Figure 2.2), whose vertices represent the biological neurons' cell bodies and its edges the biological synapses. The electrical signal used in biological neurons can be represented as a real number, and their outputs can be calculated by some non-linear function of the inputs' weighted sum. Each edge typically can have a weight, set during the training process, which amplifies or weakens the vertex's signal.

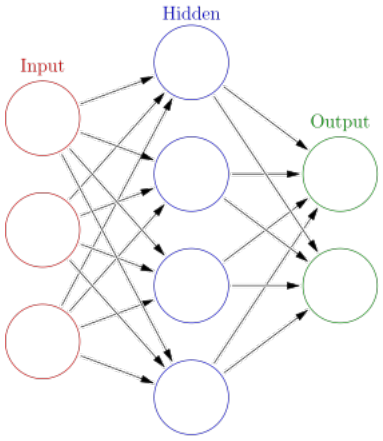

FIGURE 2.2: Simplified Neural Network Graph: URL

Neuron

A neuron receives real numbers as inputs, which are combined with their internal state, also known as activation, using an activation function and an optional threshold to produce the neuron's output.

Connections and Weights

Each neuron can be connected with multiple other neurons to be used as inputs and to feed them with its output. Each connection is characterized by its weight, which represents its relative importance.

Propagation Function

The propagation function calculates the weighted sum of each neuron's inputs and adds a bias term.

Activation Function

The activation function receives the propagation function's result and applies a transformation, which creates the neuron's final output. There are a lot of different activation functions, with specific characteristics for the training and inference process. However, they all provide a smooth and differentiable transition as the input values change. The most commonly used ones are shown below [15].

- **Identity:** $f(x) = x$
  No transformation is applied.
- **Binary Step:** $f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$
  While being the original activation function developed when neural networks were invented, it is no longer used as it is incompatible with backpropagation. Backpropagation is the process of updating the weights during the training phase using the gradient descent algorithm. The binary step function is not convex; hence, gradient descent is unable to find a local minimum.
- **Logistic (Sigmoid or Soft step):** $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$
  Often used, however, in real-world neural networks, it is avoided due to the vanishing gradient problem [16].
- **TanH:** $f(x) = tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}-e^{-x}}$
  Same as Logistic.
- **Rectified Linear Unit (ReLU):** $f(x) = \begin{cases} 0 & x < 0 \\ x & x > 0 \end{cases}$
  The most popular activation function, due to its fast backpropagation speeds, its low penalty on generalization accuracy, and its resistance to saturation conditions [17].
- **Softmax:** $f_i(\mathbf{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}$, for i = 1, ..., J
  Commonly used as a final output activation function for multiclass classification. It normalizes the output to [0, 1], and makes its sum equal to 1. After this transformation, the i-th output's value designates the input's probability to be the class i.

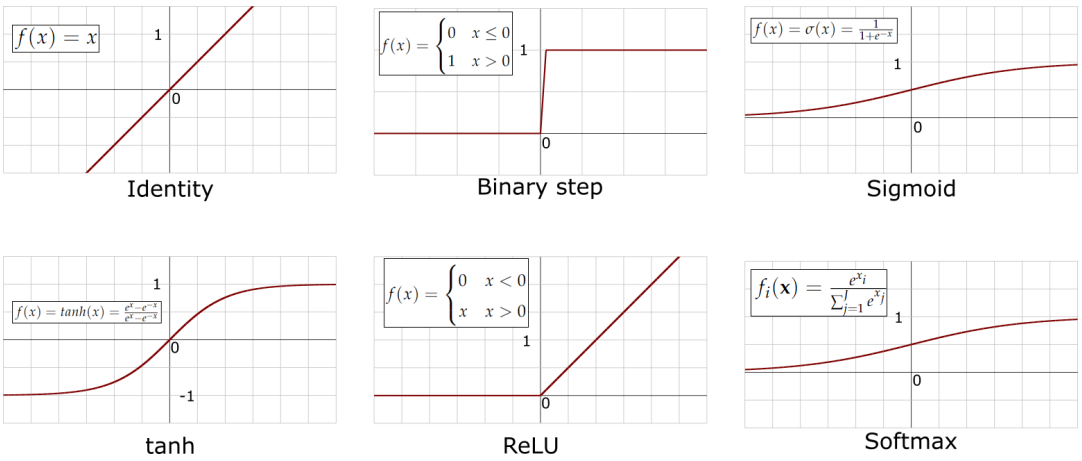

FIGURE 2.3: Activation Function Graphs

2.2.2 Organization

An ANN's neurons are typically organized into groups, called layers, in which each neuron has the same distance from the inputs as all the other neurons of its group. The input layer is the layer that gets as inputs the external data, and the output layer, the last layer in the graph, is the one that produces the final output results. Any in-between layer is called a hidden layer. An ANN is called a Deep Neural Network (DNN), when, by convention, it has three or more hidden layers.

2.2.3 ANN Architectures

There are many ANN architectures, each one serving different use cases. They are separated into two main groups, Feedforward networks and Recurrent networks [18].

Feedforward Networks

The basic idea with the feedforward networks is that the data flows from the input layer through the hidden layers to the output layer without any cycles, so they can be represented as directed, acyclic graphs. Some architectures of this group are:

- Multiclass Perceptron
- Group method of data handling
- Autoencoder

- Probabilistic
- Time delay
- Convolutional
- Deep stacking network

Recurrent Networks

Recurrent Neural Networks (RNN), similarly to the Feedforward networks, data flows from the input layer through the hidden layers to the output layer. However, they allow for data cycles, in other words, outputs of the $layer_n$ can be fed to the inputs of the $layer_{n-1}$, so they can be represented as directed, cyclic graphs. Some architectures of this group are:

- Fully recurrent
- Simple recurrent
- Reservoir computing
- Long short-term memory
- Bi-directional
- Hierarchical
- Stochastic
- Genetic Scale

For every ANN Architecture, there are specific types of layers that apply different kinds of mathematical operations on their input data. Each layer type has characteristics on the way its mathematical operations are applied to its input. Those characteristics are generally called hyperparameters. For example, in a Fully-Connected layer, a hyperparameter is its number of outputs. Hyperparameters are set by the engineers during the training phase, which are fine-tuned, concerning the application's input data.

This work is focused on the Convolutional Neural Networks (CNN), which are described in detail below.

2.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are deep feedforward neural networks that specialize in processing data with grid-like topologies and are typically used in visual imagery analysis. They are simple neural networks that, for at least one of their layers, use the convolution mathematical operation instead of the general matrix multiplication [1].

CNNs imitate the brain's visual cortex, which is the area responsible for the visual processes. Cortical neurons cover small areas of the visual field, with partial overlaps resulting in full visual field coverage.

CNNs most significant advantage over other image classification algorithms is their little need for pre-processing, meaning that their filters are learned during the training phase while using traditional algorithms, they have to be hand-engineered.

Some of their applications are image and video recognition, image classification, object detection, recommendation systems, medical image analysis, natural language processing, and financial time series [19] (e.g. [20], [21], [22], [23], [24], [25]).

2.3.1 Structure

A typical CNN consists of two parts. The first part includes multiple convolutional and pooling layers, which extract features from the given input. The second part, also known as the classifier, includes multiple fully connected layers, which classify the given input using the features extracted from the first part (Figure 2.4).

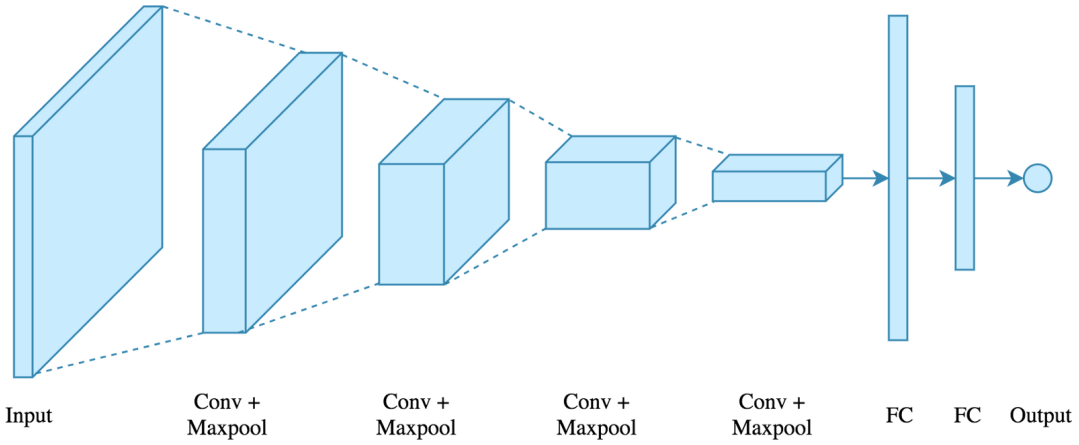

FIGURE 2.4: Typical CNN Architecture - First five layers (Convolution + Max Pooling layers) used for feature extraction, last three layers (Fully-Connected) used for classification, also called the classifier: URL

The input data is typically structured as multidimensional arrays, also known as tensors. For example, if the input data are RGB images, then the input tensor's shape is (number of images) x (image width) x (image height) x (image depth), where image depth is called the image's color channels, in this example 3 channels. Moreover, the input data can also be grayscale images or even hyperspectral images. Hyperspectral images have multiple color channels, even in the non-visible for the human eye spectrum, with applications including the medical and space fields. There are also implementations such as 1D and 3D CNNs, but this work examines 2D CNNs only.

The trainable parameters (weights and biases) needed for the computation are initially assigned random values [26], rendering the network useless. However, during the training phase, using the backpropagation process [27], those weights are being optimized to form features from the training set. The trainable parameters are also considered to be shared [28], meaning that the same parameters are to be used in the entirety of the input data, dramatically decreasing their number and consequently their memory footprint and also increasing the network robustness against overfitting.

As stated above, there are three types of layers in 2D CNNs; Convolutional, Pooling, and Fully-Connected layers. Each layer is being described below.

Convolutional Layer

A convolutional layer creates and outputs a similarity map between the input data and the convolution's filters, also known as kernels. More specifically, every filter is convolved across the input's width and height, producing their dot product. The result is multiple two-dimensional arrays, whose each cell holds the similarity of each filter to some spatial position in the input.

Each filter is a specific type of feature, depending on the training set. For example, in image classification, a filter can be a rough shape of cats' mustaches, so that after the convolution, it can be indicated if they are contained somewhere in the input image. If they are, then, to some probability defined during the training phase, there might be a cat in the input image.

Each convolutional layer is defined by its hyperparameters. Those are:

- **Kernel size:** The width and height of the kernels' (filters') size, typically, smaller than the given input.
- **Output channels:** The number of feature maps to be created as outputs. Consequently, the number of kernels used in this operation also equals the output channels number.
- **Stride:** The number of pixels to be skipped horizontally and vertically in each partial convolution. Typically, this number does not differ between the two dimensions.
- **Zero padding:** There might be a need for zero-padding the input to include as much data as possible in the final computation. There are three different ways of padding:
  - **Valid:** No padding is applied; some data may not be included in the computation.
  - **Same:** Applies the amount of padding needed to result in the same width and height as the input.
  - **Full:** Applies padding on the input's edges with a specified number of pixels per dimension.

The mathematical expression of the convolution operation is defined bellow (Equation 2.4) and visualized on Figure 2.5.

Let I be the input with C channels, H height and W width, and let K be the kernels with N number of kernels, C channels, KH height and KW width.

Also, let B be the bias with K values, and let S be the stride size and P be the padding size. So the convolution operation's output, Out, is defined as:

$$I_{padded}(c,i,j) = \begin{cases} 0, & i \in [1,P], j \in [1,P] \\ I(c,i-P,j-P), & i \in [P+1,P+1+H], j \in [P+1,P+1+W] \\ 0, & i \in [H+1,H+1+P], j \in [W+1,W+1+P] \end{cases} \tag{2.1}$$

$$OH = \frac{H+2P-KH}{S}+1 \tag{2.2}$$

$$OW = \frac{W+2P-KW}{S}+1 \tag{2.3}$$

$$Out(k,i,j) = B(k) + \sum_{c=1}^{C}\sum_{kh=1}^{KH}\sum_{kw=1}^{KW} I_{padded}(c,kh+(i-1)*S,kw+(j-1)*S)K(k,c,kh,kw),$$

$$\text{for k = 1, 2, ..., C,}$$

$$\text{for i = 1, 2, ..., OH,}$$

$$\text{for j = 1, 2, ..., OW} \tag{2.4}$$

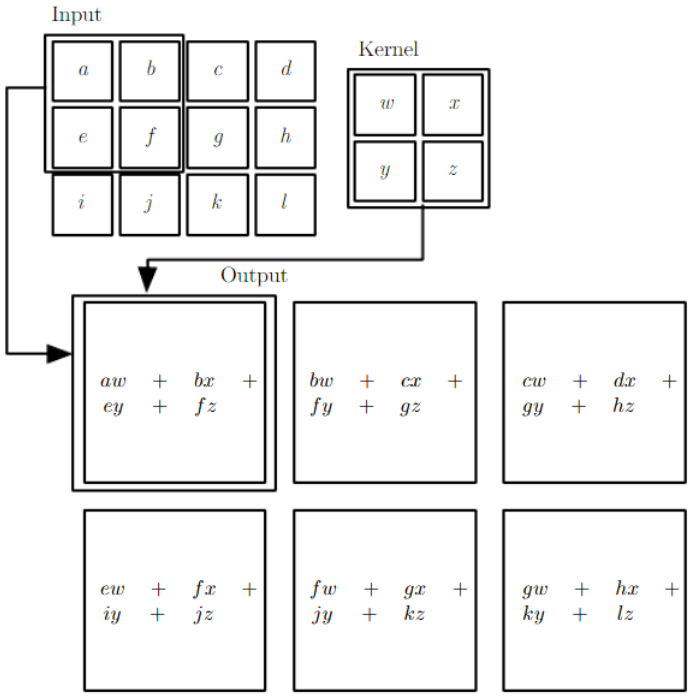

FIGURE 2.5: Convolution Operation - A 2x2 Kernel filter is applied on the 4x3 example input matrix with stride 1 and valid padding. Figure from [1].

Nowadays, most applications might require multiple Convolutional layers to extract useful features from their complex inputs. Deeper architectures, in general, create more detailed characteristics. Furthermore, activation functions can be interjected between adjacent Convolutional layers to enhance the network's non-linearity. Activation functions can make the network function as a universal function approximator [29].

Pooling Layer

A pooling layer sub-samples its input to decrease the computation footprint needed for the next layers and make the network more prone to over-fitting. It reduces the input dimensions by combining multiple neurons into a single neuron. Max-Pooling layers combine groups of neurons by outputting their maximum value. Average-Pooling layers combine groups of neurons by outputting their average value.

Similarly to the convolutional layer, a pooling layer slides a window of some size, called kernel size, across the input data. The data to be combined are

those that the sliding window has selected. In 2D CNNs, the pooling layers have 2D windows; the channels are not combined.

A pooling layer can be local, combining small groups of neurons, which means that the layer's kernel size is small compared to the input size. It can also be global, combining the whole input to a single neuron.

Each pooling layer is defined by its hyperparameters. Those are:

- **Kernel size:** The kernel's (sliding window's) width and height.
- **Stride:** The number of pixels to be skipped horizontally and vertically in each slide. Typically, this number does not differ between the two dimensions.

The mathematical expression of the average-pooling operation (Equation 2.7) and max-pooling operation (Equation 2.8) is defined and visualized (Figure 2.6) bellow.

Let I be the input with C channels, H height and W width, let KH be the kernel's height, let KW be the kernel's width, and let S be the stride size. So the pooling operations are defined as:

$$OH = \frac{H - KH}{S} + 1 \tag{2.5}$$

$$OW = \frac{W - KW}{S} + 1 \tag{2.6}$$

$$\begin{aligned} AvgPool(c,i,j) = \frac{\sum_{kh=1}^{KH}\sum_{kw=1}^{KW} I(c, kh + i * KH, kw + j * KW)}{KH * KW}, \\ \text{for c = 1, 2, ..., C,} \\ \text{for i = 1, 2, ..., OH,} \\ \text{for j = 1, 2, ..., OW} \end{aligned} \tag{2.7}$$

$$\begin{aligned} MaxPool(c,i,j) = \max_{1 \le kh \le KH, 1 \le kw \le KW} I(c, kh + i * KH, kw + j * KW), \\ \text{for c = 1, 2, ..., C,} \\ \text{for i = 1, 2, ..., OH,} \\ \text{for j = 1, 2, ..., OW} \end{aligned} \tag{2.8}$$

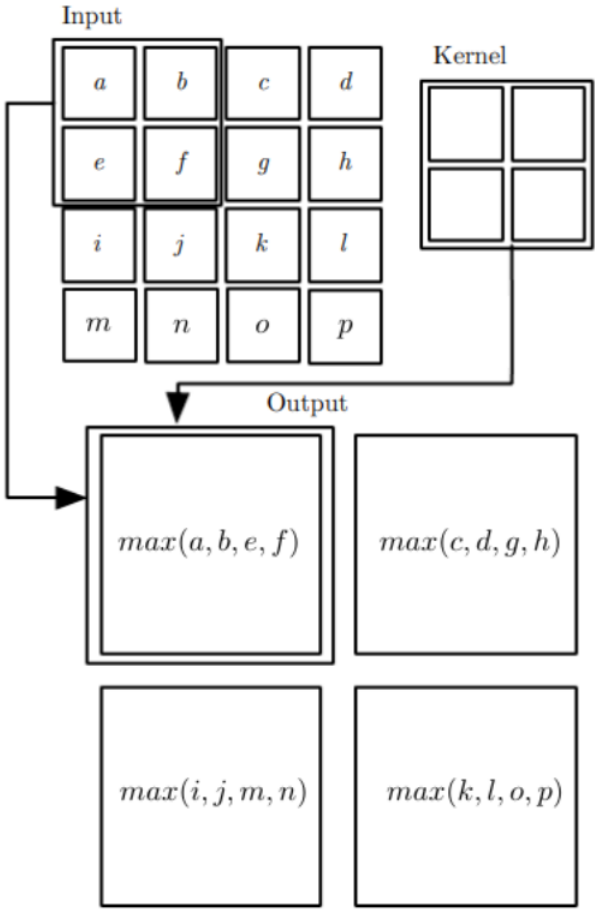

FIGURE 2.6: Max-Pooling Operation - A max operation is applied with a 2x2 Kernel on the 4x4 example input matrix with stride 2. Figure from [1].

Fully-Connected Layer

The CNNs classifier part comprises several Fully-Connected layers, which serve as the high-level reasoning. It is the part that finally classifies the given input.

A Fully-Connected layer is the simplest type of layer, as it is the one used in Multi-Layer Perceptron (MLP) neural networks. More specifically, it receives input, with which it computes a weighted sum for each of its output values. This input is derived from the flattened output of several convolutional and pooling layers.

Each Fully-Connected layer is defined by its hyperparameters. Those are:

- **Output Features:** The number of features to output. Consequently, this also configures the number of weights needed for the computation,

hence, its memory and computation footprint.

The mathematical expression of the Fully-Connected layer's operation (Equation 2.9) is defined bellow.

Let I be the input with N input features, let W be the layers weights, let B be the layer's bias, and let M be the layer's output features. So the Fully-Connected layer's output, Out, is defined as:

$$Out(i) = B(i) + \sum_{j=1}^{N} I(j)W(i,j), \text{for i = 1, 2, ..., M} \tag{2.9}$$

The parameters' reuse of Fully-Connected layers from a specific application to another cannot be done since they are strictly bonded to the classes and high-level features of the particular Convolutional neural network.

The final Fully-Connected layers is typically followed by a Softmax activation layer, which, as stated on section 2.2.1, it calculates the probability of the input being a certain class. The use of Softmax enables the confidence quantification for every estimation, and easy troubleshooting when the input it misclassified.

Activation Layer

There can be an activation layer after each Convolutional and Fully-Connected layer, which applies an activation function on the output of its previous layer. An activation layer increases the network's non-linearity without affecting the convolutional layers' receptive fields. The activation function can be one of those presented in section 2.2.1, but ReLU is generally preferred, due to its fast training characteristics and its low penalty on generalization accuracy.

2.4 Theoretical knowledge sources

The aforementioned theoretical background was mostly obtained from the Statistical Modeling and Pattern Recognition course of Electrical and Computer Engineering school at the Technical University of Crete. In addition, the book Deep Learning [1] was used when finer details were needed. Moreover, the Udacity course Intro to Deep Learning with PyTorch by Facebook AI [30] was used for a more hands-on approach, focusing on PyTorch and

Python in general. Last but not least, a great resource has been all the papers mentioned above.

Chapter 3

Related Work

3.1 CNN Architectures

The essential part of every machine learning application is its dataset. A dataset is the collection of data that a machine learning application can be based on; for a neural network application, a dataset is the collection of data that the network is trained and tested. Nowadays, there are numerous datasets for visual recognition applications, with various sizes and qualities. The size of a dataset defines the number of data per class, and its quality can define many aspects of the data, such as their noise characteristics and their factual correctness of assigned labels (the creators of the dataset may wrongly label data). Some of the most popular datasets are MNIST with grayscale images of handwritten digits [31] [32], Fashion-MNIST with grayscale images of various pieces of clothing [33] [34], CIFAR-10 & CIFAR-100 with color images of 10 and 100 classes, respectively, of everyday items [35] [36], Microsoft COCO with color images for object recognition / detection of everyday items [37] [38] and ImageNet with high resolution color images of 22000 classes of everyday items [39] [40].

ImageNet is one of the biggest datasets, containing more than 14 million hand-annotated images and more than 1 million bounding boxes for those images. Since 2010, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is organized annually by the ImageNet project, where software programs compete in the classification of a trimmed list of one thousand non-overlapping classes. The contestant softwares have been of many different types throughout the years; however, from the ILSVRC 2012 and then on, Convolutional Neural Networks have dominated the challenge, achieving near human-like accuracy. A CNN called AlexNet [41] managed to achieve a top-5 error of 15.3% in the ILSVRC 2012, where the top-5 error rate is the

fraction of test data for which the correct label is not among the five most probable labels.

Many CNN architectures have been created using the aforementioned datasets. Some of the most important ones are being described below.

3.1.1 LeNet-5

LeNet-5 [42] (Figure 3.1), created by LeCun et al in 1998, was, at the time, a pioneering 7-layer convolutional neural network designed to recognize simple 32x32 grayscale digit images, similar to those on the MNIST dataset. It uses two convolutional layers, two pooling layers, and three fully-connected layers. However, it can only support low-resolution images and few classes, due to its low learning capacity. For higher resolution images, deeper networks are required, which was not feasible back then because of limited hardware performance. LeNet-5 was the first success for CNNs.

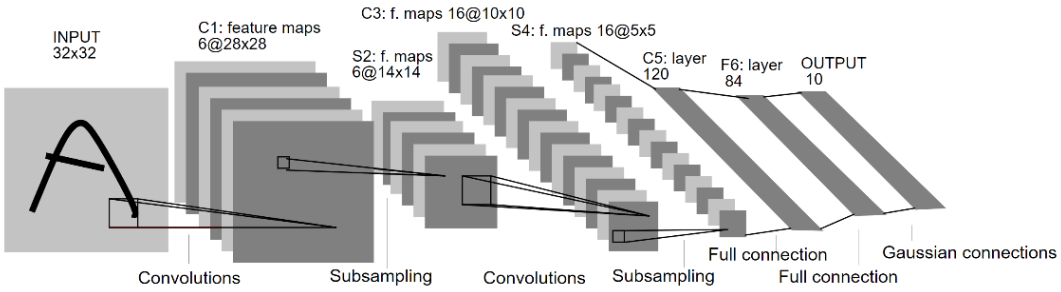

FIGURE 3.1: LeNet-5 architecture

3.1.2 AlexNet

AlexNet [41] (Figure 3.2), created by Alex Krizhevsky et al in 2012, outperformed all prior contestants of the ILSVRC by almost twice increase in accuracy, reducing the top-5 error rate from 26.2% to 15.3%. It takes as input RGB 224x224 images. This is achieved by using the same types of layers as LeNet-5, but more of them, making it is deeper and with more filters per layer. Using such deep networks was made feasible due to the utilization of GPUs during the training phase, whose performance had been significantly increased. AlexNet was, also, designed to run on two GPUs simultaneously, further exploiting the CNNs' parallelism characteristics. It needed six days in time for successful training on two NVIDIA GTX 580 GPUs.

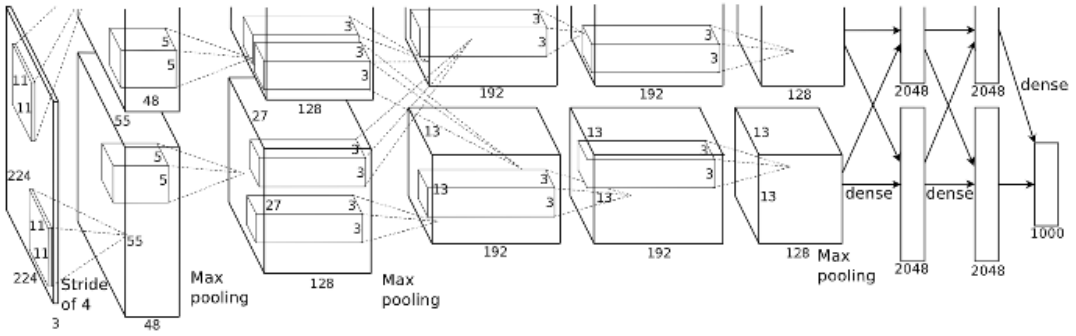

FIGURE 3.2: AlexNet architecture

Since AlexNet, most, if not all, CNNs are at least as deep, to achieve high learning capacity and accuracy.

Nowadays, AlexNet is one of the most well-known CNNs and is often used as a benchmark for various hardware solutions, due to its high complexity and number of parameters needed. In total, it uses more than 61 million parameters, which results in about 250MB.

3.1.3 ZFNet

ZFNet [43] (Figure 3.3) is a fine-tuned version of AlexNet by Matthew Zeiler and Rob Fergus, which won the ILSVRC 2013, achieving a top-5 error rate of 14.8%.

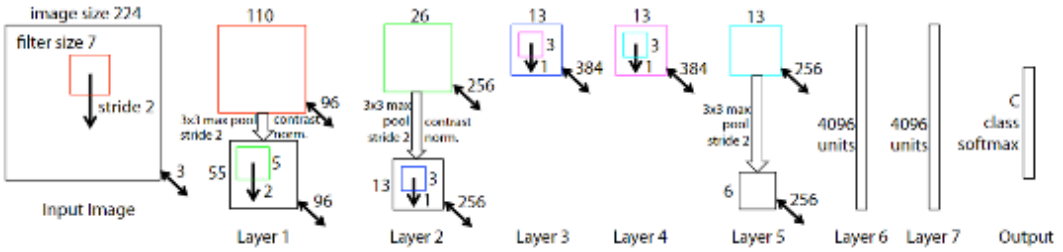

FIGURE 3.3: ZFNet architecture

3.1.4 GoogLeNet / Inception

GoogLeNet [20] (Figure 3.4), also known as Inception v1, designed by Google, won the ILSVRC 2014 with a top-5 error rate of 6.67%. Provided that GoogLeNet's top-5 error rate was near to the human level, organizers had to evaluate

these results with the help of a human expert, previously trained for a few days, who achieved a top-5 error rate of 5.1% for a single model and 3.6% for the ensemble. This architecture, inspired by LeNet-5, consists of 22 layers but reduces the number of parameters compared to AlexNet from 61 million to only 4 million. The parameter reduction was achieved using a special module called Inception, which is based on several very small convolutions. To further improve the accuracy, techniques such as batch normalization, image distortions, and RMSProp were used. Note that the auxiliary classifiers, Softmax0 and Softmax1, are only used during the training phase to combat the vanishing gradient problem and provide regularization.

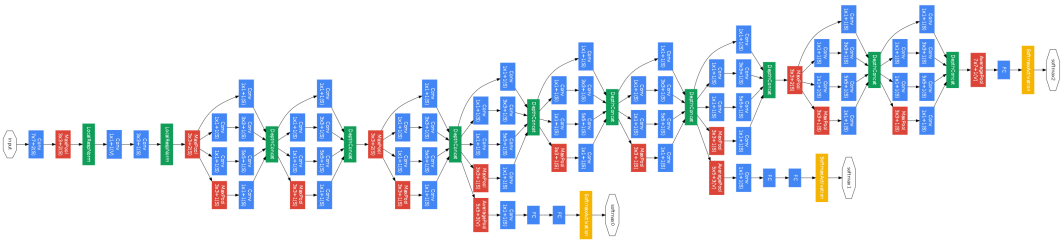

FIGURE 3.4: GoogLeNet/Inception v1 architecture

3.1.5 VGGNet

VGGNet [44] (Figure 3.5), designed by Simonyan and Zisserman, was the runner-up ate the ILSVRC 2014. The original architecture consists of 16 layers, but there are other variants with more or fewer layers. Compared to AlexNet, it uses more filters, and it was trained with 4 GPUs for up to 3 weeks. Nowadays, it is the most preferred network for image feature extraction; however, it can be very challenging, due to its 138 million parameters.

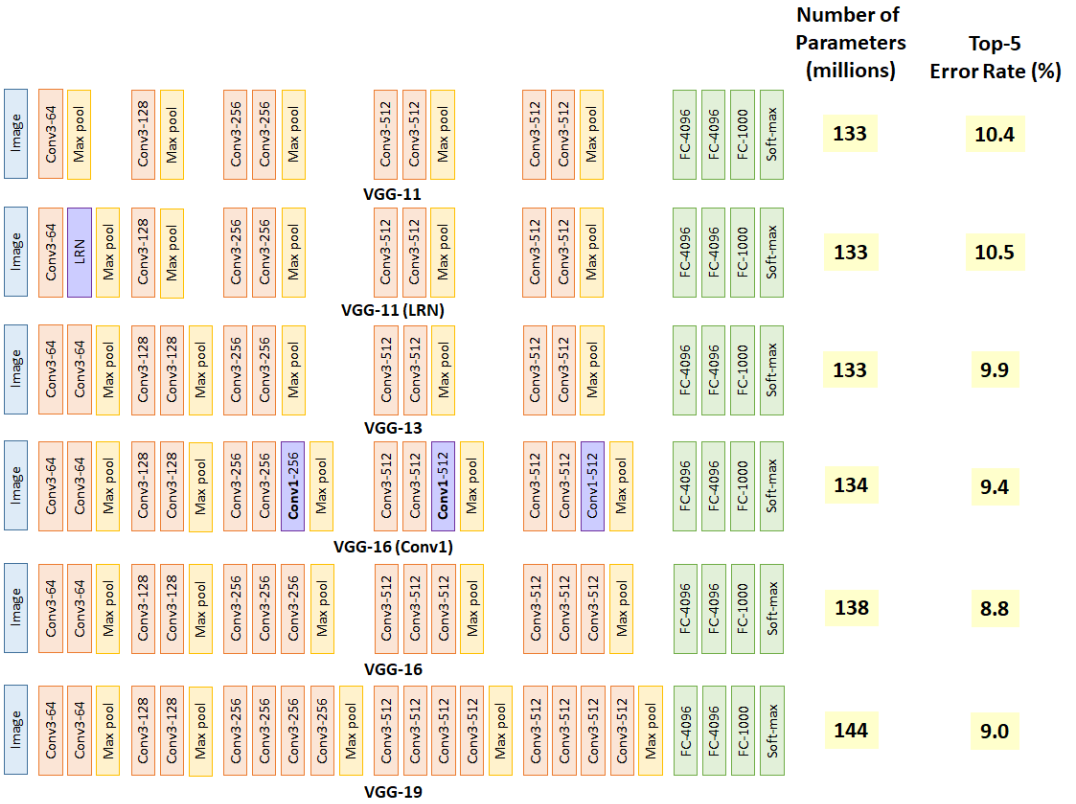

FIGURE 3.5: VGGNet architectures: URL

3.1.6 ResNet

ResNet [45] (Figure 3.6), designed by Microsoft, won the ILSVRC 2015 with an outstanding top-5 error rate of 3.57%, beating the human-level performance. This was achieved with the use of 152 layers. Although the layers' number may be high, its complexity is lower than the VGGNet due to its architecture called Residual Neural Network. It uses gated (recurrent) units, a module inspired by the recent successful elements used in RNNs, which, in a sense, skips connections. In addition, heavy batch normalizations are also used.

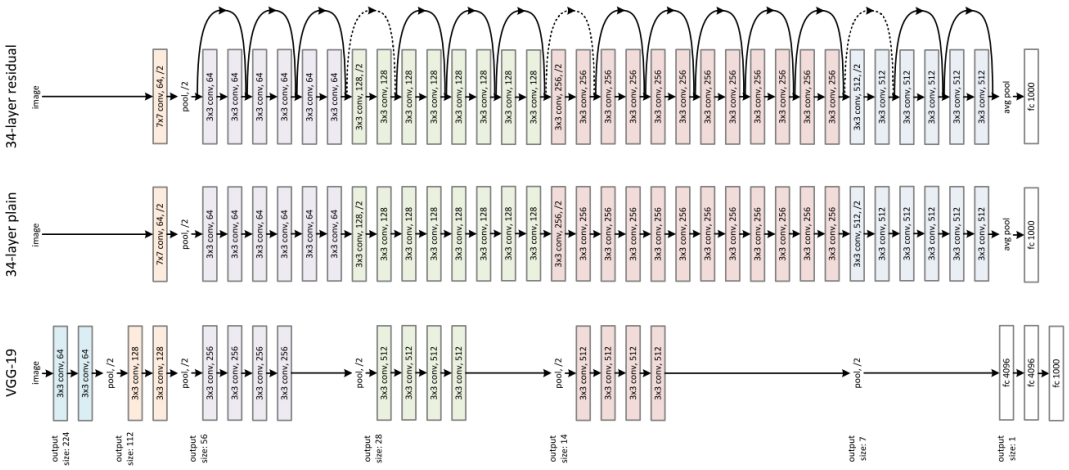

FIGURE 3.6: ResNet architecture - Bottom: the VGG-19 model (19.6 GFLOPs) as a reference. Middle: a plain network with 34 parameter layers (3.6 GFLOPs). Top: a residual network with 34 parameter layers (3.6 GFLOPs). The dotted shortcuts increase dimensions.

3.1.7 Summary

One can observe that all the aforementioned architectures use the same types of layers found on classical CNNs. They only differ in some training techniques, their depths, and hyperparameters. In this work, AlexNet is the primary benchmark used to test the various hardware architectures, because of its simplicity and its high complexity.

3.2 Deep Learning Software Frameworks

There are many software frameworks available for developers to use for their deep learning applications. Their differences, apart from their syntax, are found in the amount of abstraction, portability, and environment. Some of the most popular ones are described below.

3.2.1 Keras

Keras [46] [47] is a high-level library for Python, built on top of other lower-level frameworks such as TensorFlow, Theano and CNTK. While the high-level approach reduces the creation of massive deep learning models to single-line functions, it also makes the library's environment less configurable. It is

best suited for learning and prototyping, with the abstraction of the mathematical operations applied.

3.2.2 CAFFE

CAFFE (Convolutional Architecture for Fast Feature Embedding) [48] [49] [50] is a deep learning framework written in C++, with a Python interface, originally developed at University of California, Berkley. It supports CNN, RCNN, LSTM, and Fully-Connected neural network models, with CPU and GPU acceleration. Its successor, CAFFE2, also supports RNNs. CAFFE2 is now merged into PyTorch.

3.2.3 PyTorch

PyTorch [51] [52] developed by Facebook, is a lower-level framework based on Torch, which supports any kind of neural networks. It contains many pre-trained models, most of those in section 3.1, and can also utilize GPU acceleration. It operates with a dynamically updated graph, which allows for changes to the architecture in the process. It is best suited for small projects, prototyping, and research purposes.

3.2.4 TensorFlow

TensorFlow [53] [54] [55] is the most popular deep learning framework nowadays, released on November of 2015 [56]. Developed by Google, it has interfaces for Python, Javascript, C++, C#, Java, Go, and Julia. It is the most used framework for production, with CPU, GPU, and TPU acceleration support. It can, not only, run on powerful computing clusters, but also, on mobile platforms and embedded systems. While it is a lower-level framework and needs much coding, it provides high configurability. In contrast to PyTorch, it operates with a static computation graph, which enhances its efficiency but sacrifices the ease of model modifications; for every modification, a model retrain is needed.

3.3 Hardware Solutions

The most considerable portion of the computation needed in CNNs comes from the Multiply-Accumulate (MAC) operations on floating-point numbers.

Usually, networks are trained on GPUs, because they can fulfill the high parallelism needs. For even more exceptional performance on training, TPUs can be used. However, the training phase is considered, most of the time, to be a one-time procedure. Hence, the research community is more focused on accelerating the inference phase. While being a much less computationally intensive procedure compared to training, the inference phase can still be intensive enough to achieve real-time or even faster applications.

Neural networks' inference can be run on CPUs, GPUs, ASICs such as the TPU, and FPGAs. A brief description of each platform's advantages and limitations is given below.

3.3.1 CPUs

CPUs are multi-purpose processors that can be very flexible in the operations they can execute while being very easy to program. They can also execute Single Instruction Multiple Data (SIMD) instructions using the Advanced Vector Extensions (AVX) [57] and Streaming SIMD Extensions (SSE) [58], and utilize multiple cores. Even though CPUs run at clock speeds in the gigahertz range, they lack the vast amount of computation power compared to other solutions, and they can have high power consumption.

3.3.2 GPUs

Most of the aforementioned deep learning frameworks support GPU acceleration. GPUs can have hundreds or even thousands of more cores than CPUs, with even specialized ones called Tensor cores for tensor operations, which makes them ideal for executing algorithms with high parallelism characteristics, such as those used in deep learning. Vector processing creates the bulk of the GPUs' compute power with which the same operation is applied to a large amount of data simultaneously. This is achieved by using many Streaming Multiprocessors (SMs) on a single GPU die, which are vector processors that can process up to thousands of operations in a single clock cycle. In addition, due to the massive amounts of data that deep learning algorithms need to handle, High Bandwidth Memory (HBM) can be a perfect fit providing up to 750 GB/s compared to only 50 GB/s offered by traditional CPUs. Moreover, multiple GPUs in a single system can be utilized to further expand its parallelism capabilities.

As of right now, only NVIDIA GPUs are widely supported by most frameworks. NVIDIA's GPUs are optimized for deep learning frameworks with compatibility for Compute Unified Device Architecture Software Development Kit (CUDA SDK) [59], which supports many programming languages such as C and C++, increasing the GPUs usability. NVIDIA has also developed the CUDA Deep Neural Network library (cuDNN) [60] [61], designed to accelerate frameworks such as TensorFlow and PyTorch by providing highly-optimized implementations of routines like forward and backward convolution. Furthermore, NVIDIA's TensorRT [62] is an SDK for highly optimized inference applications, delivering low latency and high throughput (compared to CPUs). PyTorch also supports PyCUDA [63], NVIDIA's CUDA parallel computation API for Python.

Although GPUs provide very high performance and throughput, they can be very power inefficient and can introduce high latency per result relative to other solutions.

3.3.3 Tensor Processing Units (TPU)

Tensor Processing Unit (TPU) [64] is a custom Application Specific Integrated Circuit (ASIC) that accelerates the inference phase on neural networks designed by Google to improve cost-performance over GPUs. It is deployed to their datacenters since 2015 to accelerate 95% of their AI needs, and since 2017 Google made its TPU infrastructure available to the public on its Google Compute Engine. It was estimated that if every Android user began to use Google's voice search for three minutes per day, Google had to double their data centers [65]; instead, they developed their TPU.

The first architecture generation achieves up to 200x speedup compared to a server-class Intel Haswell CPU and up to 70x speedup compared to an NVIDIA K80 GPU. Nowadays, they have designed another two generations of its TPU (latest is TPU v3), and an edge computing solution called Coral Edge TPU [66], which comes in various form factors, achieving up to 4 TOPS using only 2 Watts. The first generation of TPUs targeted inference applications and was designed for high volume computation of as low as 8-bit precision. However, from the second generation, TPUs support floating-point arithmetic making them ideal for training accelerators.

Each TPU core provides 2 Matrix Multiplier Units (MXUs) and is provided

16GB of HBM (TPUv3) (Figure 3.7). Each MXU is capable of 16k Multiply-Accumulate (MAC) operation per cycle, which gives the bulk of the compute power. In contrast to GPUs which operate on vector data, TPUs operate on matrices providing hundreds of thousands of operations in a single clock cycle. MXU's architecture (Figure 3.8) is based on systolic arrays to implement such a large-scale matrix processor. Its systolic mechanism contains a matrix of 256x256 8-bit Arithmetic Logic Units (ALUs) that multiply-and-add 65536 integers per clock cycle, and because TPUs run at 700MHz, they yield performance of 92 TeraOPs per second each. The systolic array chains multiple ALUs to use the previous ALU's result, and reuse inputs by forwarding them to the next ALUs. In this manner, communication with off-chip memory is significantly reduced [67].

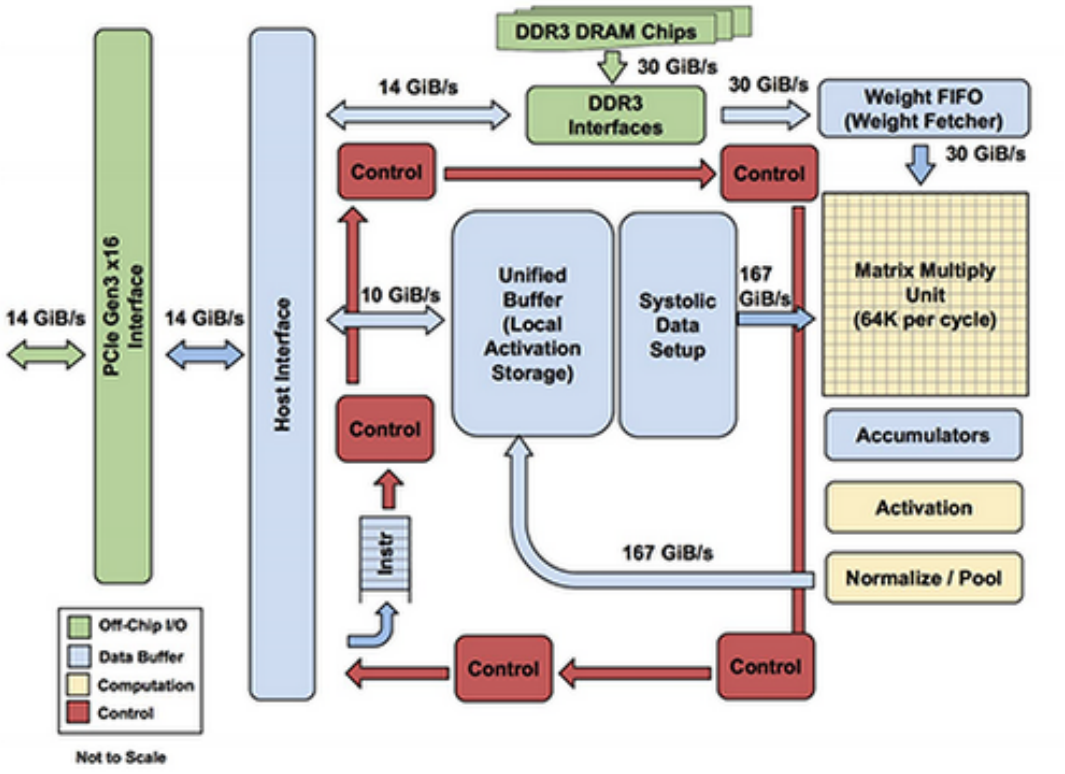

FIGURE 3.7: Google Cloud TPU v1 Architecture Block Diagram: URL

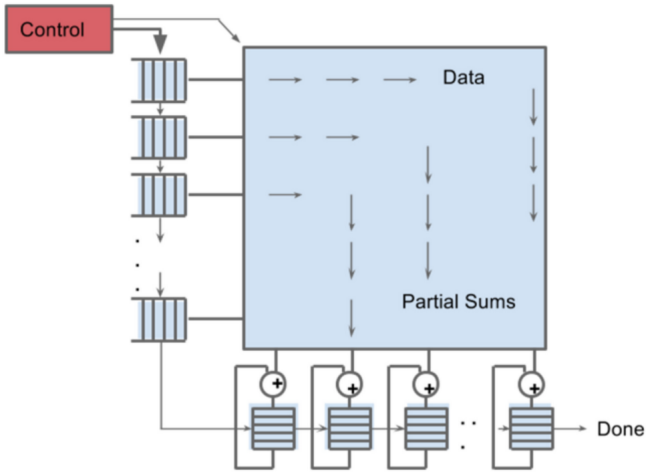

FIGURE 3.8: Google Cloud TPU Matrix Multiplier Unit (MXU): URL

Every TPU v3 board comprises four chips (Figure 1.3), with each chip containing 2 TPU cores. TPU boards are organized into pods (Figure 3.9), with each pod containing up to 2048 TPU cores and 32TB total memory [5], providing a total of up to 92 PetaFLOPS of performance [68].

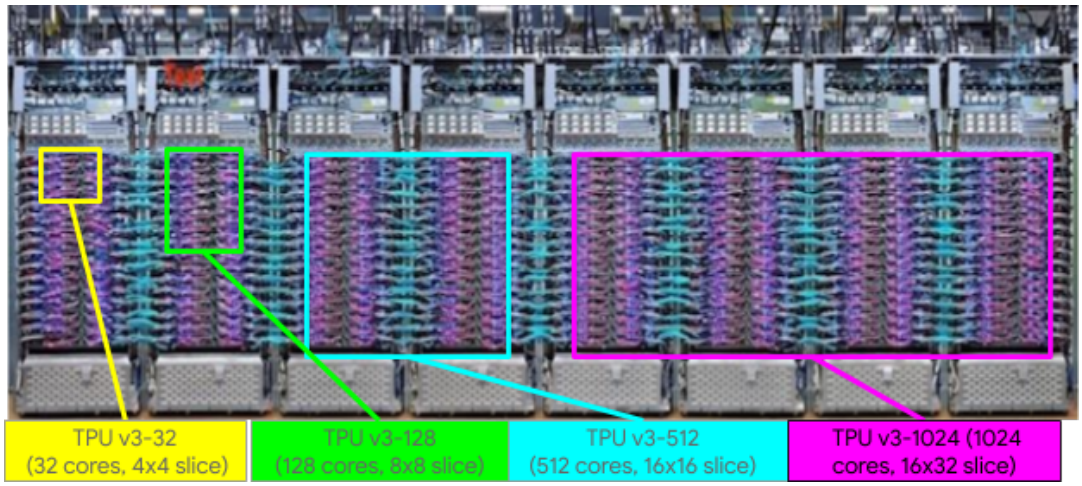

FIGURE 3.9: Google Cloud TPU v3 Pods: URL

The TensorFlow framework is used to run applications on TPUs, with a reduced bfloat16 precision. Google created the bfloat16 16-bit floating-point representation standard to provide better training and model accuracy than IEEE half-precision representation.

In general, TPUs are worth using with very large models, needing weeks or even months of training, dominated by matrix computations, and no custom TensorFlow operations inside the main training loop. Otherwise, GPUs or ever CPUs may be more suitable and cost-effective [69].

ASICs, like Google's TPUs, are very costly to design and produce. While they provide the best performance and efficiency, they can become deprecated fast, due to the speed the AI field is developing.

3.3.4 FPGAs

FPGAs provide high flexibility in hardware architectures, unlike all other hardware solutions that are limited to their fixed architecture. A custom design is developed for every application to provide the computation units in types and volume needed. Computation units are then placed on the so-called FPGA fabric, the programmable part of the chip, to create systems that accelerate specific applications.

Often FPGAs integrate numerous Digital Signal Processing (DSP) blocks, which are hardware elements into the FPGA fabric. DSPs provide optimized hardware for various common mathematical operations such as multiply-accumulate (MAC) and division for representations from floating-point to fixed-point or even integer. DSPs can also be configured as Double-Pumped, allowing them to be clocked at twice the core clock.

Moreover, FPGAs can come with hard processor cores in the same chip that can be used for communication, scheduling, and data pre and post-processing. Such configurations are typically called Multi-Processor System on Chip (MPSoC). FPGAs are a perfect match for fields that come with rapid development, such as AI and ML. Furthermore, they provide high energy efficiency and low latency per result due to the design being specific to the application.

Although FPGAs have Block RAM (BRAM) and some even Ultra RAM (URAM), a high-speed type of memory inside the FPGA fabric, they can be bottlenecked due to its limited size. Hence, FPGAs are often provided with big DRAM modules. However, compared to GPUs', which are provided with HBM, FPGAs may still face bandwidth limitations. Such limitations can be overcome the same way ASICs do; by reducing data precision. Reducing data precision is accomplished by converting them from floating-point to lower bitwidth numeric representations, such as half-precision floating point and fixed point. It can also be achieved by data reuse or quantization.

3.4 Quantization

In order to compete with GPUs, FPGAs and TPUs have to minimize their memory bandwidth needs. One of the most effective ways to tackle this problem is by reducing the parameters' memory footprint. This process is called quantization, and it consists of various techniques. While, quantization is applied independently of the framework that trains the network and the hardware that runs its inference, the techniques to be used are selected explicitly to the network's environment for best performance optimization.

3.5 The FPGA Perspective

While FPGAs can provide significant advantages over other hardware platforms, as of right now, there is no software framework, like TensorFlow and PyTorch, that natively supports FPGA acceleration. Hence, third party architectures have to be used.

3.5.1 Xilinx CHaiDNN

The CHaiDNN [70] is an open-source Deep Neural Network inference accelerator library developed by Xilinx for its Ultrascale+ MPSoC devices, mainly focusing on Convolutional Neural Networks. It was initially released in February 2018, with its second version been released in June 2018.

The CHaiDNN is designed for maximum compute efficiency at 6-bit fixed-point data type for both parameters and activations, while also supporting 8-bit fixed-point representations. The data precision can vary even throughout the layers, claiming that a well-crafted data precision combination can result in accuracy similar to a floating-point single precision model. To avoid retraining the network with 6-bit or 8-bit fixed-point representations, Xilinx has developed two quantization techniques, the Dynamic fixed-point quantization, and the Xilinx Quantizer, which are both supported by the CHaiDNN.

Moreover, the CHaiDNN provides configurations that can utilize from 128 up to 1024 Double-Pumped DSPs, with some designs achieving up to 700MHz. For smaller MPSoCs, there is the DietCHai, a miniature version of CHai. URAM is also supported.

Even though the most commonly used layers in various image classification and object detection neural networks are supported, unsupported layers can

also be added as software layers and run the network's inference. For performance optimization, some layers, such as the Fully-Connected and Softmax, are implemented as software layers. However, software layers can decrease the inference throughput based on their latencies. In order to tackle this problem, hardware and software layers can be run in parallel, hiding each others' latencies.

To run the inference using the accelerator, PetaLinux has to be running on the MPSoC to handle all the layer job scheduling across all CPU cores and hardware IPs. Network configuration and parameters are given using the Caffe framework's standard (.prototxt, .caffemodel, and mean files).

3.5.2 Xilinx Deep Learning Processing Unit (DPU)

Xilinx Deep Learning Processing Unit (DPU) [71] is a configurable Convolutional Neural Network 8-bit integer inference accelerator developed by Xilinx for its Zynq-7000 SoCs and Zynq Ultrascale+ MPSoCs. It supports most CNNs and can be configured appropriately according to parallelism needs and resource constraints. It was initially released in February 2019, with its latest version 3.2 been released in March 2020.

The DPU IP is implemented in the Programmable Logic (PL) part with direct connections to the Processing System (PS). Instructions are sent from the program running on the Application Processing Unit (APU) to the corresponding DPU core with memory addresses of input images and temporary and output data. The system's architecture is shown in Figure 3.10 (Left) with the APU, High Speed Data Tube, and DPU part all being in the same MPSoC, and RAM being the external Random Access Memory. A more detailed hardware architecture is shown on Figure 3.10 (Right). It is worth noting that the on-chip memory is utilized as a buffer for input, temporary and output data, increasing throughput, and efficiency by reusing as much data as possible and reducing external memory communication. The Hybrid Computing Array (Left) or Computing Engine (Right) is consisted of Processing Elements (PEs), creating a deep pipelined design. A PE is based on the fine-grained building blocks found in Xilinx devices, such as multipliers, adders, and accumulators.

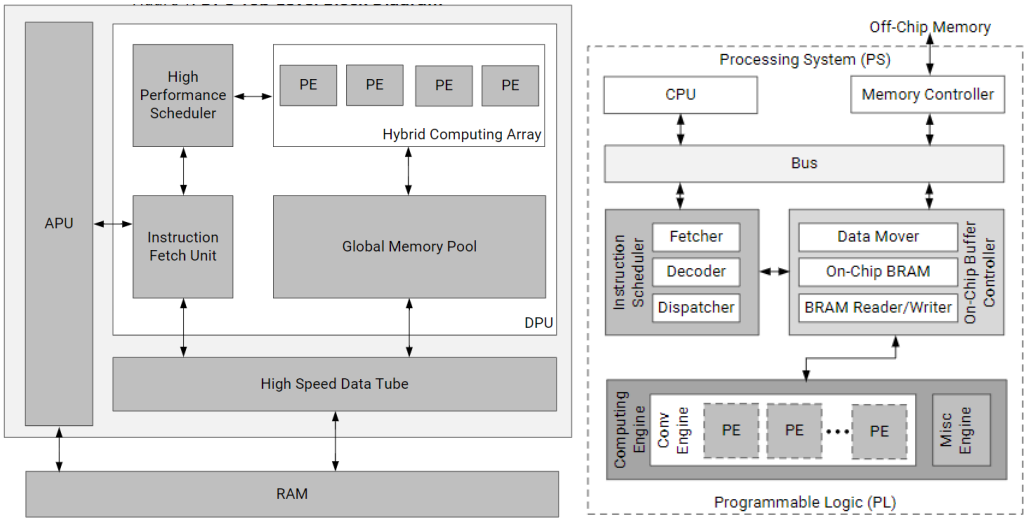

FIGURE 3.10: Xilinx DPU Architecture: Left top-level block diagram, Right hardware architecture: URL

DPU-V1, previously known as xDNN, uses a 96x16 DSP Systolic Array as its main compute unit (Figure 3.11). DPU-V2 uses a Hybrid Computing Array (Figure 3.12), and DPU-V3E uses multiple instances of Batch Engines in each DPU core (Figure 3.13).

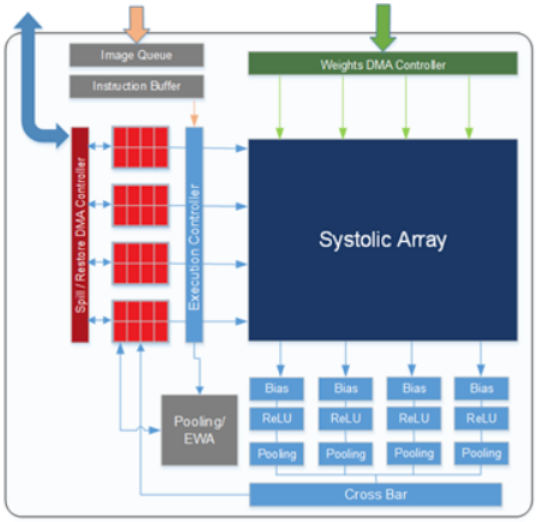

FIGURE 3.11: Xilinx DPU-V1 Architecture: URL

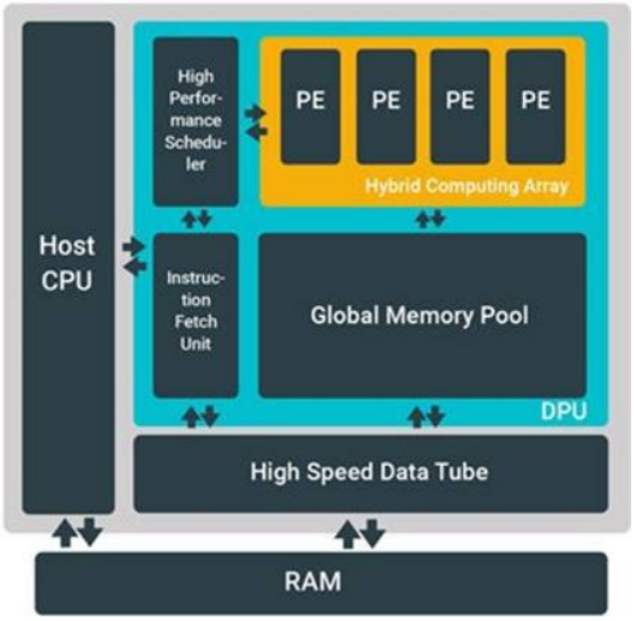

FIGURE 3.12: Xilinx DPU-V2 Architecture: URL

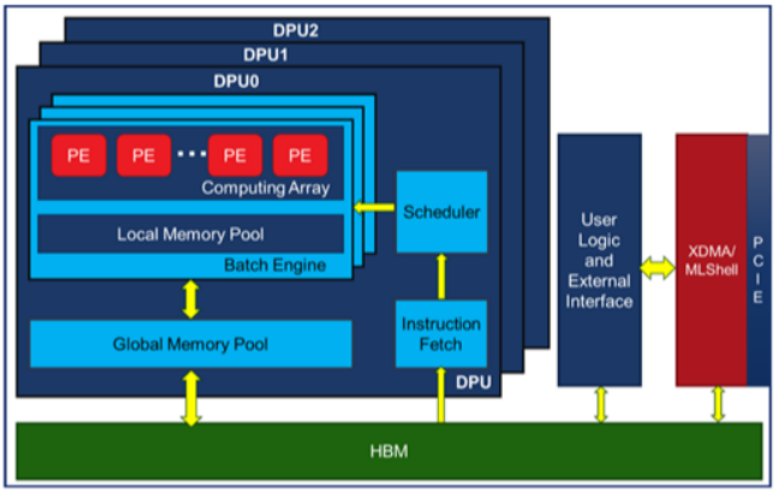

FIGURE 3.13: Xilinx DPU-V3 Architecture: URL

In contrast with the CHaiDNN, all layers, including the Fully-Connected and Softmax layers, are hardware accelerated. Currently, implementing up to four DPU cores in a single DPU IP is supported, and there is an option of using Double Data Rate DSPs (Double-Pumped DSPs). Furthermore, there are architectures for various parallelism needs, starting from 512 operations per cycle per core up to 4096 operations per cycle per core.

The Xilinx Vitis AI development environment [72], released in December 2019, is Xilinx's development platform for AI inference applications using its hardware platforms, which provides optimized IPs, tools, and libraries including the aforementioned Xilinx DPU. The Xilinx Vitis AI stack (Figure 3.14) has to be used to run a network's inference using the DPU accelerator. The network's program giving instructions and orchestrating the data transfers has to run on top of PetaLinux. This program is generated with substantial instruction optimizations using the Xilinx Vitis AI compiler. This program has to be regenerated with every network or hardware configuration change. The network's quantization is done using the Xilinx Vitis AI Quantizer, generating the appropriate 8-bit integer parameters.

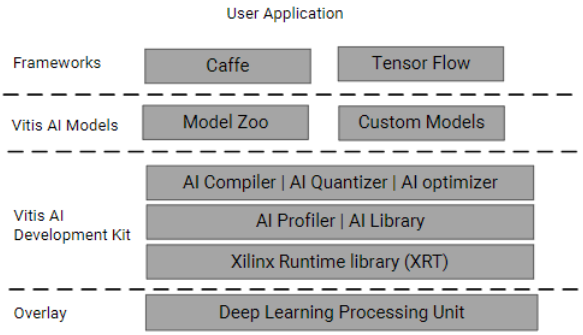

FIGURE 3.14: Xilinx Vitis AI Stack: URL

3.5.3 NVIDIA NVDLA

NVIDIA's Deep Learning Accelerator (NVDLA) [73], initially released in Q3 2017, is a free and open architecture project seeking to standardize the design of deep learning inference accelerators.

There are two main system implementations; the headless and the headed. The headless implementation expects the main system's processor to manage the accelerator, while the headed implementation expects a companion microcontroller, tightly coupled to the NVDLA sub-system, to do the management.

NVDLA comprises five components, Convolution core, Single Data processor, Planar Data processor, Channel Data processor, Dedicated Memory, and Data Reshape Engine, which are separate and independently configurable. Each block's input and output data transfers can be performed either in memory-to-memory using the Independent operation mode, or by passing through

some internal results to next blocks using the Fused operation mode (Figure 3.15). Furthermore, NVDLA supports a wide range of data types, from binary and 4-bit integer up to 64-bit floating-point.

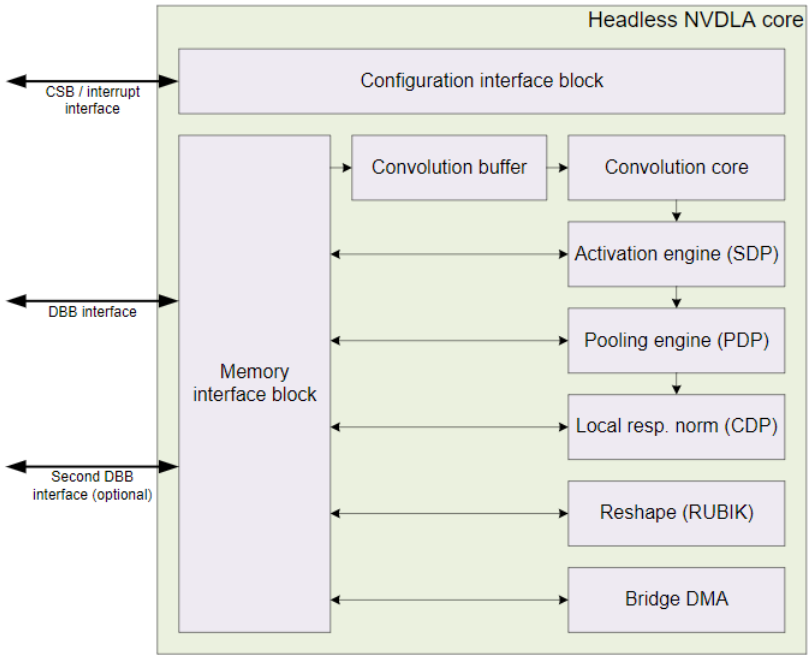

FIGURE 3.15: NVIDIA NVDLA Hardware Architecture in Fused operation mode: URL

The convolution core supports sparse weight compression to reduce memory bandwidth, as well as Winograd for computing efficiency for specific kernel sizes. Batch convolution is also supported to reduce memory bandwidth. In addition, a convolution buffer is added as internal RAM to avoid communication with external system memory.

The Single Data Point processor implements the various linear and non-linear activation functions used in CNNs. For the non-linear functions, a lookup table is used, while for the linear functions, a simple bias and scale are used.

The Planar Data processor supports minimum, maximum, and average pooling operations, with kernel sizes configurable during runtime.

The Cross-channel Data processor implements the local response normalization.

The Data Reshape engine performs data format transformations for operations such as slice and reshape-transpose.

In general, NVDLA is a highly customizable and modular architecture, with various configurations suitable for both FPGAs and ASICs.

3.6 Thesis Approach

This thesis aims to develop a hardware platform architecture targeted for FPGA devices, to run neural networks' inference, focusing on CNNs. The platform's goal is being flexible and versatile for easy design transferring, extendable to enable for easy adding of new layer types and new layer accelerators, able to run various CNNs, but most importantly, to provide easy experimentation and development of neural networks hardware accelerator architectures. Robustness analysis is carried out to investigate the FPGA's strengths and weaknesses, studying the computational workloads, memory access patterns, memory and bandwidth reduction, as well as algorithmic optimizations.

Chapter 4

Theoretical Modeling and Robustness Analysis

In this chapter, a model of CNNs structure, execution and characteristics is being created using MATLAB [74]. For a better understanding of the underlining algorithms, a C/C++ library was created to replicate PyTorch's [51] functionality, from image and parameter importing and formatting, to every CNN layer type needed for the network's inference. The library's results were evaluated using PyTorch's results. AlexNet [41] is used as a reference network for this analysis, with the parameters provided by the PyTorch's prebuilt and pre-trained model.

A Sensitivity Analysis was also performed to explore hardware implementation opportunities. Furthermore, various quantization techniques and algorithmic optimizations have been studied and tested to reduce memory footprint and better utilize hardware resources. Lastly, several implementation techniques and tools on the same functionality have been studied to minimize hardware resources and latency.

4.1 PyTorch and C/C++ implementations

Since PyTorch is a high abstraction framework, the recreation of its functionality to a lower abstraction level language, like C/C++, is a necessity for a complete understanding of the neural network's operation mechanism. Hence, after creating a C/C++ library capable of implementing most CNNs, it was evaluated for its correctness using PyTorch as a baseline. Its evaluation was conducted by implementing AlexNet as an example network, using both PyTorch and the C/C++ library, and then running the network's inference on

both implementations. A subset of the ImageNet's database was used as input images, provided by Kaggle [75], which included 1250 images of cats and another 1250 images of dogs.

Afterward, the results, taken from the last FC layer of both implementations, were compared to create an error rate for each input. The library is designed in a way that it can use any data type for weights and activations to enable further experimentations. This evaluation experiment was conducted two times, one for using double-precision and another for using single-precision floating-point data type (IEEE standard). The error rate of both experiments was almost zero, with some minor inconsistencies between the floating-point representations of C/C++ and Python. Thus, both implementations' classifications were fully matched, and the library can be characterized as correct.

4.1.1 Algorithms

The main building blocks of a typical 2-D CNN are presented below. Every algorithm in this section is part of this work's aforementioned C/C++ library.

Convolution

Algorithm 1 performs a 2-D Convolution on 3-D array inputs, like images. The *input* is characterized by its height, its width, and its number of channels. In other words, the number of its parallel matrices; for an RGB image, there are three channels, one for every color. The *kernelSize*, *stride* and *padding* are the hyper-parameters of convolution layers, and because they differ for every layer and network, they have to be included in the procedure's parameters. Last but not least, *weights* and *bias* are the networks parameters. This algorithm outputs a 3-D array with the convolution's results.

Algorithm 1 Convolution Layer

```
1: procedure CONVOLUTION LAYER(input, kernelSize, stride, padding, weights, bias)
2:     hOut ← (input.height + 2 * padding − kernelSize) / stride + 1
3:     wOut ← (input.width + 2 * padding − kernelSize) / stride + 1
4:     for i:=1 to input.channels do
5:         for j:=1 to input.height + 2 * padding do
6:             for k:=1 to input.width + 2 * padding do
7:                 if j <= padding || k <= padding || j > image.height + padding || k > image.width + padding then
8:                     arr(i, j, k) ← 0
9:                 else
10:                    arr(i, j, k) ← input(i, j − padding, k − padding)
11:    for oc:=1 to size(weights, 1) do                ▷ #Output channels
12:        for oh:=1 to hOut do
13:            imgStartH ← oh * stride
14:            imgEndH ← imgStartH + kernelSize
15:            for ow:=1 to wOut do
16:                imgStartW ← ow * stride
17:                imgEndW ← imgStartW + kernelSize
18:                pixel ← 0
19:                for ic:=1 to input.channels do
20:                    for i:=1 to kernelSize do
21:                        for j:=1 to kernelSize do
22:                            pixel ← pixel + arr(ic, i + imgStartH, j + imgStartW) * weights(oc, ic, i, j)
23:                output(oc, oh, ow) ← pixel + bias(oc)
24:    return output
```

Algorithm 2 is a slightly more optimized version of algorithm 1. Since padding creates areas of the input that are zeroed out, convolution on those areas results in zero. Therefore, the for-loops' indices are carefully calculated to avoid iterations on the padded areas. Furthermore, the creation of the "padded" input is also omitted.

Moreover, a small optimization is injecting the ReLU activation function into this algorithm. Provided that ReLU is used almost after every convolution layer, this optimization avoids rereading the whole input from memory to

make a simple decision. The procedure's *doRelu* parameter configures if the ReLU activation is used.

Algorithm 2 Convolution Layer with ReLU

1: **procedure** CONVOLUTION LAYER WITH RELU(input, kernelSize, stride, padding, weights, bias, doRelu)
2: $hOut \leftarrow (input.height + 2 * padding - kernelSize)/stride + 1$
3: $wOut \leftarrow (input.width + 2 * padding - kernelSize)/stride + 1$
4: **for** oc:=1 **to** size(weights, 1) **do** ▷ #Output channels
5: **for** oh:=1 **to** hOut **do**
6: $imgStartH \leftarrow oh * stride - padding$
7: $iStart \leftarrow imgStartH < 0?padding : 0$
8: **if** $imgStartH + kernelSize \geq input.height$ **then**
9: $iEnd \leftarrow kernelSize - (imgStartH + kernelSize - input.height)$
10: **else**
11: $iEnd \leftarrow kernelSize$
12: **for** ow:=1 **to** wOut **do**
13: $imgStartW \leftarrow ow * stride - padding$
14: $jStart \leftarrow imgStartW < 0?padding : 0$
15: **if** $imgStartW + kernelSize \geq input.height$ **then**
16: $jEnd \leftarrow kernelSize - (imgStartW + kernelSize - input.height)$
17: **else**
18: $jEnd \leftarrow kernelSize$
19: $pixel \leftarrow bias(oc)$
20: **for** ic:=1 **to** input.channels **do**
21: **for** i:=iStart **to** iEnd **do**
22: **for** j:=jStart **to** jEnd **do**
23: $pixel \leftarrow pixel + input(ic, i + imgStartH, j + imgStartW) * weights(oc, ic, i, j)$
24: $output(oc, oh, ow) \leftarrow doRelu\&\&pixel < 0?0 : pixel$
25: **return** $output$

MaxPool

Algorithm 3 performs a 2-D max-pooling operation on 3-D array inputs. Likewise to the convolution algorithm's *input*, it is characterized by its height,

width, and channel number. Also the *kernelSize* and *stride* hyper-parameters are present, in contrast to padding, which is not used, so it is omitted. This algorithm outputs a 3-D array with the max-pooling operation's results.

Algorithm 3 MaxPool Layer

```
1:  procedure MAXPOOL LAYER(input, kernelSize, stride)
2:      hOut ← (input.height − kernelSize)/stride + 1
3:      wOut ← (input.width − kernelSize)/stride + 1
4:      for i:=1 to input.channels do
5:          for j:=1 to hOut do
6:              for k:=1 to wOut do
7:                  max ← −∞
8:                  for l:=1 to kernelSize do
9:                      for m:=1 to kernelSize do
10:                         curHeight ← j ∗ stride + l
11:                         curWidth ← k ∗ stride + l
12:                         curPixel ← input(i, curHeight, curWidth)
13:                         if max < curPixel then
14:                             max ← curPixel
15:                 output(i, j, k) ← max
16:     return output
```

Fully-Connected

Algorithm 4 performs a matrix multiplication on the *input* and *weights*. The *input* is a vector, either coming from the previous FC layer's output or the flattening of the previous convolution or max-pooling layer's results. After the matrix multiplication a *bias* is accumulated. This algorithm outputs a vector with the results of the matrix multiplication.

Algorithm 4 Fully-Connected Layer

```
1:  procedure FULLY-CONNECTED LAYER(input, weights, bias)
2:      inputSize ← size(input)
3:      outputSize ← size(weights, 1)
4:      for i:=1 to outputSize do
5:          output(i) ← bias(i)
6:          for j:=1 to inputSize do
7:              output(i) ← output(i) + input(j) * weights(i, j)
8:          if doRelu && output(i) < 0 then
9:              output(i) ← 0
10:     return output
```

Algorithm 5 is a slightly more optimized version of algorithm 4. Likewise to the convolution algorithm 2, the ReLU activation is injected and can be configured through the *doReLU* parameter.

Algorithm 5 Fully-Connected Layer with ReLU

```
1:  procedure FULLY-CONNECTED LAYER WITH RELU(input, weights, bias,
    doReLU)
2:      inputSize ← size(input)
3:      outputSize ← size(weights, 1)
4:      for i:=1 to outputSize do
5:          output(i) ← bias(i)
6:          for j:=1 to inputSize do
7:              output(i) ← output(i) + input(j) * weights(i, j)
8:          if doRelu && output(i) < 0 then
9:              output(i) ← 0
10:     return output
```

ReLU

While ReLU activation function is already injected to the convolution 2 and FC 5 algorithms, its algorithms are shown below for completeness. Algorithm 6 performs the ReLU activation on vector inputs, suitable for use after FC layers. Algorithm 7 performs the ReLU activation on 3-D data inputs, suitable for use after convolution and max pooling layers.

Algorithm 6 ReLU (1-D)

```
1: procedure RELU (1-D)(input)
2:     for i:=1 to size(input) do
3:         if input(i) > 0 then
4:             output(i) ← input(i)
5:         else
6:             output(i) ← 0
7:     return output
```

Algorithm 7 ReLU (3-D)

```
1: procedure RELU (3-D)(input)
2:     for i:=1 to size(input, 1) do
3:         for j:=1 to size(input, 2) do
4:             for k:=1 to size(input, 3) do
5:                 if input(i, j, k) > 0 then
6:                     output(i, j, k) ← input(i, j, k)
7:                 else
8:                     output(i, j, k) ← 0
9:     return output
```

SoftMax

Algorithm 8 performs a SoftMax activation function on vector inputs. This operation is applied after the last FC layer and outputs the probability distribution of each target class.

Algorithm 8 SoftMax

```
1: procedure SOFTMAX(input)
2:     sum ← 0
3:     for i:=1 to size(input) do
4:         sum ← sum + e^input(i)
5:     for i:=1 to size(input) do
6:         output ← e^input(i) / sum
7:     return output
```

4.2 Memory Footprint

Memory footprint plays a considerable role in neural network applications based on FPGA or ASIC systems. While neural networks are compute-bound on classic hardware architectures, on FPGAs' and ASICs', they can become memory bound due to their high parallelism capabilities but low memory bandwidth. In general, an application can fully benefit from an FPGA when its memory requirements fit into the FPGA's internal BRAM. Otherwise, modern FPGAs support external DRAM modules that can be utilized to fulfill the application's memory requirements. However, this introduces latency and memory I/O stalls to the system.

Consequently, it is of high importance to minimize the network's memory footprint in order to make it fit into the internal BRAM, or at least, minimize the memory bandwidth requirements. It is firstly needed to explore how the network's memory requirements are distributed throughout its stages. Using AlexNet as a reference network, tables 4.1 and 4.2 show AlexNet's parameters memory requirements per layer and each layer's output memory requirements respectively, using single-precision floating-point numbers.

TABLE 4.1: AlexNet Parameters Memory Footprint

Layer	**#Parameters**	**Footprint**	**Memory (%)**
Conv1	$64 * 3 * 11 * 11 = 23232$	92.92KB	0.04
Conv2	$192 * 64 * 5 * 5 = 307200$	1.22MB	0.5
Conv3	$384 * 192 * 3 * 3 = 663552$	2.65MB	1.09
Conv4	$256 * 384 * 3 * 3 = 884736$	3.53MB	1.45
Conv5	$256 * 256 * 3 * 3 = 589824$	2.35MB	0.97
FC1	$9216 * 4096 = 37748736$	150.99MB	61.79
FC2	$4096 * 4096 = 16777216$	67.10MB	27.46
FC3	$4096 * 1000 = 4096000$	16.38MB	6.70
Total	61090496	244.36MB	100

This work focuses on the Xilinx ZCU102 [76] [77] and the QFDB [6] target boards, which both integrate the same Zynq UltraScale+ MPSoC XCZU9EG-2FFVB1156E. Unfortunately, this MPSoC provides only 2MBs of BRAM, which, according to tables 4.1 and 4.2, creates some serious memory constraints.

In respect to table 4.1, only the first and the second convolutional layers' parameters can fit both simultaneously in the MPSoC's BRAM. Moreover, not only do all other layers not fit, but they also need up to 75 times more memory each (see first Fully-Connected layer).

TABLE 4.2: AlexNet Data Stages Memory Footprint.

Layer	**#Data**	**Footprint**	**Memory (%)**
Image	$3 * 224 * 224 = 150528$	150.52KB	6.07
Conv1	$64 * 55 * 55 = 193600$	774.40KB	31.22
MaxPool1	$64 * 27 * 27 = 46656$	186.62KB	7.52
Conv2	$192 * 27 * 27 = 139968$	559.87KB	22.57
MaxPool2	$192 * 13 * 13 = 32448$	129.79KB	5.23
Conv3	$384 * 13 * 13 = 64896$	259.58KB	10.46
Conv4	$256 * 13 * 13 = 43264$	173.05KB	6.98
Conv5	$256 * 13 * 13 = 43264$	173.05KB	6.98
MaxPool3	9216	36.86KB	1.49
FC1	4096	16.38KB	0.66
FC2	4096	16.38KB	0.66
FC3	1000	4KB	0.16
Total	682856	2.48MB	100

In respect to table 4.2, all layers' outputs can fit individually into the MPSoC's BRAM; however, they cannot fit all together combined.

Under those circumstances, the exploration of various techniques for memory footprint reduction, memory footprint partitioning, and caching is an essential requirement.

4.3 Data Types

As mentioned in the Related Work section, memory bandwidth needs can become a severe bottleneck to applications like neural networks on FPGAs and ASICs. Consequently, the investigation of the most suitable data type for parameters and activations is a necessity. However, decreasing the bit-width does come with its costs, classification accuracy. This creates a trade-off between network classification accuracy and inference speed. In general,

lowering the bit-width increases performance due to both decreasing memory bandwidth and computation time, but decreases accuracy. The cost of each bit-width lowering is different for every application, so an application-specific investigation has to be conducted.

4.3.1 Evaluation

Using AlexNet as a reference, various data types have been tested to achieve the best performance to accuracy ratio. From double, single, and half-precision floating-point (IEEE standard) representations to fixed-point from 64 down to 8-bit representations have been tested. This experiment's baseline uses a single-precision floating-point, as it is the pre-trained model's data type. After inferencing 2500 images using every data type, each image classification Top-1 result was compared to the baseline's result. The Top-1 error rates for each data type are presented in tables 4.3 and 4.4.

4.3.2 Floating Point

It is worth noting that a MATLAB implementation of the network had to be used in this investigation, due to the fact that, at the time of writing, neither PyTorch nor C/C++ support half-precision floating-point arithmetic. Unfortunately, MATLAB inference was really slow compared to PyTorch, and even slower when using half-precision (see table 4.3 Avg. inference time column). The half-precision floating-point slowdown was caused because there is no hardware acceleration on the test CPU (Intel i7 4710MQ [78]) for such data type, forcing the mathematical operations to be calculated using software emulation.

Before every test, the network's parameters were converted from the given single-precision floating-point pre-trained model to the corresponding data type. Also, the same data type was used for the layers' activations during inference. While converting single-precision to double-precision does not result in higher arithmetic accuracy, experiments with doubles were conducted for completeness, as shown in table 4.3.

TABLE 4.3: Top-1 error rate of every floating-point tested data type.

Tool	Data type	Top-1 Error rate (%)	Avg. inference time (sec)
PyTorch	float64	0	0.091
PyTorch	float32	0	0.034
MATLAB	float64	0	6.624
MATLAB	float32	0	8.162
MATLAB	float16	0.36	147.480

In respect to the floating-point data types, the Top-1 error-rate for both double and single-precision is zero, while for the half-precision, there is a minor error rate that can be ignored. As a result, the half-precision is the leading one in the floating-point space due to its excellent performance to accuracy ratio.

4.3.3 Fixed Point

The usage of fixed-point data types can benefit the application with its lower memory bandwidth and its more straightforward hardware requirements and faster arithmetic operations compared to the floating-point data types. In contrast to the floating-point data types conversion, when converting to fixed-point data types, it is vital to select the position of the radix point that most accurately represents the given numbers. In this experiment, the uniform conversion technique was used in which given a set of numbers S and the wanted representation bit width, W, the radix point is selected as shown on equation 4.1 by minimizing the overall error between the given number set S and the converted to fixed-point number set.

$$Position = argmin_{i=0}^{W}[\frac{\sum_{j=1}^{size(S)} |S_j - FixPtConvert(S_j, W, i)|}{size(S)}] \quad (4.1)$$

In this experiment, a different radix point position was calculated for every layer's parameters and every bit width with respect to each layer parameter's characteristics. This technique was first introduced by Williamson in 1991 [79], and is called Dynamic Fixed-Point arithmetic. Instead of using a global scaling factor for all layers (ordinary fixed-point arithmetic), dynamic fixed-point arithmetic uses a different scaling factor for every layer. On low

bit-widths, using dynamic fixed-point because it can maintain most of the weights representation accuracy.

It should be mentioned that MATLAB's conversion of floating-point to any bit-width fixed-point data types results in a set of double-precision floating-point numbers to enable mathematical operations without needing fixed-point hardware acceleration or software emulation. Consequently, each layer's activations had to be converted to fixed-point so that the system's modeling was as accurate as possible.

Unfortunately, only the 64 and 32-bit tests resulted in no Top-1 error-rate, as shown in table 4.4. Even the 16-bit test resulted in a very high error-rate, rendering this data type unusable for this application. However, the high error-rate can be explained after visualizing the weights' distributions before and after the conversion, with its three side effects. The following figures were selected because of their dramatic transformation caused by this conversion to demonstrate its side effects.

TABLE 4.4: Top-1 error rate of every fixed-point tested data type.

Tool	**Data type**	**Top-1 Error rate (%)**	**Avg. inference time (sec)**
MATLAB	fixed64	0	7.318
MATLAB	fixed32	0	7.692
MATLAB	fixed16	22	6.650
MATLAB	fixed14	28.44	6.813
MATLAB	fixed12	36.24	6.797
MATLAB	fixed10	77.07	6.929
MATLAB	fixed8	100	6.312

First of all, figure 4.1 depicts the second convolution layer's weights distributions of both their original single-precision floating-point and their 8-bit fixed-point converted representations. It is evident that the right histogram's limits have been significantly altered, with the weights $W\epsilon(-\infty, -0.5) \cup (0.5, \infty)$ been suppressed to the $(-0.5, 0.5)$ range.

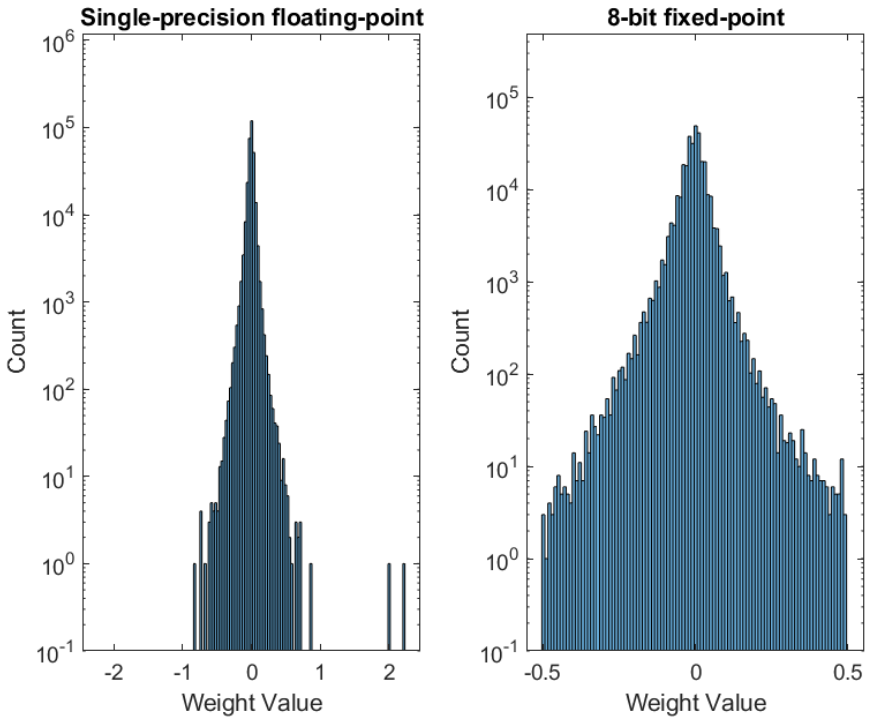

FIGURE 4.1: Second Convolution layer's weights distribution comparison between single-precision floating-point and 8-bit fixed-point representations: right histogram's limits are significantly altered.

This suppression is caused by the nature of the 8-bit fixed-point data type, which only allows 256 different values to be represented. Equation's 4.1 result is position 8, which means that the representation's step is $1/2^8 = 0.00390625$, hence its limits are $-2^7/2^8 = -0.5$ and $2^7/2^8 = 0.5$ (the most significant bit is used for the sign). Although there are some weights in the range of $(-2, -0.5) \cup (0.5, 2)$, their summed conversion accuracy error is less than the sum of accuracy errors of the smaller numbers, which are orders of magnitude more in count, when another radix-point position is selected. In other words, the sum of a lot of small errors is greater than the sum of a few big errors, causing the big errors the be ignored.

Second of all, figure 4.2 depicts the fifth convolution layer's weights distributions of both their original single-precision floating-point and their 8-bit fixed-point converted representations. While this layer's weights distribution limits are not altered, various spikes on the right histogram can be observed.

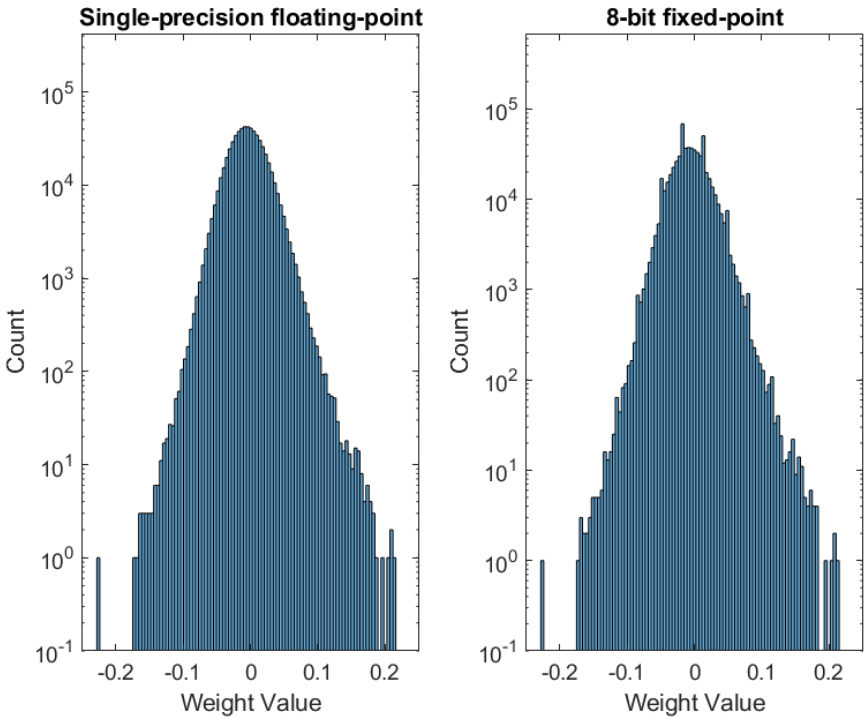

FIGURE 4.2: Fifth Convolution layer's weights distribution comparison between single-precision floating-point and 8-bit fixed-point representations: various spikes can be observed on the right histogram.

Those spikes are caused by the fixed-point's inability to represent all real numbers. It is the same reason the floating-point data type exists. The fifth convolution layer's radix-point is positioned on the 8th bit, which results in a representation step of $1/2^8 = 0.00390625$. This means that any weight with value $(i/8, (i+1)/8)$, with $i = -2^7, -2^7 + 1, ..., 2^7$, is rounded to the nearest fixed-point number available. Therefore, many weights are rounded to their nearest fixed-point representation, creating those spikes, which is expected.

Last but not least, figure 4.3 depicts the first Fully-Connected layer's weights distributions of both their original single-precision floating-point and their 8-bit fixed-point converted representations. In this figure's right histogram, a high amount of subsampling can be observed compared to the left one.

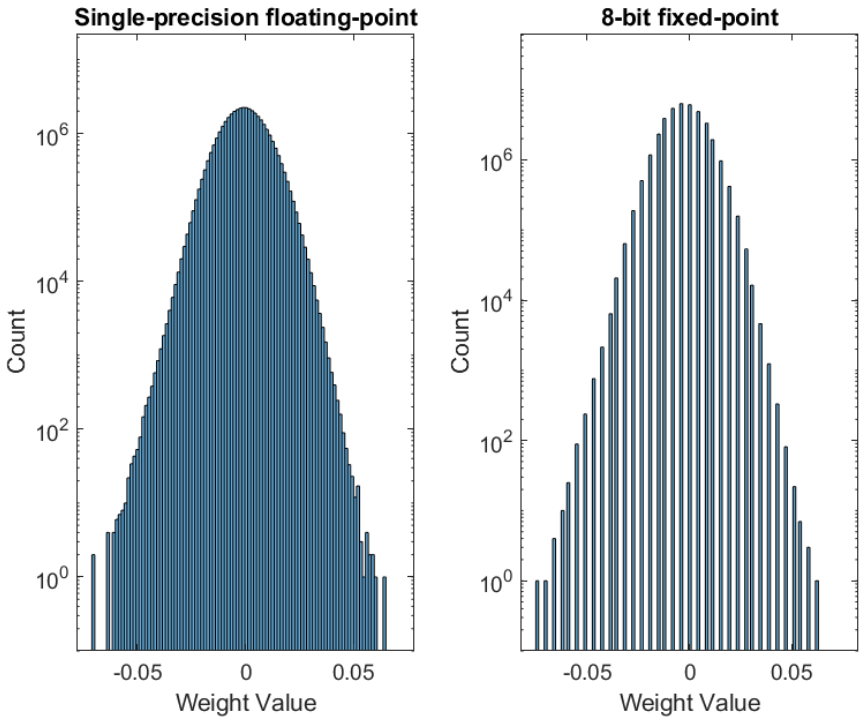

FIGURE 4.3: First Fully-Connected layer's weights distribution comparison between single-precision floating-point and 8-bit fixed-point representations: significant subsampling on the right histogram.

Again, this subsampling can be explained by the fixed-point's inability to represent all weights accurately, resulting in jumps. In this layer, equation 4.1 positioned the radix-point on the 8th bit, which, again, means a step of $1/2^8 = 0.00390625$.

To tackle with the suppression and subsampling side effects, an enhanced version of this technique was used. More specifically, instead of merely measuring the representation error of each weight and the summing it up with all other errors, as equation 4.1 does, it is essential to further amplify significant errors in order for them to stand out compared to the big count of small errors. This amplification can be done by adding non-linearity to the error's computation. The Mean Squared Error (MSE) can be used for this purpose, as shown in equation 4.2.

After observing the weights' distributions, it is clear that some distributions need a more accurate representation step. This is given by the radix point's position. So, instead of searching for a position on the given bit width, the

radix-point should be allowed to exceed the number's bit-width. Therefore, the maximum radix-point was selected greedily to be 32.

$$Position = argmin_{i=0}^{32}[\frac{\sum_{j=1}^{size(S)} |S_j - FixPtConvert(S_j, W, i)|^2}{size(S)}] \tag{4.2}$$

Although equation 4.2 resulted in better inference accuracy, after some experimentation, more amplification was needed to achieve the best accuracy possible with this technique. For AlexNet, the Mean Quarted Error (MQE), shown on equation 4.3, yields the best results, and no further amplification is needed.

$$Position = argmin_{i=0}^{32}[\frac{\sum_{j=1}^{size(S)} |S_j - FixPtConvert(S_j, W, i)|^4}{size(S)}] \tag{4.3}$$

As shown on figures 4.4, 4.5 and 4.6, the suppression and the subsampling side effects are eliminated. There is still some spiking; however, it is expected behavior due to the lack of representation accuracy with fixed-point data types.

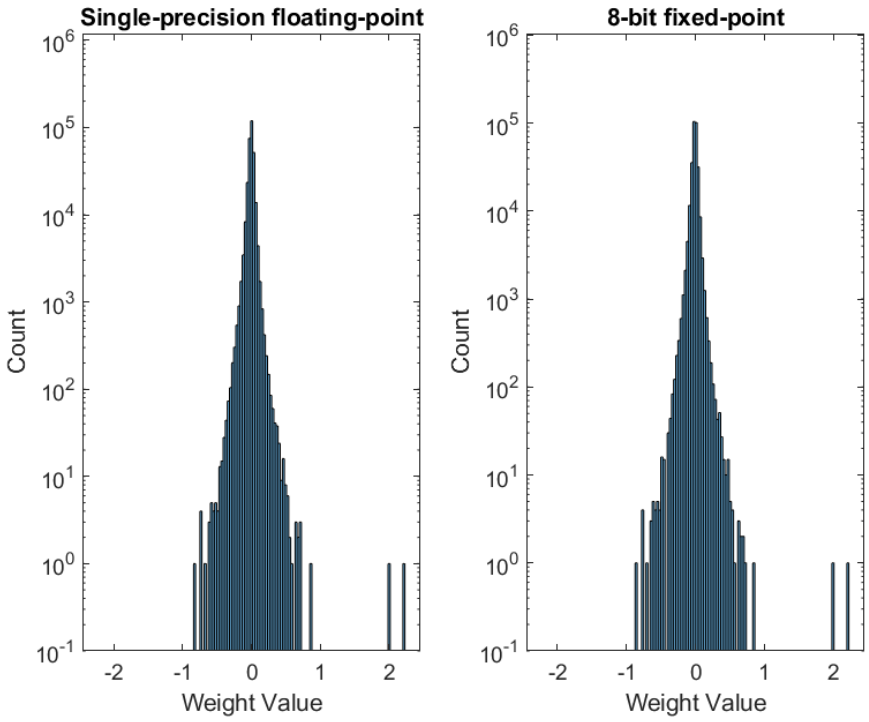

FIGURE 4.4: Second Convolution layer's weights distribution comparison between single-precision floating-point and 8-bit fixed-point representations using MQE: right histogram's limits are identical.

Using MQE, equation 4.3 resulted in the 5th radix-point position for the second convolution layer's weights. This results in a representation step of $1/2^5 = 0.03125$, hence its limits are $-2^7/2^5 = -4$ and $2^7/2^5 = 4$. Consequently, the whole weights range can be represented, removing the suppression problem.

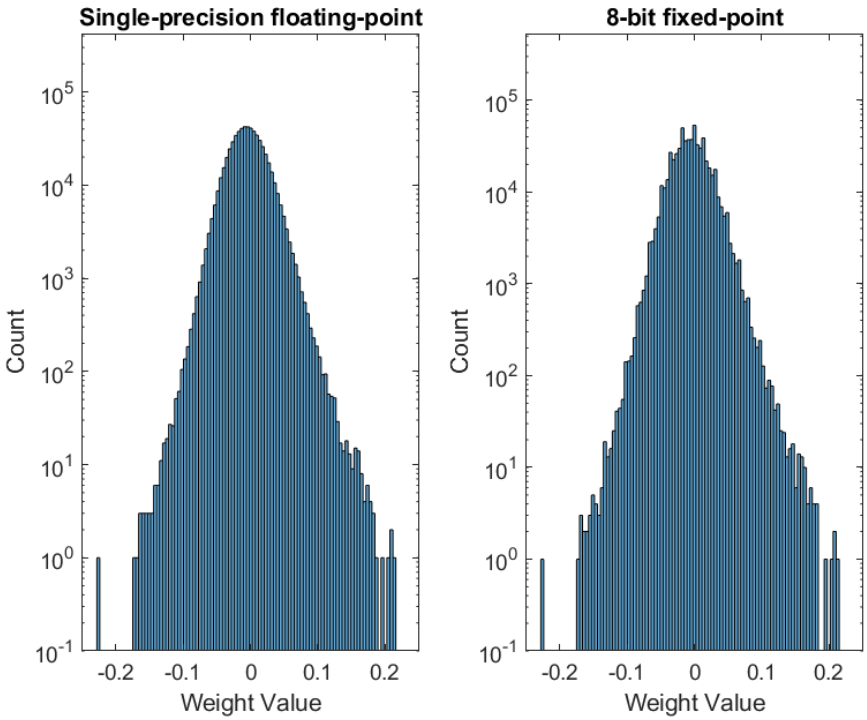

FIGURE 4.5: Fifth Convolution layer's weights distribution comparison between single-precision floating-point and 8-bit fixed-point representations using MQE: various spikes can still be observed on the right histogram.

Using MQE, equation 4.3 resulted in the 9th radix-point position for the fifth convolution layer's weights. This results in a representation step of $1/2^9 = 0.001953125$, hence its limits are $-2^7/2^9 = -0.25$ and $2^7/2^9 = 0.25$. Although, the spikes are not removed, there is a slight improvement in the weights representation accuracy.

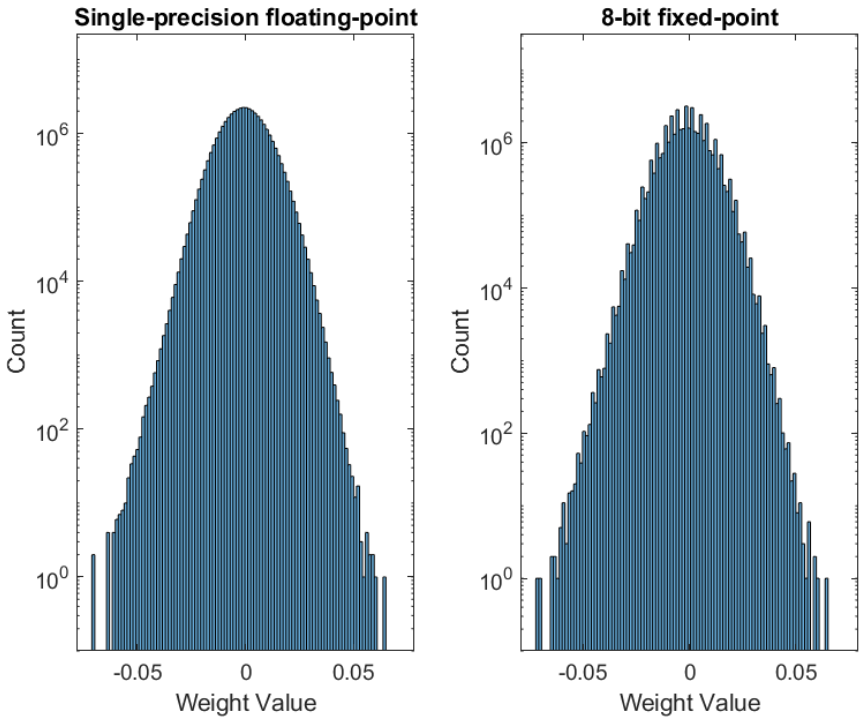

FIGURE 4.6: First Fully-Connected layer's weights distribution comparison between single-precision floating-point and 8-bit fixed-point representations using MQE: no subsampling on the right histogram.

Using MQE, equation 4.3 resulted in the 10th radix-point position for the first Fully-Connected layer's weights. This results in a representation step of $1/2^10 = 0.0009765625$, hence its limits are $-2^7/2^10 = -0.125$ and $2^7/2^10 = 0.125$. The added representation accuracy fully eliminates the subsampling problem, though some spiking is introduced as expected.

Table 4.5 shows the Top-1 error-rate of fixed-point data types using the Mean Quarted Error equation 4.3. While 8, 10, and 12 bit width fixed-point data types are still unusable, the 14 and 16 bit width representations have improved significantly.

TABLE 4.5: Top-1 error rate of fixed-point data types using Mean Quarted Error equation 4.3.

Tool	Data type	Top-1 Error rate (%)
MATLAB	fixed64	0
MATLAB	fixed32	0
MATLAB	fixed16	4.42
MATLAB	fixed14	17.59
MATLAB	fixed12	48.11
MATLAB	fixed10	86.91
MATLAB	fixed8	99.3

For better clarity on the results, figure 4.7 depicts all tested data types and their Top-1 accuracy.

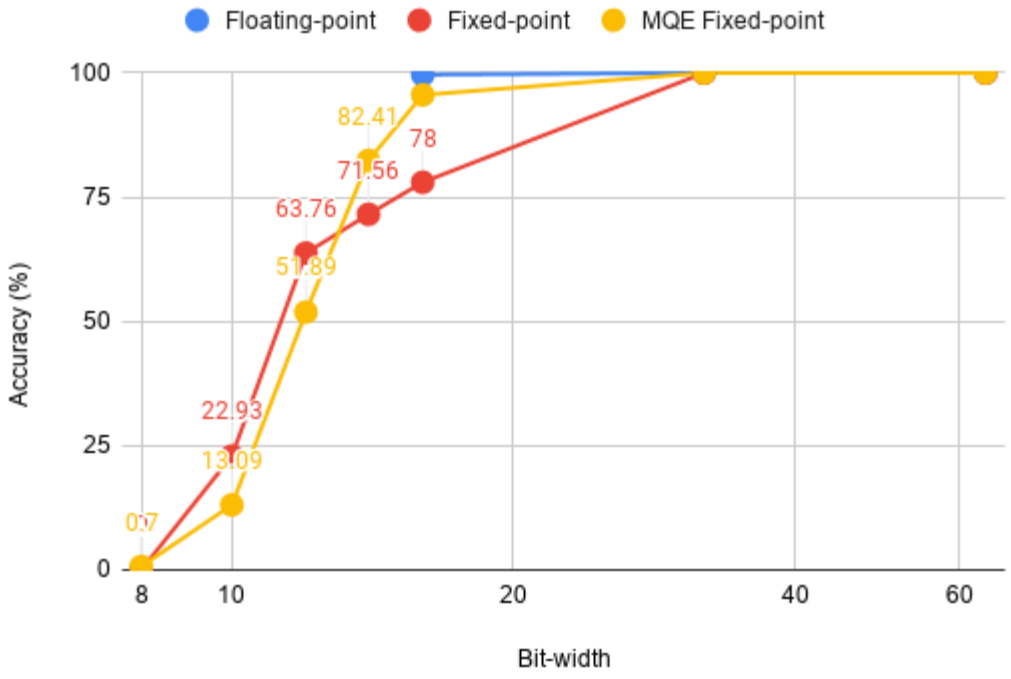

FIGURE 4.7: All data types tested accuracy.

After some experimentation, dynamically assigning different bit-widths on each layer, according to its accuracy needs, rather than a single bit-width for the whole network, can yield even better results. However, further investigation has to be conducted.

4.3.4 Fixed Point Activations

Since floating-point arithmetic handles scaling automatically, the network does not face any overflow defects on its activations. However, when using fixed-point arithmetic, activations overflow is a severe problem. For instance, when multiplying 8-bit fixed point activations and weights, a new activation is generated with at most 16-bit width. Furthermore, adding two 16-bit activations can generate a new activation of 17-bit width. Therefore, if activations are not quantized between each layer, there will be a very wide output at the end of the network, wasting resources and performance when implemented in FPGAs.

Hence, between every layer, the activations should be quantized from their high-detail, large bit-width representations, back to some low bit wide representation. A question arises on how to quantize them in order to keep the most detail available. This work tries to keep the upper n bits from the first one. For example, let an activation implementation be 32-bit wide, its most significant one be on the 17th bit, and the new, low-detail representation be 8-bit wide. Then, if the upper 8 bits are kept, from the 31st bit down to the 24th bit, they will all be zeros, and as a result, its value will be lost. However, if the 8 bits from the first upper one are kept, from the 17th bit down to the 10th bit, the most significant detail is retained.

On the contrary, finding the most significant one on every activation in a single layer is not only computationally intensive but also requires that every activation has its own scale factor, rendering the fixed-point's benefits obsolete. All activations in a single layer should have the same scale factor, simplifying their arithmetic operations and minimizing required I/O.

Hence, an experiment was conducted to find each layer's optimal activation scale factor, whose results are depicted in table 4.6, using AlexNet as a reference. The theoretical bit-width represents the maximum bits needed to represent the layer's output adequately and can be calculated using equation 4.4.

$$Theoretical_{bitWidth} = input_{bitWidth} + weight_{bitWidth} + \lceil \log_2 \#Additions \rceil \quad (4.4)$$

However, equation 4.4 calculates the maximum bit-width that could be generated after all operations, when all numbers are of max value, which is not

the case. Therefore, the experiment includes a practical study of the maximum incurring bit-width, conducted by inferencing 2000 images and finding the maximum valued activation on each layer. Results of this experiment are depicted on column Practical bit-width of table 4.6.

TABLE 4.6: Theoretical and Practical activations' bit-widths

Layer	**Theoretical bit-width**	**Practical bit-width**
Input	8	8
Conv1	$8+8+\lceil \log_2 3*11*11 \rceil = 25$	17
Conv2	$8+8+\lceil \log_2 64*5*5 \rceil = 27$	14
Conv3	$8+8+\lceil \log_2 192*3*3 \rceil = 27$	15
Conv4	$8+8+\lceil \log_2 384*3*3 \rceil = 28$	15
Conv5	$8+8+\lceil \log_2 256*3*3 \rceil = 28$	17
FC1	$8+8+\lceil \log_2 9216 \rceil = 30$	17
FC2	$8+8+\lceil \log_2 4096 \rceil = 28$	17
FC3	$8+8+\lceil \log_2 4096 \rceil = 28$	17

As of table 4.6, the theoretical and practical bit-widths appear significant differences. Fortunately, the maximum theoretical bit-width is 30 bits, and consequently, all layers' activations can fit in 32-bit integer representations, before quantizing them.

The scale factor after the quantization of the layer's activations can be calculated using equation 4.5, where $Practical_{bitWidth}$ is the activations' maximum practical bit-width as shown in table 4.6 and $RepBits$ is the bit-width of the representation after the quantization.

$$ScaleFactor = Practical_{bitWidth} - (\#RepBits - 1) + input_{scaleFactor} + weight_{scaleFactor} \tag{4.5}$$

Table 4.7 depicts the optimal scale factor for each layer's weights after their conversion from floating-point to 8-bit fixed-point as instructed by section 4.3.3, and the scale factor of each layer's outputs.

TABLE 4.7: Scale Factor per layer

Layer	Weights	Bias	Output
Input	-7	-	-
Conv1	-7	-5	-2
Conv2	-5	-7	0
Conv3	-7	-7	3
Conv4	-8	-6	5
Conv5	-9	-5	10
FC1	-10	-10	15
FC2	-10	-9	19
FC3	-9	-9	23

4.4 Weight Pruning

Synaptic pruning is the biological brain's phase [80], during which both axons and dendrites of mammals brains decay and die off, typically occurring from the time of birth until the mid-20s on humans. Inspired by the Synaptic Pruning, Weight Pruning is a technique for artificial neural network compression, leading to smaller in size networks, higher memory and power efficiency, and faster inference times.

During the weight pruning procedure, the weights that contribute little or even nothing to the network's knowledge are getting pruned, or in other words, are getting zeroed out. This zeroing means that any activation multiplied by a zero weight also results in a zero. Hence, hardware accelerators can ignore zero weights and skip calculations, speeding up the overall inference procedure. In addition, more zeros also means higher compression rates of a layer's weights, and consequently, lower memory bandwidth requirements. However, similar to data type selection, weight pruning creates a tradeoff between inference performance and classification accuracy.

Also similar to the data type selection, the best weight pruning amount varies according to the network in question. Therefore, a specific investigation has to be conducted per network. For this work's reference network, AlexNet, a specific investigation has been conducted, whose results are shown on table 4.8 and figure 4.9, the process of which is described below.

The weight pruning process requires a pruning factor f, which will zero out every weight $w\epsilon[-f, f]$. Its selection is essential as it controls the pruning amount and the network's post-pruning accuracy. If the factor is too big, there might be a significant impact on accuracy, while if the factor is too small, then the compression might be ineffective.

A strategy for selecting the best pruning factor might be to use the same factor on all layers and measure the post-pruning accuracy. However, such a strategy leads to low accuracy on AlexNet. Since each AlexNet's layer weights distribution differs in variance, significant weights are valued differently. Figure 4.8 shows each AlexNet's layer weights distribution, making it clear that a global pruning factor cannot be used for high-performance rates; every layer has different limits on its weight distribution and different concentrations per weight value range.

A more suitable strategy would be to investigate a pruning factor for each layer. Starting from the first layer, prune its weights by some factor f_1, test the network for its accuracy. If accuracy results are not acceptable, repeat this step by selecting another value for f_1. If the accuracy is acceptable, continue to the second layer by selecting a new factor f_2. Test the network for classification accuracy with only the second layer's weight pruned by f_2. This process continues until the last layer, which in this work's network, AlexNet, is the third Fully-Connected layer. Finally, there will be the array $F = [f_1, f_2, ..., f_8]$ (for AlexNet), which includes all pruning factors, one per layer.

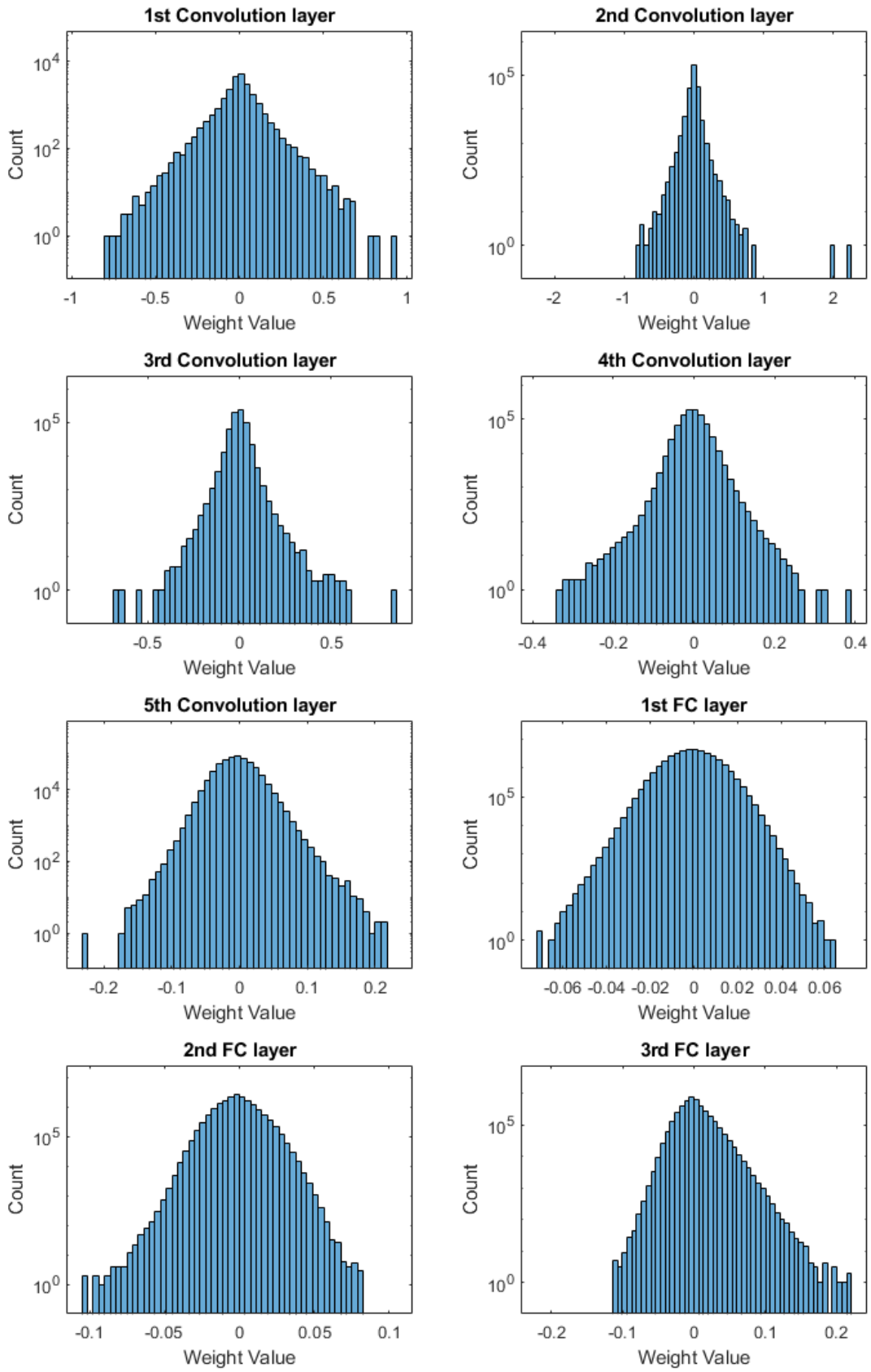

FIGURE 4.8: All layer weights distributions.

Table 4.8 shows the weight pruning percentage per layer and all-layer total, with the classification accuracy yielded by each test. Seven tests are depicted, with different configurations each. Tests 1-4 use pruning on all layers, while tests 5-7 use pruning only on the Fully-Connected layers due to their enormous size (see table 4.1).

Tests explanation:

- **Test 1:** Low pruning amount on convolution layers, low pruning amount on Fully-Connected layers.
- **Test 2:** Medium pruning amount on convolution layers, low pruning amount on Fully-Connected layers.
- **Test 3:** High pruning amount on convolution layers, low pruning amount on Fully-Connected layers.
- **Test 4:** High pruning amount on convolution layers, high pruning amount on Fully-Connected layers.
- **Test 5:** No pruning on convolution layers, low pruning amount on Fully-Connected layers.
- **Test 6:** No pruning on convolution layers, medium pruning amount on Fully-Connected layers.
- **Test 7:** No pruning on convolution layers, high pruning amount on Fully-Connected layers.

TABLE 4.8: All pruning amount configurations tested and their accuracy.

Layer	**Test 1**	**Test 2**	**Test 3**	**Test 4**	**Test 5**	**Test 6**	**Test 7**
Conv1 (%)	7.15	13.66	91.3	91.3	0	0	0
Conv2 (%)	13.82	26.9	95.83	95.83	0	0	0
Conv3 (%)	13.54	26.63	98.62	98.62	0	0	0
Conv4 (%)	15.32	29.99	93.14	93.14	0	0	0
Conv5 (%)	15.55	30.53	94.02	94.02	0	0	0
FC1 (%)	41.23	41.23	41.23	94.48	41.23	71.89	96.61
FC2 (%)	36.69	36.69	36.69	90.61	36.69	62.52	90.61
FC3 (%)	27.27	27.27	27.27	89.68	27.27	47.74	75.56
Total (%)	37.97	38.54	41.22	93.11	37.38	64.79	89.65
Accuracy (%)	91.74	80.8	0	0	90.87	71.77	15.06

From table 4.8, it is evident that increasing the pruning amount leads to lower accuracy. Test 1, which has the lowest pruning amount overall, yielded the best results. Similarly, concerning the tests 5-7, where only the Fully-Connected layers have been pruned, the test with the least amount of pruning, Test 5, resulted in the best accuracy. From the tests 1-4, one could argue that the convolution layers are more prone to error when increasing their pruning amount compared to Fully-Connected layers.

Tests 1 and 5 have identical pruning amount on the Fully-Connected layers; however, test 1 also uses pruning on convolution layers. While test 5 uses less pruning overall, it yields slightly lower accuracy compared to test 1. This could be explained because of the denoising characteristics that the weight pruning provides to the systems. Low and non-important weights could be characterized as noise, originating from the training procedure. Hence, cutting off the "noisy" weights could lead to higher accuracy, which is the case with tests 1 and 5.

Figure 4.9 shows the two test groups' classification accuracy to total pruning amount. The first test group, tests 1-4, depicted with blue, is the one that all layers are getting pruned, while the second test group, test 5-7, depicted with red, is the one that only the Fully-Connected layers are getting pruned.

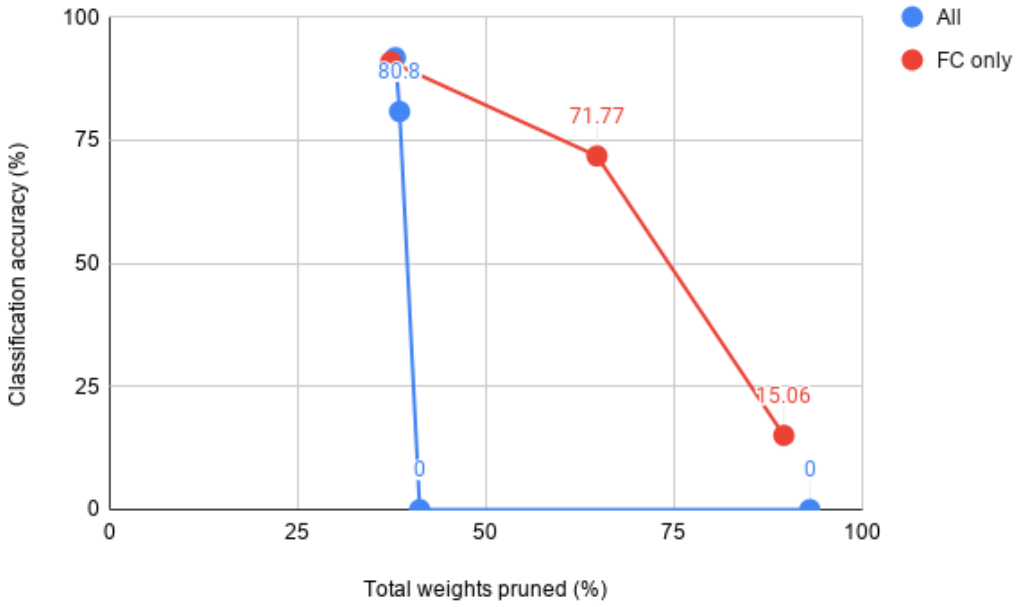

FIGURE 4.9: Accuracy per pruning amount.

It is obvious that pruning only the Fully-Connected layers has much potential and can lead to both high pruning amount and high classification accuracy. However, further investigation has to be conducted for the optimal pruning amount per layer on the whole network.

The effect of weight pruning can be clearly observed in figure 4.10, which shows the original weight distribution of the first convolution layer (left) compared to the pruned weights distribution (right) of test 3.

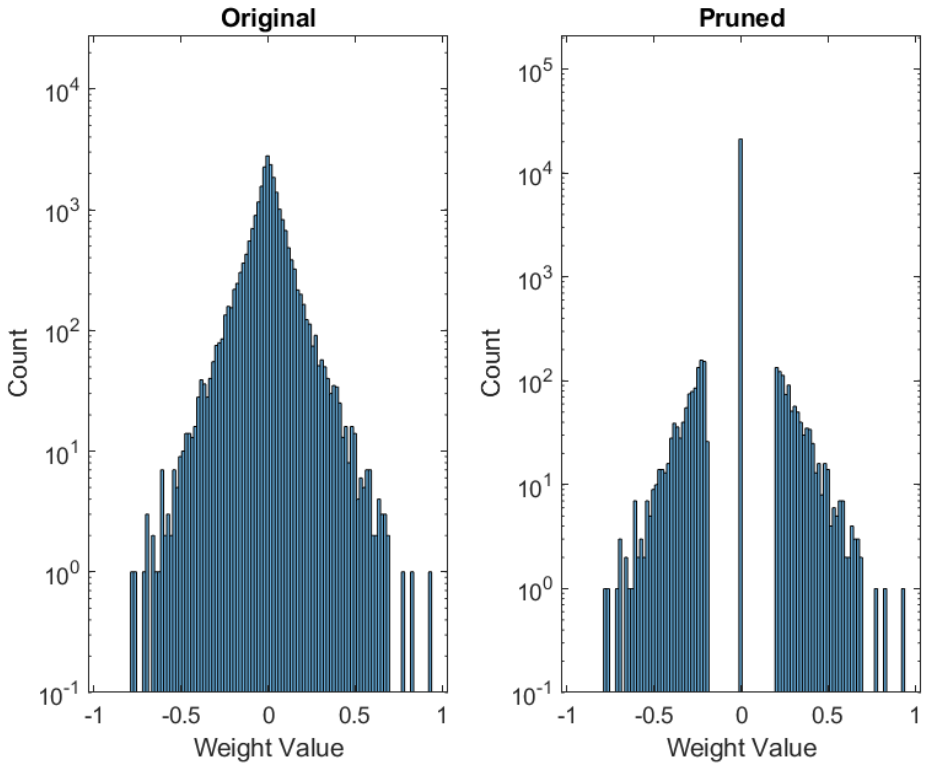

FIGURE 4.10: Weight distribution comparison of the original and the pruned weights of the first convolution layer of test 3: A high concentration of zero valued weights can be observed on the pruned weights histogram (right), with a severe absence of near-to-zero valued weights.

A high concentration of zeros can be seen on the right histogram, combined with the absence of the near-to-zero valued weights ranging from -0.2 and 0.2. Similar are all layer's weight distributions of test 3, which uses aggressive pruning, creating severe alterations. The weights distributions discontinuity is responsible for the low classification accuracy.

Figures 4.11 and 4.12 demonstrate weight distributions of test 1, which yielded the best classifications overall tests. Those figures are an example of how a pruned distribution should look like to result in relatively high classification accuracy.

Figure 4.11 shows the weight distributions of the first convolution layer with the original and the pruned weights. No discontinuation can be observed on the pruned distribution, and similar to the original concentration of zero-valued weights.

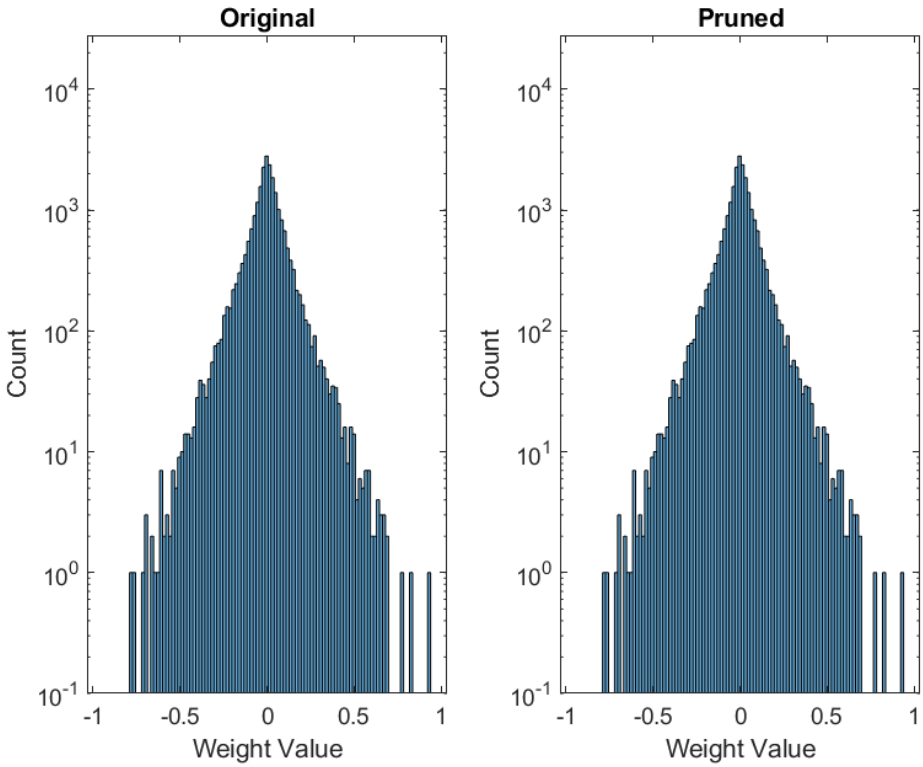

FIGURE 4.11: Weight distribution comparison of the original and the pruned weights of the first convolution layer of test 1: Similar concentration of zero valued weights can be observed on the pruned histogram (right), with almost no absence of near-to-zero valued weights.

Figure 4.12 shows the weight distributions of the first Fully-Connected layer with the original and the pruned weights. Slight discontinuation can be observed on the pruned distribution, with a higher concentration of zero-valued weights. It is vital that the discontinuation is kept really small.

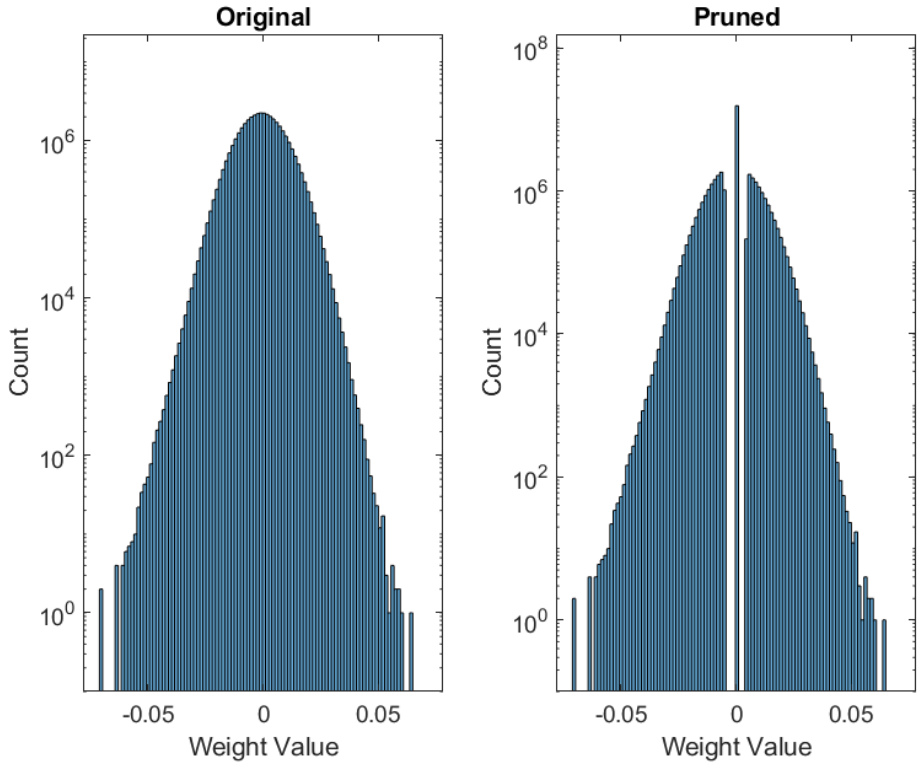

FIGURE 4.12: Weight distribution comparison of the original and the pruned weights of the first Fully-Connected layer of test 1: A high concentration of zero valued weights can be observed on the pruned weights histogram (right), with a slight absence of near-to-zero valued weights.

Chapter 5

Architecture Design

For this work, a platform was created capable of CNN inference on FPGA devices. This platform had to be created with flexibility and versatility in mind to be able to be transferred to other FPGA devices while being based on the Xilinx ZCU102, which was available in the lab. It should also be scalable to enable multi-FPGA implementations, and it should be extendable to enable for easy adding of new layer types and new layer accelerators. Furthermore, it should be able to run various CNN models' inference, but most importantly, it should provide easy experimentation and development of hardware accelerator architectures.

The basic building blocks of this platform consists of its volatile and non-volatile memory, and its compute engine. Figure 5.1 depicts the platform's block diagram, whose functionality is explained below. It should be noted that everything described below is implemented on the aforementioned Xilinx ZCU102 Evaluation Kit.

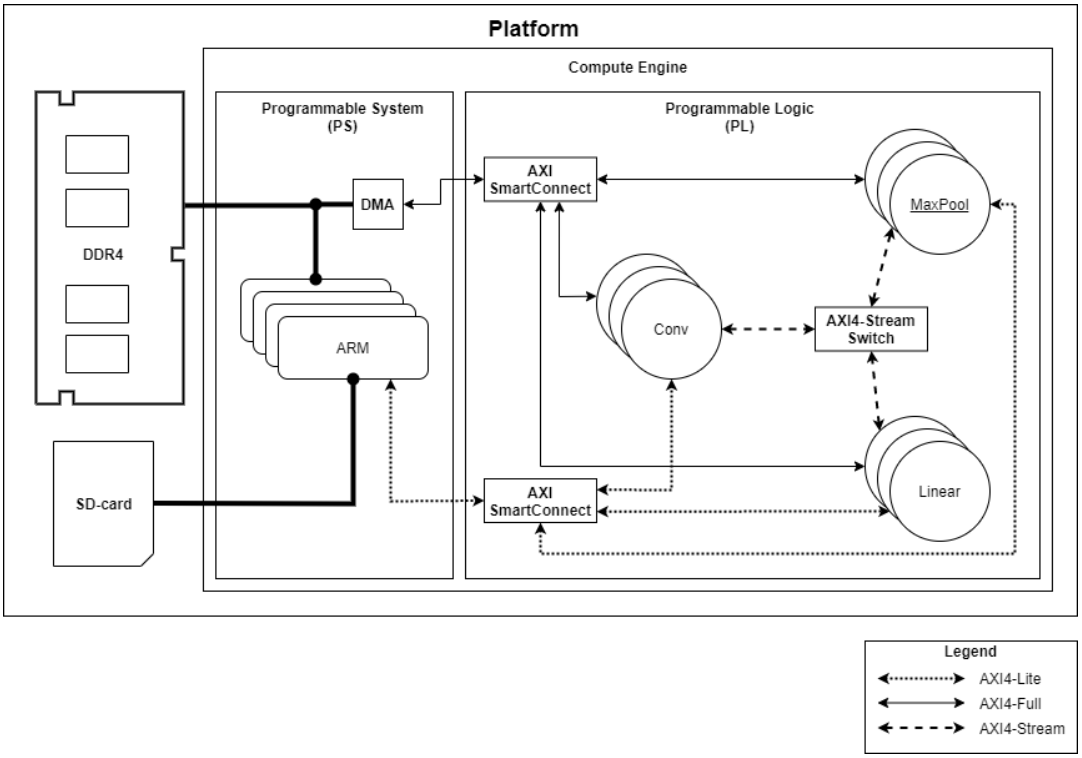

FIGURE 5.1: The platform's block diagram.

5.1 Non-Volatile Memory

This platform's non-volatile memory serves as a storage medium for all the data that networks require for their inference, which include the initialization data (network model configurations, parameters (weights and biases), class labels), and the input data (e.g., images). An SD-card is used as the non-volatile memory for this platform (also depicted as an SD-card in Figure 5.1). However, with some software extensions, other storage devices can also be used, such as SATA hard drives or even M.2 SSDs (on other FPGA platforms, e.g., QFDB). Moreover, external storage devices can also be utilized, via the Ethernet port through local-network/Internet, or via the JTAG port to avoid copying files over and over again on the platform's primary storage device but also to enable the single-source access for multiple FPGA devices.

While SD-cards have little bandwidth than other storage devices like SATA drives and M.2 SSDs, the non-volatile memory's purpose is to initialize the platform, which is a one time cost, and feed it with input data. When the platform is on its initialization phase, the initialization data from its non-volatile memory are transferred to its volatile memory to get later used from

the compute engine. Because most, if not all, networks' initialization data fit in the volatile memory, leaving a lot of empty memory space, input data are also transferred, filling as much space as possible to utilize the volatile memory's high bandwidth for input data consumption. Consequently, the SD-card's low bandwidth can be safely ignored.

If the input data are large enough to not completely fit in the volatile memory and their consumption is faster than their feed through the SD-card, a faster storage device should be used. However, in this work's experiments, this has never been the case.

5.2 Volatile Memory

The platform's volatile memory consists of two types of memory devices, the DDR4 modules found on the ZCU102 and the on-chip BRAM. The ZCU102 also provides a separate DDR4 component accessible from the FPGA's PL part, however for simplicity and generality (it is not provided on all FPGA platforms) purposes, and because it provides lower bandwidth compared to the DDR4 modules, it is not utilized on this platform.

The platform's primary memory medium is the DDR4 memory modules (also depicted as a DDR4 module in Figure 5.1) connected to both the PS and PL parts of the device. As mentioned before, it stores and serves all the required data for the platform to run. Those data are read from various files found on the non-volatile memory, in this case, the SD-card, using the ARM cores found on the PS part, and are then stored onto the DDR.

The DDR has to feed the PL part with data in chunks because, as explained in section 4.2, the integrated BRAM cannot store the whole network's parameters and activations. Consequently, every network's layer has to know where to find and how to access every piece of data it requires, information that is found on the platform's software structures and passed to the layers using the ARM cores during the setup phase.

On this platform, there is no central BRAM component that every accelerator accesses. Instead, every accelerator should implement its own BRAM, without permitting access to others. It should be the only one to know how to manage its storage. Of course, to fulfill the accelerator's needs in bandwidth, read and write ports, and latency, it can implement its BRAM as it requires, and utilize individual registers.

5.3 Compute Engine

The compute engine consists of both the PS and PL part of the FPGA device, which includes the ARM cores and the FPGA accelerators, respectively. While the bulk of the computation is handled by the PL part, the PS part handles the more sophisticated computations, such as the input data preprocessing and the accelerator configuration and scheduling.

In this platform, a network can be run only if its layers are supported by either software or hardware implementations. Software implementations are running on the ARM cores, and hardware implementations are the FPGA accelerators. By allowing both types of implementations, this platform expands its flexibility. This can not only help the development stage of FPGA accelerators by comparing their outputs with the ones generated by the software implementation, but it can also help to fully utilize the device's resources by running parts of even whole layers on the ARM cores in parallel to the hardware accelerators. Furthermore, layers that have not yet been implemented in hardware can easily be implemented in software for experimentation purposes, such as the Depth Concat layer found on the GoogLeNet (Figure 3.4).

As shown in Figure 5.1, in this work, three layer types have been implemented in hardware; the Convolutional (Conv) layer, the Max-Pooling (MaxPool) layer and the Fully-Connected (Linear) layer. In addition, multiple instances of the same layer type can also be added in the platform to enable for either parallel execution of different inferences with different inputs or even different network models, or better pipelining in a single inference. This is discussed in detail in section 5.6.

In this work, every accelerator implements a layer type, and, therefore, the platform assumes that every accelerator can handle and compute a whole layer type's computation. Every accelerator comes with its driver, which is integrated into the platform's software. The driver is responsible for setting up its accelerator, which includes setting the layer's hyperparameters and passing the information of where to find its parameters (weights and biases), its input data, and where to send its output data. Also, the driver handles the accelerator's interrupts, which, in this work, only includes the accelerator completion interrupt.

The assumption that every accelerator implements a whole layer type simplifies the accelerator's driver. As a result, the driver does not have to know its accelerator's inner workings and architecture, except for its aforementioned

settings, treating it as a black box with inputs and outputs. However, those settings are the same for every implementation of the same layer type; hence, experimenting with different architectures is made easy without developing a different driver for every single architecture. The driver can be reused on the same layer type accelerators, letting the engineer focus their efforts on the accelerator's architecture. Of course, every different layer type needs different settings; therefore, different drivers.

It is worth mentioning that every accelerator has an AXI4-Lite [81] slave port through which it can get configured. All accelerators' AXI4-Lite slave ports are connected to a Xilinx AXI SmartConnect IP [82], onto its master ports. The AXI SmartConnect, a replacement of the AXI Interconnect IP, acts as a router, providing a single slave port that is finally connected to the Zynq's (ARM cores) master port. AXI SmartConnect automatically detects the port's protocol, in this case, AXI4-Lite, and adjusts its ports accordingly. This creates a hardware connection so that the software can access and configure the platform's accelerators. Details on the AXI protocol are given in section 5.4.1.

5.4 I/O

As explained in previous sections, the I/O bandwidth provided to any hardware platform plays an essential role in its performance due to the CNNs' high bandwidth requirements. Hence, the platform should implement a network for its accelerators to enable the communication between them, the PS part, the DDR, and each other. There are three ways for high bandwidth communication, which are discussed below.

- **Memory-Mapped I/O (MMIO):** In this method, the PS communicates with its PL part's accelerators using a global address space for both its main memory (DDR) and the accelerators' I/O extensions, mapping every memory component, such as the DDR, BRAM, and registers, onto its own address range. While it is straightforward to implement, it can create a bottleneck on random memory accesses onto the DDR, with each request costing up to 50 clock cycles for the DDR's initialization. Fortunately, this is not the case with burst accesses, which request only once a range of addresses so that there is only one initialization cost for a big transfer of data.

- **Streaming (AXI4-Stream):** Using streaming interfaces, such as the AXI4-Stream, continuous communication can be established between components in a FIFO manner. Every streaming connection creates a channel between the two components with a predefined FIFO size, with a producer-consumer behavior. Communication between hardware accelerators and the DDR can be established using the Xilinx AXI DMA IP [83], which hides the DDR's initialization costs. However, AXI DMA requires knowledge of the data flow, and its function is instructed by the PS part for every transfer, increasing the system's complexity.
- **BRAM:** This method is an MMIO variation, which uses BRAM IPs to store the required data on-chip, taking advantage of the BRAM's high bandwidth. Data are transferred from the DDR to the BRAM in bursts using a Xilinx CDMA IP [84] connected to a Xilinx BRAM Controller IP [85] and its Xilinx Block Memory Generator IP [86]. Though, a major constraint is that the data have to be small enough to fit into the chip's BRAM, which is in the order of a few MBs in size.

As explained in section 4.2, the integrated BRAM cannot store the parameters for most layers. There is the case that parameters can be transferred in chunks to the central Block Memory IP; however, this creates further complexity and bandwidth bottleneck due to the low number of read and write ports. Therefore, the BRAM method described above cannot be used in this platform.

Hence, the question that arises is which method is more suitable for this platform, the MMIO or the Streaming one. To answer it, a test system was created, integrating a simple IP, created using Vivado HLS, that adds a specific value to its input data, and then returns its output data. The input data are generated on the PS part and are stored on the system's DDR. The IP's output is returned to the PS part and stored on its DDR. Then the PS validates the outputs to ensure their correctness. The IP's input and output ports are implemented both as streams and as MMIO.

The test measures the average number of clock cycles it takes for both implementations to process 40MB of data in chunks of 40kB. The chunk's size was selected greedily, because most, if not all, accelerators' requests need to be at least 40kB. The time measurement includes the input data transfer from the DDR to the IP's BRAM, the input data processing, and the output data transfer from the IP's BRAM to the system's DDR. Both implementations were

tested with 32 and 128 input and output ports bit-width. The test's results are shown on table 5.1.

TABLE 5.1: MMIO vs Stream: Processing 40MB data of 40KB bursts, showing a slight advantage over the MMIO method.

Port bit-width	MMIO avg. cycles	Streaming avg. cycles
32-bit	62700922	65611580
128-bit	15761270	16201797

Both implementations for both port bit-widths show similar results, with a slight advantage over the MMIO implementation. Selecting MMIO for the primary method of data feeding the accelerators benefits the platform not only in terms of bandwidth, according to the slight advantage depicted on table 5.1, but also in terms of simplicity of both software and hardware implementation and efficiency of hardware resources.

Consequently, the connection between the accelerators and the PS part and its DDR is established using the MMIO method, as shown on the platform's block diagram (Figure 5.1). Every accelerator has an AXI4-Full [81] master port, which is connected to a Xilinx AXI SmartConnect IP, onto its slave ports. The SmartConnect's master port then gets connected onto the Zynq's slave port. Communication to the DDR is established via the Zynq's slave port, and an integrated DMA found on the PS part.

However, the streaming method cannot be dismissed entirely, as it is perfect for communication between accelerators. Passing a layer's activations to its next layer, in terms of hardware means passing the accelerator's output data to the next accelerator as input data. Implementing this data transfer in a streaming manner avoids transferring data from the FPGA to the DDR and then transferring them back to the FPGA using the MMIO method, which can be a significant bottleneck. As a result, every accelerator can optionally have additional input stream and output stream ports. The accelerators driver configures it to either use the MMIO or the streaming method.

It is vital that every accelerator implements at least the MMIO method, to avoid running into deadlocks. For example, let there be a system with one instance of a layer type's accelerator, and the accelerator's output stream needs to get connected back to the same accelerator's input stream. A deadlock occurs when the output stream is full while the accelerator has not finished its

execution. In this case, the accelerator hangs, waiting for the stream to accept more data, while the stream hangs, waiting for its data to get consumed. This problem can easily be tackled by simply sending the accelerator's outputs to the DDR using the MMIO method, and when its execution is complete, it can then read them back again from the DDR.

A Xilinx AXI4-Stream Switch IP [87] is used to create a star topology [88] stream network, connecting every accelerator's stream ports, as shown on the platform's block diagram (Figure 5.1). Every connection is assigned an address in the AXI4-Stream Switch IP before synthesis so that the TDEST signal of the AXI4-Stream protocol can be set accordingly by the sender accelerator. The AXI4-Stream Switch IP was preferred to the AXI4-Stream Interconnect IP because the latter is the same as the Switch, but it also allows connections of streams with different characteristics in the cost of some hardware resources. However, all accelerators are implemented with the same stream characteristics, so the Interconnect's use is redundant.

5.4.1 AMBA AXI4 Interface Protocol

The AMBA AXI4 (Advanced eXtensible Interface 4) [81] is the AMBA interface specification from ARM, which is integrated into the Xilinx Design Suite tools to offer a single standard interface and simplify the IP integration. All Xilinx IPs that require any configuration or large amounts of data implement at least one of the AXI4's variation. The AXI4 protocol has three variations for different use cases.

- **AXI4-Full:** Used for high-performance memory-mapped applications, supporting bursts of up to 256 beats.
- **AXI4-Lite:** The simplest of all three variations, with the least hardware resources requirements. It is used for low-throughput memory-mapped applications, usually for accessing control registers, with every transaction having a burst length of one
- **AXI4-Stream:** Used for high-speed streaming unidirectional data transfers from master to slave, supporting multiple data streams using the same set of shared wires, and multiple data widths withing the same interconnect.

This platform makes usage of all three variations for different purposes, as shown on its block diagram (Figure 5.1). The AXI4-Lite is used for configuring the accelerators; the AXI4-Full is used to implement the accelerators' MMIO, as described in section 5.4, and the AXI4-Stream is used to create the star topology network between the accelerators.

5.5 Software

The platform's software was developed using the Xilinx Vitis IDE, previously known as Xilinx SDK. Xilinx Vitis provides the necessary compilation toolchain, and the ability to program the FPGA and run software on the ARM cores. This platform cannot function without its software. It should be clarified that everything that is considered a software part runs on the device's processor (ARM cores). The software consists of the accelerators' drivers, the scheduler, the application logic, and the user interface.

The drivers are responsible for the communication between the software and hardware of this platform. They are used to handle the initialization, configuration, and use of each hardware component. They also handle the accelerators' various interrupt events, affecting the software's execution. Every accelerator's driver has to implement several functions with specific functionality and naming scheme, creating an abstraction layer between the drivers and the rest of the software. This makes integrating a new accelerator into the platform easier because the other software parts "know" how to use the new driver, expanding the platform's modularity. Moreover, it is essential that the interrupt service routines are kept as small as possible to avoid missing any other interrupt events while executing them.

The application logic is responsible for configuring the platform depending on the user's input. It handles the parsing of the network model configuration files, the loading of the network's initialization data (parameters and labels), the input data preprocessing, and the accelerator execution according to the scheduler's instructions. It is designed with data arbitration in mind, in order to easily experiment with different data types used in the network's parameters and activations. The application logic's data type is the same with the one used in the accelerators' architecture, and when set, it propagates to the whole software.

For the sake of simplicity, a command-line interface implements the user interface. Most FPGA platforms have a UART port that can be used to print

messages, accept user input, and debugging. Therefore, this platform utilizes the UART port for its user input requirements. From the application's menus, the user can select several functions, among which there is a self-test routine that tests the platform's integrity, a network model selection with its corresponding parameters, an image count selection, and an option to run both software and hardware implementations of every layer for output data checking. While not implemented, the user interface can be expanded to a graphical user interface using a standalone program or server, running on a host PC that reads and writes the aforementioned UART port.

The scheduler is the last but not least, part of the platform's software. It is responsible for scheduling all the necessary tasks for the network's inference to execute. A task can be executing a layer or a group of layers either in software or using the hardware accelerators. The scheduler is also responsible for scheduling the input and output methods of each accelerator (MMIO and Streaming), the input source and output destination. Per platform implementation, there might be needed a different scheduler strategy. For example, there is a different strategy for a platform with only one instance per accelerator type, and a different one when there are multiple, or when the accelerators' execution can be pipelined. For more information on the platform's scheduler strategies see section 5.6

A simplified flowchart of the platform's software execution can be seen in figure 5.2. On system boot-up, the drivers are discovering and initializing all peripheral devices, such as the SD-card and the UART ports. Afterward, they discover and initialize every accelerator existing in the FPGA's PL part. During this step, the interrupt handlers are also getting set. Next, the user interface asks the user for which network from the available it should run its inference and all the aforementioned running options. Given the user's input, the application logic reads the network model file, parses it, and loads it to the platform's memory. It also loads the network's parameters, labels, and input data. Then, for each input data, the scheduler adds the tasks into a list to be run whenever possible. Then, for every task, the corresponding accelerator is set up and triggered to run using its drivers. The platform finishes when there are no more tasks to be run and input data to get processed.

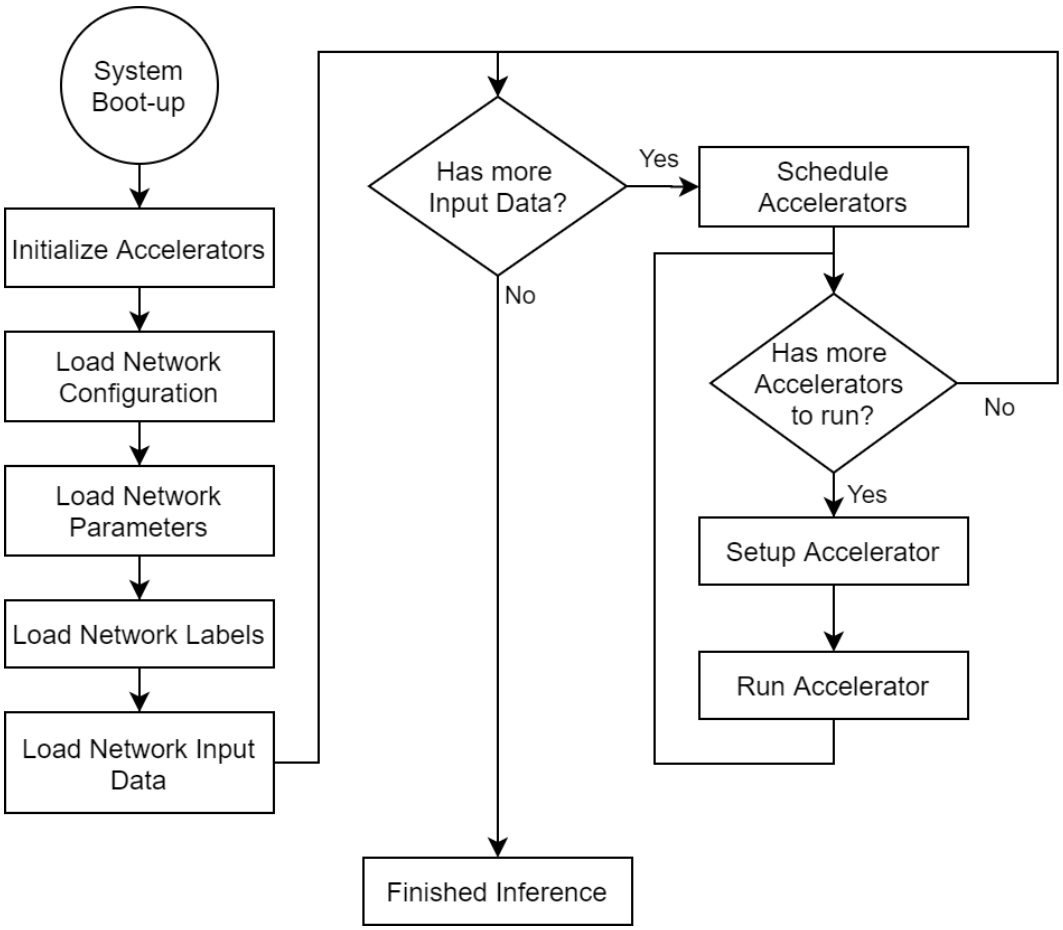

FIGURE 5.2: The platform's flowchart.

5.6 Scheduler Strategies

Scheduling can play a significant role in the platform's performance. As mentioned in the previous section, the scheduling strategy is dependent on the platform's implementation and goals. The number of same-type accelerator instances and whether the platform is throughput or latency optimized, all affect the strategy selection. To study and demonstrate the various strategies, a MATLAB model of a CNN network's inference execution was created that can compute the execution characteristics of any CNN network model. This work's main CNN network is AlexNet, on which all experiments below are based.

The MATLAB model creates a timed schedule of a CNN inference depending on its hyperparameters. It computes the number of clock cycles a layer requires for its full execution, and depending on the strategy selected, it places the layer's execution start and finish timestamps. The clock cycles required

for every full execution is equal to the sum of the clock cycles needed per assembly instruction to be executed. The number of clock cycles required for every assembly instruction can be set on the MATLAB model's parameters. However, for simplicity, all instructions are considered to run in a single clock cycle.

The following figures show the starting clock cycle, the ending clock cycle, and the duration of every layer, wherever there are colored boxes. Every colored box has its label, defining the layer that it represents. The label is coded as the layer's type and its serial number; Conv for convolutional layers, MaxPool for max-pooling layers, and Linear for fully-connected layers. It should be noted that the ReLU activation function is embedded into the convolutional and fully-connected layers, as shown on algorithms 2 and 5.

5.6.1 Serial Strategy

A baseline schedule was created using a serial execution strategy, as shown in figure 5.3. In this strategy, every layer starts when the previous layer has finished generating all its outputs. This strategy can be used when there is only one accelerator instance per layer type, and the accelerators do not support layer pipelining. It is also an excellent strategy for debugging and validating the platform and its accelerators because of its simplicity.

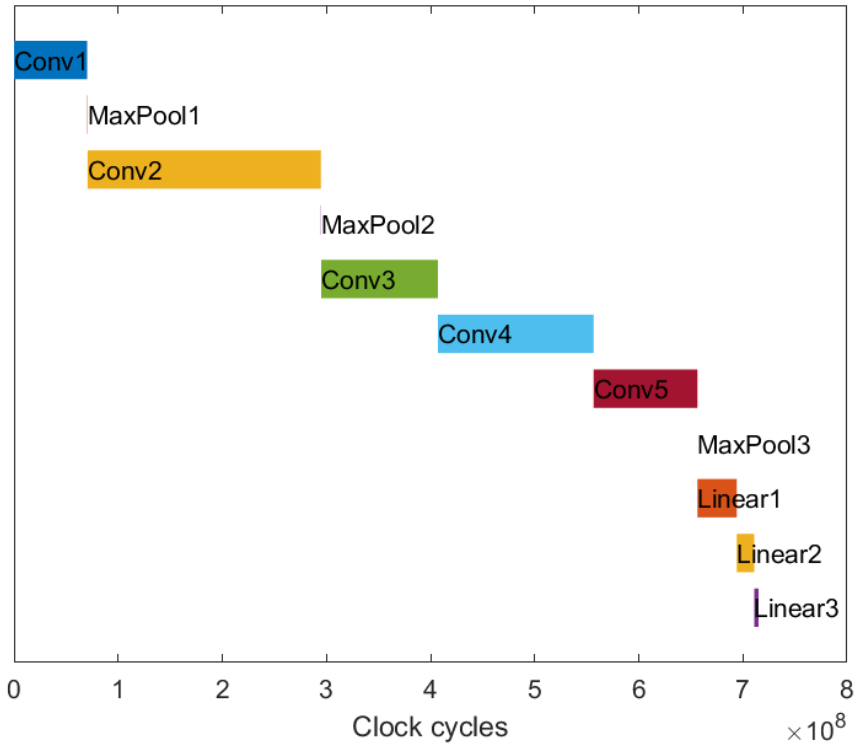

FIGURE 5.3: AlexNet serial execution: Convolution layers consume 90% of total clock cycles needed for a full inference.

It can be observed that using the serial execution strategy, around 90% of the total clock cycles is consumed from the convolutional layers, and almost 0% is consumed from the max-pooling layers. Hence, according to Amdahl's law, see section 5.7, the convolutional layers should get the most hardware resources for their accelerators to create as much parallelism as possible, while fully-connected layers come next, and max-pooling layers come last.

5.6.2 Layer-Pipelining Strategy

Another scheduling strategy is applying pipelining within the layers. In this strategy, a layer is fed with input data as soon as the previous layer generates a single output. This hides the next layer's latency as much as possible by computing outputs in parallel with the computation of its next-to-be-processed inputs. A schedule using the layer-pipelined strategy is shown in figure 5.4. To produce such a schedule using this strategy, it is considered to exist as many accelerators per layer type as needed. In this schedule, there

are needed five instances of a convolutional accelerator, three instances of a max-pooling accelerator, and three instances of a fully-connected accelerator.

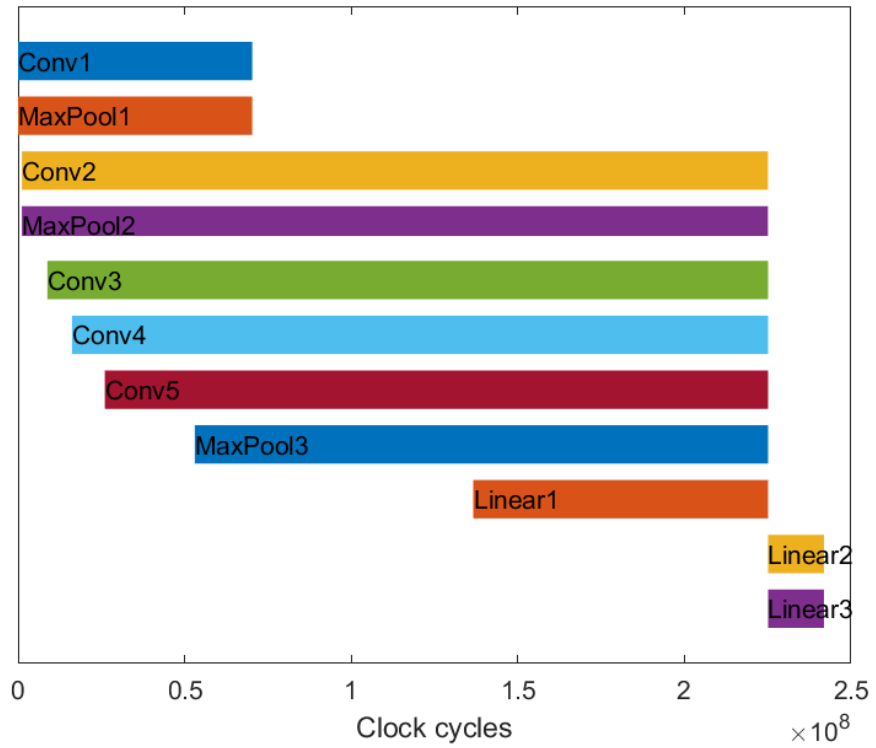

FIGURE 5.4: AlexNet layer-pipelined execution: There is a speedup of almost 3 times compared to the serial strategy.

The first four layers, Conv1, MaxPool1, Conv2, and MaxPool2 seem to start their execution immediately; however, this is not the case. In fact, they all start on different timestamps, each later from its previous one, but it just cannot be shown in figure 5.4 due to the x-axis' scale. Conv1, MaxPool1, and Conv2 generate their first outputs in a few clock cycles compared to the x-axis scale. The same artifact appears on the finishing timestamps of the first two layers, Conv1 and MaxPool1, and the next six layers, Conv2 to MaxPool3.

From figure 5.4, it can be seen that the whole execution has significantly been reduced compared to the serial strategy shown in figure 5.3, speeding it up almost three times.

However, to use the layer-pipelined strategy, the accelerators have to support it. This means that they need to process their inputs as soon as possible to generate their outputs. Because convolutional and max-pooling layers process their inputs in the 3D and 2D space, they have different implementation

requirements to support layer-pipelining, compared to the fully-connected layers that compute their inputs in the 1D space.

Convolutional and max-pooling layers need to process their inputs in a way that they can generate outputs in a specific order. The optimal output order is the one that the next layer wants its inputs to be in, to also produce useful outputs for its next layer. The convolutional and the max-pooling layers produce outputs using cubes or squares of inputs, respectively, starting from the top left input of the grid. Those inputs are created from the previous layer, which also generated them using cubes or squares of inputs. Hence, the deeper the layer, the bigger the required cube or square of the first layer's inputs. Consequently, the optimal order of generating outputs starts from the top left and moving downwards and right in zones, always creating bigger and bigger cubes or squares. Figure 5.5 depicts the order a convolutional or a max-pooling layer generates its outputs. It should be noted that the convolutional layers generate their outputs for all of their output channels, and then they move onto the next output.

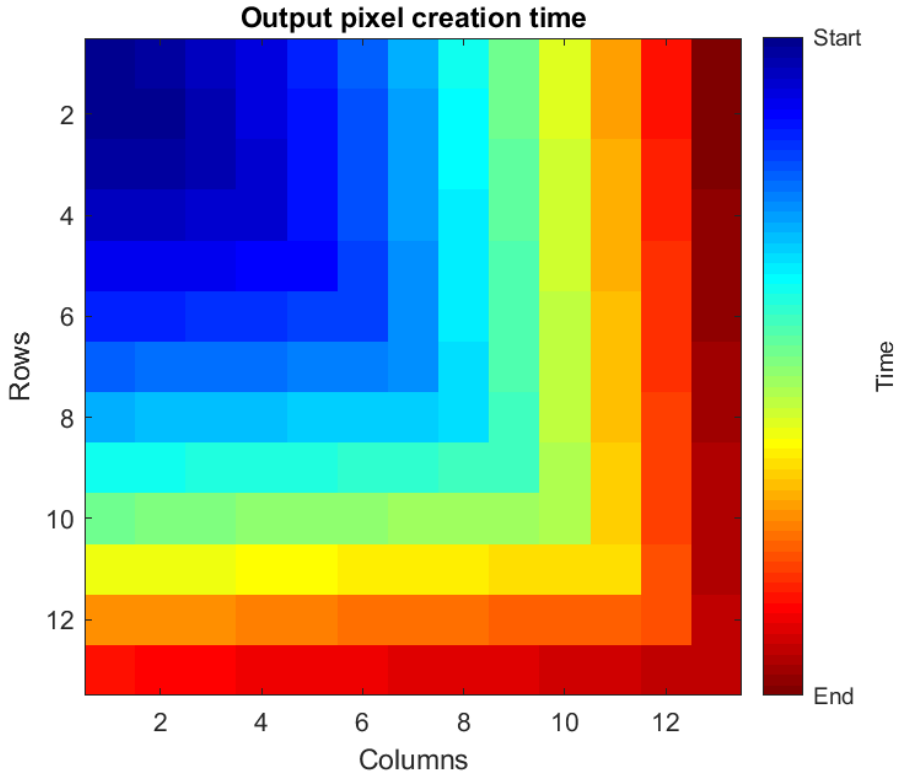

FIGURE 5.5: Convolutional and Max-Pooling layer output order for layer-pipelining: Outputs colored in blue are generated before the red ones.

In figure 5.5, every pixel is an output of a convolutional or max-pooling layer, and it is color-coded concerning the time it is generated, with blue and red colors representing the start and end, respectively, of the generation time.

However, layer-pipelining for convolutional and max-pooling layers significantly increased the implementation complexity of their accelerators. Also, caching weights and inputs in the accelerator's BRAM and registers can become a major obstacle due to their size and access patterns. Figure 5.6 shows the input usage frequency for a convolutional or a max-pooling layer with a stride of one.

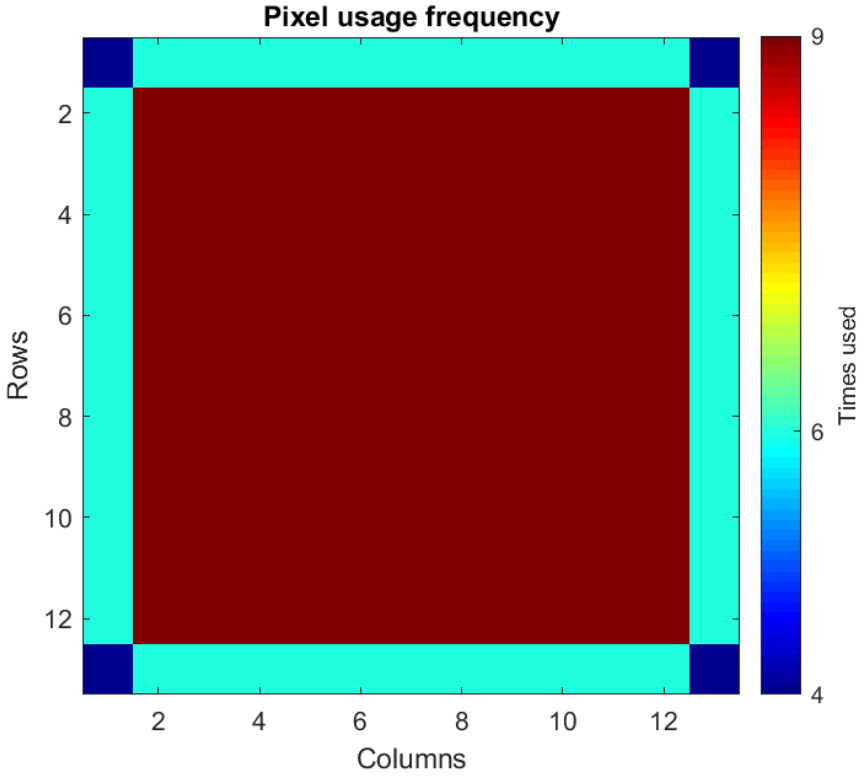

FIGURE 5.6: Convolutional and Max-Pooling layer input pixel usage frequency a stride of one: Blue inputs are rarely used, while red ones are used frequently.

Figure 5.7 depicts the input usage frequency for a convolutional or max-pooling layer with a stride of four, showing even more complex access patterns.

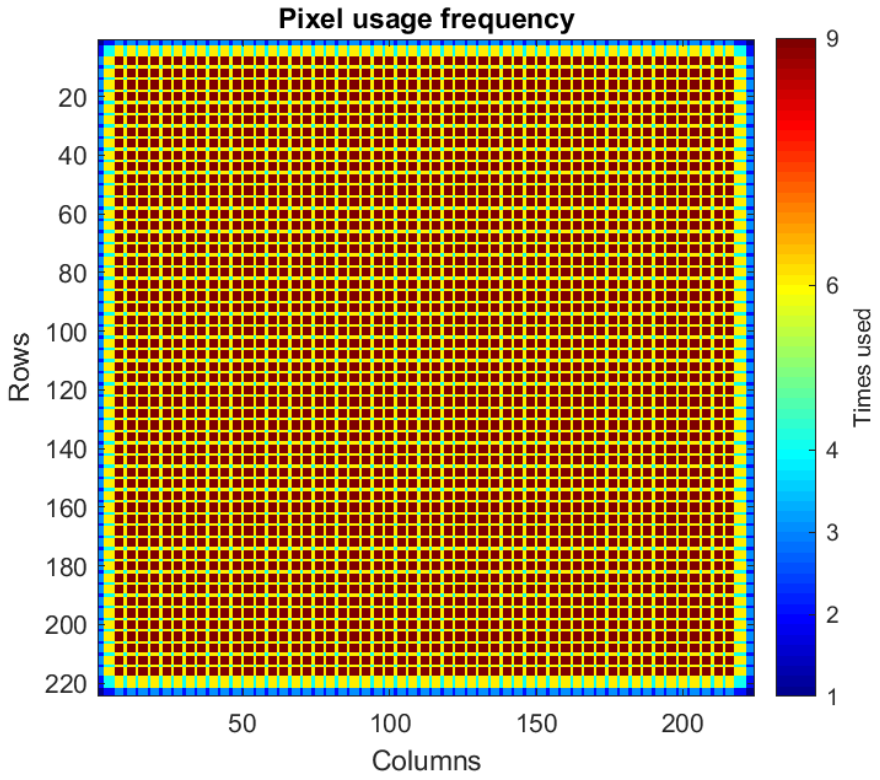

FIGURE 5.7: Convolutional and Max-Pooling layer input pixel usage frequency using a stride of four: Blue inputs are rarely used, while red ones are used frequently.

On the other hand, fully-connected layers do not require a specific order for their inputs. However, they need to store partial results for their outputs. Whenever an input is given to the fully-connected layer, it adds it to the partial result of each output after multiplying it with its corresponding weight per output. This creates higher memory requirements for the implementation of the fully-connected accelerator.

Unfortunately, due to the layer-pipelining's significantly increased complexity, this work does not implement it on its accelerators, and it is only presented for completeness and ideas for future work.

5.6.3 Multi-Inference Strategy

When multiple accelerators per layer type can be placed into the FPGA device's PL part, they can be utilized simultaneously by running multiple inferences in parallel. The multi-inference strategy schedules two or more images for inference, increasing the platform's overall throughput.

5.6.4 Image-Pipelining Strategy

Similar to the multi-inference strategy, when there are multiple accelerators per layer type, multiple images can be fed into the network in a pipelined manner. Every accelerator instance represents a single layer of the network. Hence, the first image can be fed into the first layer. Afterward, when the first layer has generated its outputs, they are fed to the second layer, and the first layer is fed with the second image. This continues for all images to be inferenced. Therefore, when the pipeline is full, all accelerators process different input images from each other. This strategy can decrease the platform's inference latency, and, potentially, the platform's throughput.

5.7 Amdahl's Law

Amdahl's law [89] is a formula that calculates the theoretical speedup in latency of the execution of a fixed workload task, when the system's resources are improved or increased. While speedup was firstly used on parallel processing, it can also be used after any resource enhancement.

Latency is the time required for a system to compute a single task and is defines as:

$$\begin{aligned} Latency = \frac{1}{v} = \frac{T}{W}, \\ \text{v: the task's execution speed,} \\ \text{T: the task's execution time,} \\ \text{W: the task's execution workload} \end{aligned} \tag{5.1}$$

Throughput is the maximum processing rate of a specific task and is defined as:

$$\begin{aligned} Throughput = r * v * A = \frac{r * A * W}{T} = \frac{r * A}{L}, \\ \text{r: the execution density,} \\ \text{A: the execution capacity} \end{aligned} \tag{5.2}$$

The speedup is defined for both latency and throughput, as shown in the equations below:

$$S_{Latency} = \frac{L_1}{L_2} = \frac{T_1 * W_2}{T_2 * W_1} = \frac{1}{(1-p) + \frac{p}{s}},$$

p: the task's portion that benefits from the resource enhancements,

s: the speedup of the task's portion that benefits from the resource enhancements

(5.3)

$$S_{Throughput} = \frac{Throughput_2}{Throughput_1} \tag{5.4}$$

The maximum theoretical speedup can also be defined as:

$$MaxSpeedup = \lim_{s \to \infty} S_{Latency} = \frac{1}{1-p} \tag{5.5}$$

5.8 Platform Accelerator Architectures

In this work, three different accelerators have been developed, one per layer type used in most CNNs. Those are the convolution, the max-pooling, and the fully-connected accelerators. Every accelerator has two variations; a simple one that only processes a single input similar to serial execution for testing purposes, and a performance-oriented one for production/system deployment. Some of the accelerators' components are being described separately to increase the readability of their architecture diagrams.

5.8.1 Convolution Accelerator

The convolution accelerator is the most complex, requiring lots of BRAM slices and clock cycles to complete. Figure 5.8 depicts the simple version of the convolution accelerator's architecture. While it is not meant to be used in production, it is great for testing purposes and validating the platform's and accelerator's subsystems.

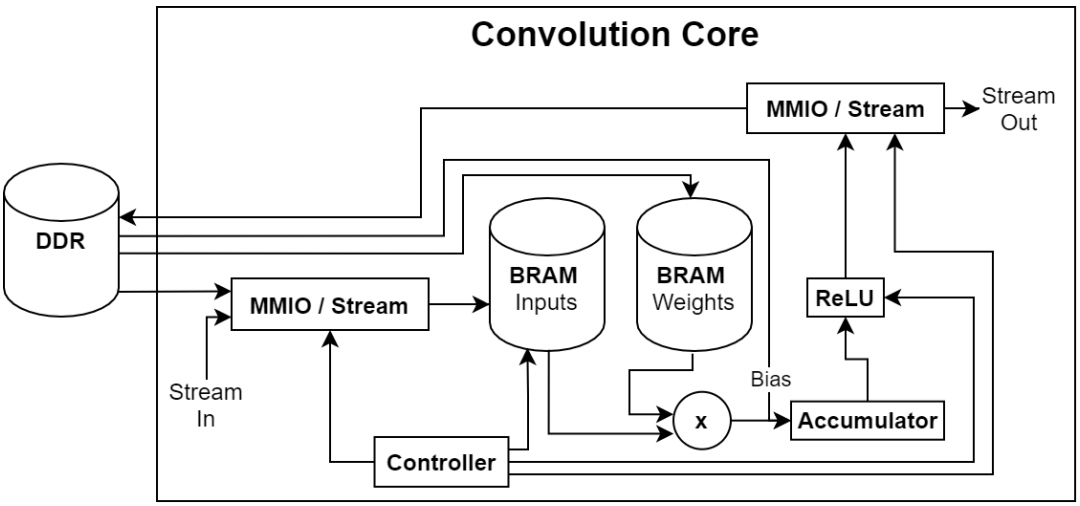

FIGURE 5.8: Convolutional layer serial accelerator.

The convolution accelerator reaches the platform's DDR memory for its parameters through an AXI4-Full port, while its input data are given either through the same AXI4-Full port or through a dedicated AXI4-Stream port. The input data source is selected using a *MMIO / Stream* component, as instructed by the accelerator's *controller* component, and the data are then stored into a BRAM instance for later use. The layer's weights are stored kernel by kernel onto another BRAM instance. This way, the BRAM requirements are kept low, compared to storing the whole layer's weights, which might also be impossible, due to the BRAM's limited size. Because every kernel is convoluted with the entire input, every weight in that kernel and every input pixel is accessed multiple times. Hence, the inputs and weights BRAM instances are used as caches. It is also worth noting that each BRAM instance implements only one read and one write port because only one input and one weight is accessed every clock cycle.

The computation starts when the input data and weights are stored in their corresponding BRAM instances. Firstly, the *accumulator* component, whose architecture is described below, is initialized by the kernel's bias. Afterward, the controller asks every BRAM instance for specific data to be sent to the *multiplier* component, whose results are then sent to the *accumulator*. This operation continues with the controller selecting the appropriate data for a correct convolution until a single output is ready. Then, it is sent from the *accumulator* to the *ReLU* component, whose architecture is described below. The *controller* then instructs the *ReLU* component to either apply its activation function or pass it through. Lastly, the *ReLU* component sends its output

to another *MMIO / Stream* component, which either sends it back to the platform's DDR or a dedicated AXI4-Stream port, also instructed by the *controller*. This procedure continues until the full convolution operation is completed, processing the entire input. The *controller's* configuration is handled by an AXI4-Lite port, accessed by the PS part.

The *accumulator* component, shown in figure 5.9, is a simple accumulator, which has an input port that gets added with the register's output, used to store the partial result. The adder's output is connected back to the register's input port and the *accumulator* component's output port.

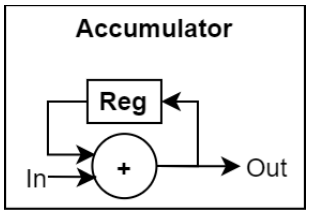

FIGURE 5.9: Accumulator component.

The *ReLU* component, shown in figure 5.10, applies the ReLU activation function (see section 2.2.1) to the component's input, when the enable port is set, otherwise it passes through the input to the output port. It consists of a simple comparator and two multiplexers.

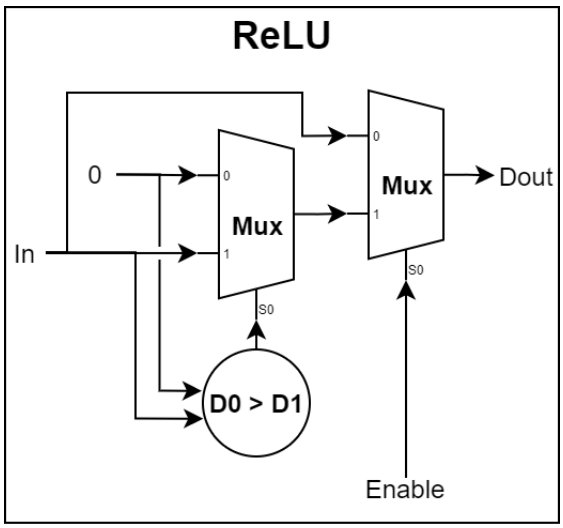

FIGURE 5.10: ReLU component.

Figure 5.11 depicts the high performance version of the convolution accelerator's architecture. It is very similar to the simple version, with the main difference being the number of inputs that can be processed in parallel. This version uses a *multiplier* array, an adder tree, and multiple read and write ports on the input and weights BRAM instances. In this version, the multiplier array is fed with data using all BRAM read ports to feed all of its *multiplier* components in parallel, and then it passes its results to the adder tree in order to create a single partial output. The data fed to the multiplier array consist of a single kernel row of weights and inputs. Then, the partial output is fed to the *accumulator*, and a new row of data is given to the multiplier array. The rows come in the order of top to bottom row of a single channel, and then the next channel follows. This process continues until every row, and channel of a specific kernel is convoluted with the corresponding input area. Afterward, the *accumulator* passes its output to the *ReLU* component, and the process continues similar to the accelerator's simple version.

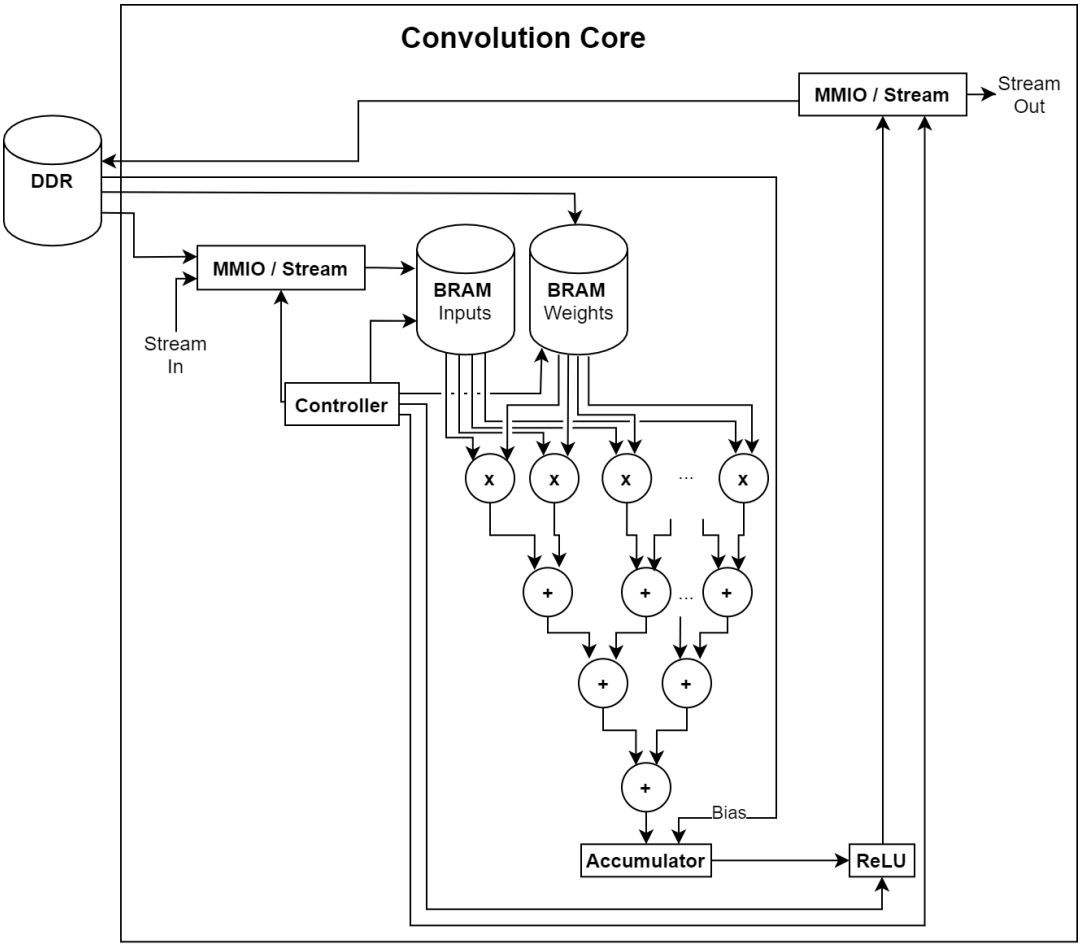

FIGURE 5.11: Convolutional layer kernel-row-parallel accelerator.

The number of *multiplier* components is set to be at least the maximum kernel size of every convolutional layer found in the network to be able to process its whole row at once. Granted that the convolution core also has to support the smaller kernel sizes found on the network's convolutional layers, there are structures that feed the excess *multiplier* components with zeros to not affect the computation's result.

5.8.2 Max-Pooling Accelerator

The max-pooling accelerator is the lightest, requiring little BRAM and external I/O. Figure 5.12 depicts the simple version of the accelerator's architecture. Similar to the convolution accelerator, it is only targeted for testing and validation purposes. However, it might also be suitable for production to save resources due to its simplicity. Besides, as shown in the serial scheduling (Figure 5.3), max-pooling layers contribute almost nothing to the overall time required for a complete inference, allowing for slower architectures, saving resources.

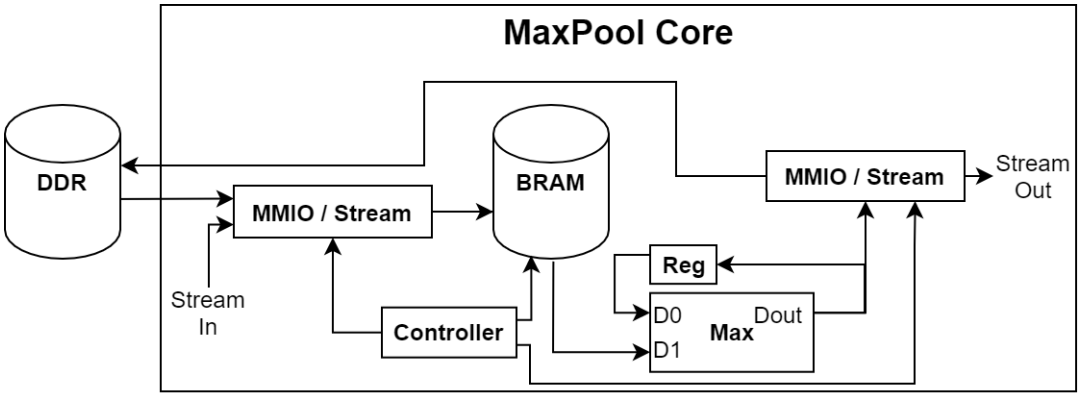

FIGURE 5.12: Max-Pooling layer serial accelerator.

The max-pooling accelerator's I/O is handled by two *MMIO / Stream* components, similar to the convolutional accelerator. The BRAM instance, using a single read and a single write port, loads a single channel from the whole input. It then feeds the *max* component, whose architecture is described below, as instructed by the accelerator's *controller*. The *max* component's output is fed to the *register*, which feeds it back to the *max* component to process the next input. In other words, this structure finds the maximum value in a given array of input data. This process continues until a whole kernel of input data is processed, generating a single output, which is then sent to the

output *MMIO / Stream* component. Afterward, the next part of the input is processed, and when the whole channel is completed, the next one gets loaded to the BRAM instance. The accelerator finishes when all of the input channels are processed.

The *max* component, shown in figure 5.13, outputs the maximum input of the two it is given. It comprises only two components, a comparator, and a multiplexer.

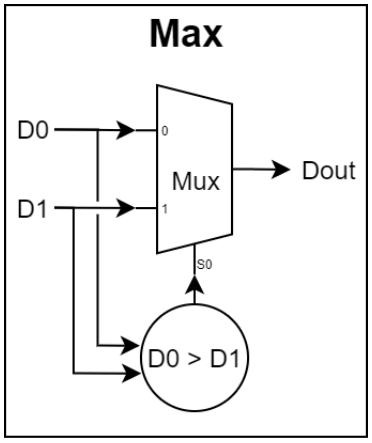

FIGURE 5.13: Max component.

The max-pooling accelerator's high performance version architecture is depicted in figure 5.14. The difference between the simple architecture lies upon the BRAM instance's read ports number and the *max tree* component, whose architecture is described below. Using multiple read ports, multiple inputs can be inserted to the *max tree* component in a single clock cycle, which in turn it outputs the max value of its given input. This architecture processes the whole kernel, generating a single output in every iteration. On every iteration, the kernel moves onto the given input, until the channel is fully processed. Then the next channel gets loaded onto the BRAM instance, and the process continues.

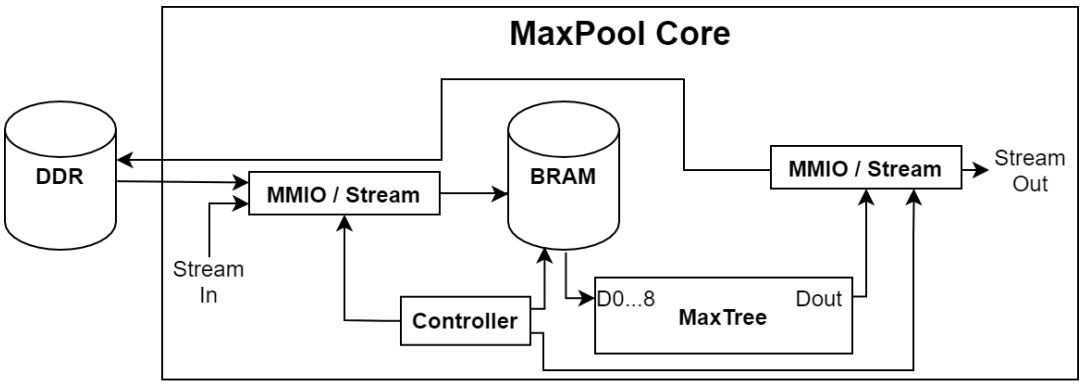

FIGURE 5.14: Max-Pooling layer kernel-parallel accelerator.

Like the convolution accelerator, the max-pooling accelerator must support all kernel sizes found in the network's max-pooling layer. Hence, the BRAM read ports number and the *max tree* component's input ports number are set to be the network's maximum max-pooling kernel size. When a layer with a smaller kernel size is processed, the *max tree* component's excess inputs are fed with the minimum possible value not to affect the computation.

The *max tree* component, shown in figure 5.15, is a tree of *max* components, capable of finding the max value from the given input. The kernel's 2D matrix is flattened into a 1D array fed into the *max tree* component to find the input area's maximum value.

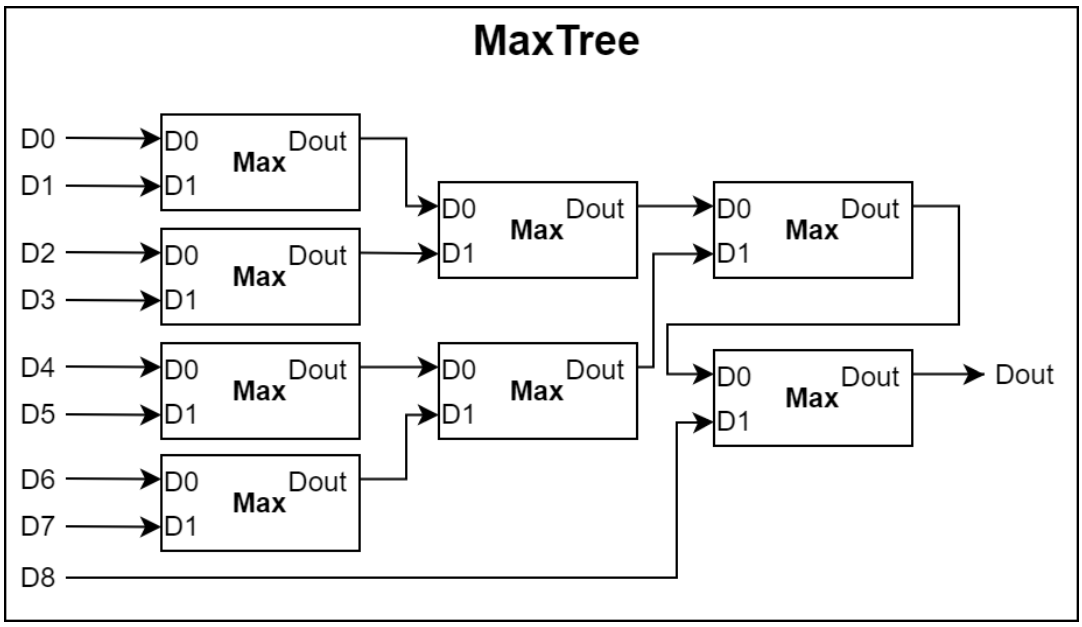

FIGURE 5.15: MaxTree component.

5.8.3 Fully-Connected Accelerator

Unfortunately, the fully-connected accelerator is the most limited in terms of parallelism capabilities since the fully-connected layer's weights are used only once, and therefore, they cannot be cached into an in-accelerator memory instance (BRAM or registers). Only the input data are used multiple times, so storing them into a BRAM instance can help avoid I/O with the platform's DDR.

Figure 5.16 shows the simple version of the fully-connected accelerator's architecture. Like the other accelerators, the input and output data are handled, as instructed by the accelerator's *controller*, using two *MMIO / Stream* components, connected to their dedicated input and output AXI4-Stream ports and the AXI4-Full port. Firstly, the input data are loaded onto the BRAM instance, through the input *MMIO / Stream* component. Afterward, the *accumulator* component is initialized with the appropriate bias, read directly from the platform's DDR through the accelerator's AXI4-Full port. Then, the *controller* instructs the BRAM instance to sequentially feed the *multiplier* with a single input datum on every iteration. It also instructs the DDR to feed the *multiplier*, through the AXI4-Full port, with the corresponding weight on every iteration. The *multiplier* component's output is then forwarded to the *accumulator*. When all weights and inputs are multiplied and accumulated, a single output is generated. It is then sent from the *accumulator* directly to the *ReLU* component, which applies its ReLU activation function if instructed so by the *controller*. Lastly, the *ReLU* component's result is sent to the output *MMIO / Stream* component to write it back to the DDR or to send it to the next accelerator using its dedicated output stream port. The accelerator finishes when all inputs and all weights have been processed, generating the total output.

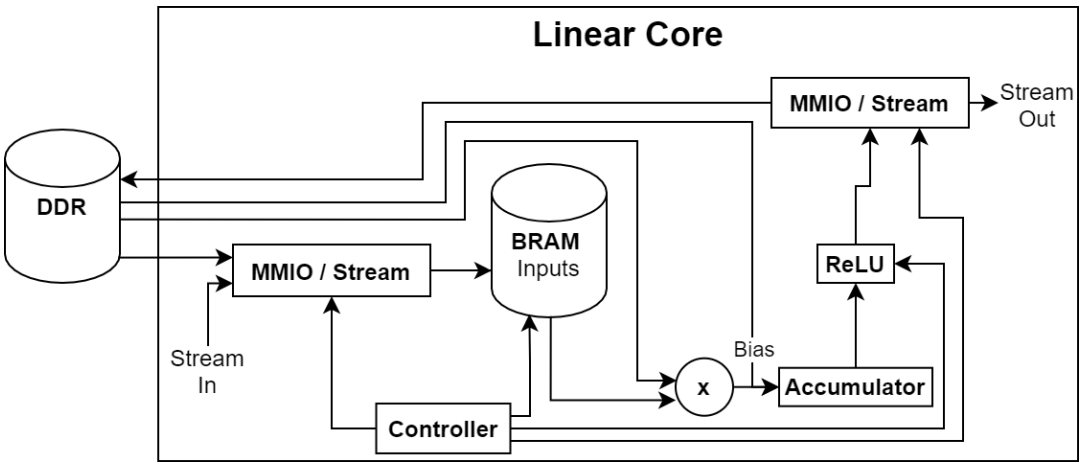

FIGURE 5.16: Fully-Connected layer serial accelerator.

Although this version is relatively simple, it uses a lot of BRAM only to access a single input, sequentially. Hence, the BRAM instance might be overkill for this use, wasting resources. Although, it should be noted that this simple version of the accelerator's architecture is only targeted for testing and validation purposes.

A more optimized architecture can be seen in figure 5.17, where a multiplier array and an adder tree are utilized, the input BRAM instance is replaced with simple registers, and the weights are given by splitting the 128-bit wide AXI4-Full port. In a sense, the accelerator processes its input data in parts creating partial outputs on each iteration.

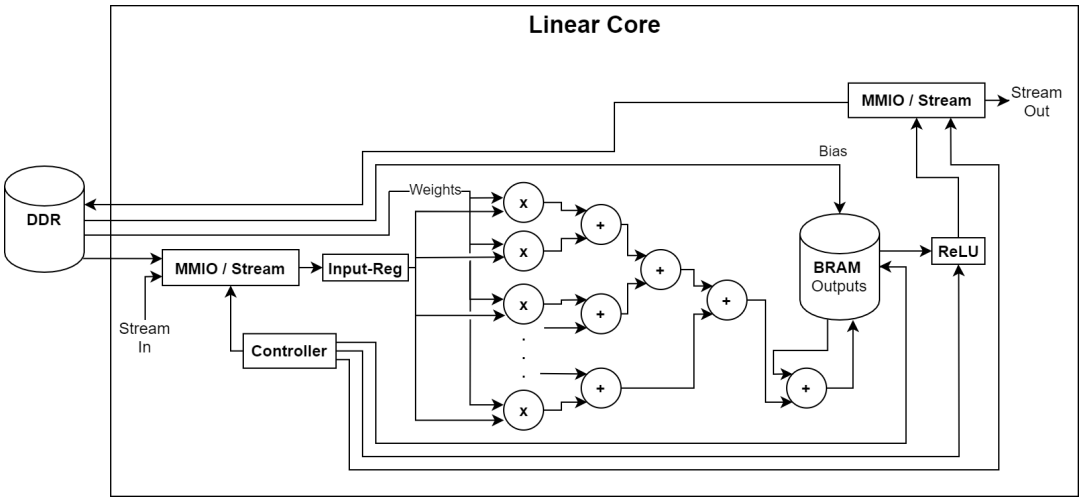

FIGURE 5.17: Fully-Connected layer accelerator with partial outputs.

Like the simple version, the input data are handled by the input *MMIO / Stream* component; however, they are then written in parts onto the *input registers*. Every register has its own read and write ports so that multiple inputs can be sent to the multiplier array in parallel. Furthermore, the weights are still read from the DDR through the accelerator's AXI4-Full port, taking advantage of the port's width. For example, if a single weight is a 32-bit float and the port is 128-bit wide, every 128-bit word can include four weights. Hence, the size of the multiplier array depends on the weight's data type. Also, there is no need for more input registers than the number of weights that can fit into a single 128-bit word, so the input data are saved and processed in parts throughout the whole output. The process starts by initializing the output BRAM instance with the biases. The output BRAM instance is used to store the partial outputs; therefore, every cell is initialized with a single bias, corresponding to the output it represents. Starting the computation, the first input part is multiplied with its corresponding weights, and then the multiplication's results are sent to the added tree, generating a partial output stored into its BRAM cell. Afterward, the same input part is multiplied with the next set of weights, and a new partial output is generated and stored. When the first input part has been processed throughout all the partial outputs, the second part takes its place, and the process continues, adding the newly created partial outputs to the already existing ones in the BRAM's cells. The process continues until the whole input has been processed, and the full outputs are generated. Lastly, the full outputs pass through the *ReLU* component and are then sent to the *MMIO / Stream* component.

Chapter 6

FPGA Implementation

6.1 Tools Used

This work's CNN inference accelerator was implemented and optimized for FPGA platforms using the Xilinx Vivado Design Suite - HL System Edition 2019.2 [90]. Vivado Design Suite is a software suite developed by Xilinx for its FPGA devices for analysis and synthesis of Hardware Description Language (HDL) designs, written in VHDL or Verilog. It is superseding Xilinx ISE [91], as a complete rewrite, with additional features for System-on-Chip (SoC) design and High-Level Synthesis (HLS). The tools used in this work are Xilinx Vivado HLS, Xilinx Vivado IDE, and Xilinx SDK.

6.1.1 Vivado IDE

Xilinx Vivado Integrated Design Environment (IDE), released in 2012, is the basis for all Xilinx tools. It serves as a GUI front-end for the Vivado Design Suite. All Vivado Design Suite tools integrate a native TCL interface, which can be accessed from IDE's GUI and the TCL console. Vivado IDE can compile, synthesize, implement, place and route FPGA hardware designs written in high-level languages such as C/C++, and HDLs such as VHDL and Verilog.

In addition, using the IP Integrator tool, hardware systems can be designed by graphically connecting IP blocks and configuring them through their GUI, with no coding involved, hence, accelerating the design process. Integration automation features such as auto-connecting and auto-configuring blocks further accelerate the design process. IP blocks can be created using the integrated IP Packaging functionality for VHDL or Verilog designs and via the Xilinx HLS tool for C/C++ designs. Xilinx also provides many IP blocks

for free, including but not limited to on and off-chip-network IPs, memory blocks and memory management IPs, I/O interface IPs, and even various compute IPs. There are also additional IP's that can be purchased from Xilinx or even other vendors and developers.

After the design process is completed, a bitstream can be created and then downloaded to the target FPGA device to run as a standalone hardware device or in combination with firmware running on the FPGA's integrated ARM cores. The device's firmware is developed, compiled, and deployed using the Xilinx Software Development Kit (SDK) tool.

Designs can be tested on IP level or ever system-wide using the IDE's integrated simulator or any other RTL simulator. Moreover, Vivado IDE provides various debug tools that, combined with Integrated Logic Analyzer (ILA) IPs, can scan, check, and visualize the system's behavior during runtime.

6.1.2 Vivado High-Level Synthesis (HLS)

Xilinx Vivado High-Level Synthesis (HLS) [92], currently rebranded as Xilinx Vitis HLS, is a tool included in the Xilinx Vivado Design Suite, allowing for a higher level of abstraction design of HDL systems. Vivado HLS synthesizes C/C++, SystemC and OpenCL functions into IP blocks, generating their VHDL and Verilog HDL designs that can then be implemented into hardware systems using Vivado and its Block Design tool.

While HLS accepts non-hardware-optimized code, it provides a set of directives that can instruct the synthesis procedure to implement a specific behavior, optimization, and resource management. Directives are optional and do not affect the input code's behavior. Their usage can both benefit the generated IP's performance and even hurt it when not used correctly. Furthermore, constraints, like clock period, clock uncertainty, and FPGA target, are added to the HLS synthesized IP blocks to ensure the desired behavior and performance.

A C/C++ testbench is used to debug the input code's behavior prior to synthesis, which should feed the input code with test data and check its output for correctness. Verification of the exported IP block is done using the C/RTL Cosimulation functionality, which uses the same C/C++ testbench, but replaces the function's call with the exported IP block call.

Synthesis Report

A synthesis report is created whenever HLS successfully synthesizes an IP Block, showing various performance and resource utilization metrics. Using the synthesis report, the designer can easily find and target the bottleneck to further optimize their design in terms of both performance and resources. Since Vitis HLS 2020.1, the report summary also contains the aforementioned information per loop per module, making the optimization procedure even more targeted. Some of the metrics are presented and explained below.

- **Latency:** The number of clock cycles required for a complete run of a module or loop.
- **Iteration Latency:** The number of clock cycles required for running a single iteration of a module or loop.
- **Iteration/Initiation Interval (II):** The number of clock cycles required before a module can accept new input or a loop can initiate a new iteration.
- **Pipelined:** Whether a module or loop is implemented using a pipelined architecture.
- **Area:** The number of hardware resources a module requires for its implementation into the target FPGA. The hardware resource types are Block RAM (BRAM) and Ultra RAM (URAM), Digital Signal Processing (DSP) units, Flip Flops (FF), and Lookup Tables (LUT). A table is also given on the detailed report, showing the number of hardware resources required for every hardware component type, which include DSPs, Expressions, First-In-First-Out (FIFO) queues, Instances, Memories, Multiplexers, and Registers.

Optimization Directives

As mentioned above, HLS provides a set of directives for design optimization in terms of latency, throughput, and resource utilization of the exported IP block. Those directives can be added directly into the input code in the form of pragmas that the preprocessor can read. Another way of adding directives is by creating a new solution and automatically adding them to it. Every solution combines a set of directives and configurations into TCL files, one TCL file per solution. Multiple solutions can be created, each with different directive combinations and configurations. This way allows for better

experimentation and fine-tuning of the design. Some optimization directives are presented and explained below.

- **Interface:** The top-level function's arguments have to be mapped to RTL ports to configure the IP block's functionality. The interface directive specifies each argument's port type.
- **Stream:** By default, the top-level function's array arguments are implemented as RAM channels. However, when data are being produced or consumed sequentially, a more efficient data type is to use FIFOs, which can be specified using the stream directive.
- **Pipeline:** Given an *Initiation Interval (II)* parameter, the pipeline directive reduces the number of clock cycles a function or loop can accept new inputs, targeting *II* clock cycles, by allowing the overlapped execution of operations.
- **Unroll:** Given a *factor*, the unroll directive unrolls a loop *factor* times, creating multiple instances of the loop body, that can then be scheduled independently or run in parallel.
- **Loop Flatten:** Allows perfectly nested loops, loops that no logic is injected between them, to get collapsed into a single loop, reducing latency. Essentially, it handles all the indexing logic of the loop flattening.
- **Loop Merge:** Merges consecutive loops, often initialization loops, reducing overall latency and resource utilization.
- **Resource:** Specifies the resource for a variable to get implemented.
- **Array Partition:** By default, every array is implemented as a set of at least one BRAM unit with a single read and a single write port. The array partition directive partitions an array into multiple smaller arrays or assigns each array's element to its register. This partitioning increases the read and write ports of the array on the hardware level, allowing for parallel I/O and computations. In the potential expense of more memory instances and more register, array partitioning can improve overall throughput and performance of memory bounded applications.
- **Array Map:** The array map directive combines multiple small arrays into a single large one, to avoid BRAM waste on small arrays, which can occupy a BRAN unit for just a few elements.

- **Array Reshape:** Reshapes an array of many elements of small bit-width to an array of fewer elements but of higher bit-width, increasing the sequential BRAM access speeds.
- **Data Pack:** Similar to the array reshape directive, the data pack directive combines struct data fields to a single scalar of higher bit-width.
- **Dataflow:** Enables parallel execution of functions and loops, increasing throughput and latency.
- **Inline:** Similar to C/C++ macro preprocessor functionality, the inline directive injects a function's body to each of its calls, reducing latency and initiation interval due to lower function call overhead.
- **Allocation:** Limits the number of hardware resources used for implementing the IP block, and may result in hardware sharing and latency increase.
- **Latency:** Limits the minimum and maximum latency in clock cycles.

6.1.3 Xilinx SDK and Xilinx Vitis IDE

Xilinx Software Development Kit (SDK) [93], currently unified with SDSoC and SDAccel into Vitis Unified Software Platform, is an IDE for embedded-software development Xilinx's microprocessors. Based on the Eclipse IDE [94], it includes a C/C++ editor, a compilation toolchain for ARM microprocessors with automatic Makefile generation, system performance analysis and optimization tools, and several debug and profiling tools. It is used to create applications that run on the ARM cores either external to the FPGA die or internal like on the MPSoCs. Often those applications play the role of the coordinator/master that organizes, schedules, and configures the FPGA hardware. They often handle the external I/O, like data transfers to and from storage devices (SD cards, Hard Drives, Flash Memories, etc.) or Ethernet, to and from volatile memory (RAM, BRAM, etc.). They can also handle the data pre-processing needed to feed the FPGA hardware. Furthermore, multi-core processors can be utilized simultaneously using bare-metal applications. If multi-processing is required with a sophisticated scheduler, Linux applications can be built and run on Linux operating systems like PetaLinux [95] and FreeRTOS [96].

Xilinx SDK is strongly coupled with the Xilinx Design Suite and its hardware designs and bitstreams. After the successful implementation and bitstream

generation of the hardware design from Vivado IDE, Xilinx SDK imports the project's hardware wrapper to generate the Board Support Package (BSP) and various C/C++ libraries useful for communication with and configuration of the FPGA hardware.

Xilinx SDK can create three main types of applications; bare-metal, First Stage Boot Loader (FSBL), and Linux applications. Their main difference is on the way the application is loaded onto the system's processor.

- **Baremetal:** A bare-metal application is loaded using the SDK's built-in functionality that can program the FPGA (PL part) and load it onto the corresponding ARM core through the JTAG port.
- **FSBL:** A FSBL application is a set of files generated by the SDK that, when put on the root folder of the system's primary storage device, e.g., SD card, are read during the system's boot-up, triggering a boot loader sequence. The system has to be appropriately configured, typically configuring some jumpers and switches on development boards to instruct the processor to read the FSBL files. When the bootloader sequence is triggered, programming of the FPGA and loading the application are done using the primary storage device as a source, with no need for an external computer, and Xilinx SDK or JTAG.
- **Linux:** A Linux application is similar to the FSBL one, with the only difference that a Linux operating system is required to be running on the system's processor. Similarly, the Linux OS is loaded using the primary storage device. When Linux is fully loaded, the application can be started like any other Linux application through the provided console window to program the FPGA and run it.

A console window is used for input and output functionality using the UART port in all application types.

Debugging applications is as simple as regular locally running applications using the built-in System Debugger or other debugging tools like GDB [97]. In addition, Vivado IDE's Hardware Manager can be used in combination with SDK's System Debugger to debug hardware designs and their driver applications. Vivado IDE's Hardware Manager connects to the hardware's ILA IPs, enabling the monitoring in real-time of the hardware's state concerning the driver application's state.

Chapter 7

Results

This work's implementation of the proposed platform is described in section 7.2. Its performance metrics, such as its latency, throughput, power, and energy consumption, are compared to the available alternative technologies and FPGA architectures. AlexNet is the selected CNN to be used as a benchmark for the various technologies compared.

7.1 Specifications of the Compared Platforms

The proposed platform is compared with an Intel i7 4710MQ CPU, an NVIDIA RTX 2060 Super 8GB GPU, a Xilinx CHaiDNN implementation, and a Xilinx DPU implementation. All FPGA implementations, including this work's platform, use the Xilinx ZCU102 Evaluation board.

7.1.1 Intel i7 4710MQ

The Intel i7 4710MQ CPU [78], released in 2014, is a mobile processor targeted for high-performance laptops. Its specifications are presented in table 7.1.

TABLE 7.1: Intel i7 4710MQ processor specifications

Cores / Threads	4/8
Max Turbo Frequency	3.5GHz
TDP	47W
Max Memory Bandwidth	25.6GB/s
Lithography	22nm

7.1.2 NVIDIA RTX-2060 Super 8GB

The NVIDIA RTX-2060 Super [98], released in 2019, is a desktop GPU, and while targeted for raytraced gaming, it is also suitable for CNN inferencing due to its large and high-bandwidth memory. Its specifications are presented in table 7.2.

TABLE 7.2: NVIDIA RTX 2060 Super specifications

CUDA Cores	2176
Tensor Cores	32
GPU Memory	8GB GDDR6
Boost Clock	1650 MHz
Memory Interface	256-bit
Memory Bandwidth	448GB/s
Power Consumption	175W

7.1.3 Xilinx CHaiDNN

The Xilinx CHaiDNN accelerator library, presented in section 3.5.1, was implemented for this work's comparisons. The resource utilization for its implementation is depicted in table 7.3. It should be noted that Double Pumped DSPs are used.

TABLE 7.3: Xilinx CHaiDNN resource usage

PL/DSP Clock Frequency	250/500 MHz
LUT Usage	59.1%
FF Usage	27.66%
BRAM Usage	74.12%
DSP Usage	53.65%

7.2 Proposed Platform

This work's proposed platform can be implemented using several data types and accelerators. For this comparison, it was implemented based on AlexNet's requirements and characteristics, presented in chapter 4. Both parameters and activations are represented as 8-bit fixed-points; hence, the accelerators use the same data type. There is a single accelerator instance per layer type, using their high-performance versions as presented in section 5.8. The serial

execution scheduler (see section 5.6) was selected because the implemented accelerators do not support layer pipelining, and there are not multiple instances per layer type.

The resource utilization for implementing the aforementioned configuration of the proposed platform is depicted in table 7.4.

TABLE 7.4: Proposed platform resource usage

Clock Frequency (MHz)	300MHz
LUT Usage	7.34%
LUTRAM Usage	2.05%
FF Usage	4.03%
BRAM Usage	7.51%
DSP Usage	1.9%
BUFG (%)	0.25%

7.3 Performance Metrics

7.3.1 Throughput

Throughput, defined in equation 5.2, is the number of tasks that can be accomplished in a unit time. It is preferred to be as high as possible to generate as much work as possible in the unit time.

7.3.2 Latency

Latency, defined in equation 5.1, is the time required for accomplishing a single task. It is preferred to be as low as possible to finish tasks as quickly as possible from the time they are issued.

7.3.3 Power Consumption

Power consumption is defined as the energy consumed per unit time for accomplishing a specific task, from a chemical reaction and lifting materials using a crane, to emitting light through a light bulb and inferencing CNNs on electronic hardware. Average power consumption is always preferred to be as low as possible to increase the system's energy efficiency, minimizing energy losses. In addition, low power consumption leads to simpler system

designs and lower building costs. It is usually measured in Watts (w) or kilo-Watts (kW).

7.3.4 Energy Consumption

Energy consumption is defined as the energy required for accomplishing a specific task in a specific time amount. It can be calculated as $Energy = Power * Time$, where $Power$ is the required power, and $Time$ is the required time for accomplishing the task. Energy consumption is also preferred to be as low as possible while accomplishing the given task within the time constraints, to minimize the operational costs. It is usually measured in Joule (J) or kiloJoule (kJ).

7.4 CPU and GPU Performance

PyTorch provides the option to select the appropriate input batch size that best suits the application's needs concerning the hardware in use. Every CPU and GPU has different architecture and configuration, and, consequently, the best batch size can vary. In general, larger batch sizes increase both inference throughput and latency. However, very large batch sizes can damage the application's throughput. Hence, a low-latency application should use small batch sizes but expect lower throughput, and high-throughput applications should use large batch sizes with higher latency costs.

A Python script was created to measure the CPU and GPU inference performance regarding the selected batch size. The script uses the prebuilt and pre-trained AlexNet model provided by PyTorch and tests the performance using all the available batch sizes. Figure 7.1 depicts the inference latency on both platforms in milliseconds per image.

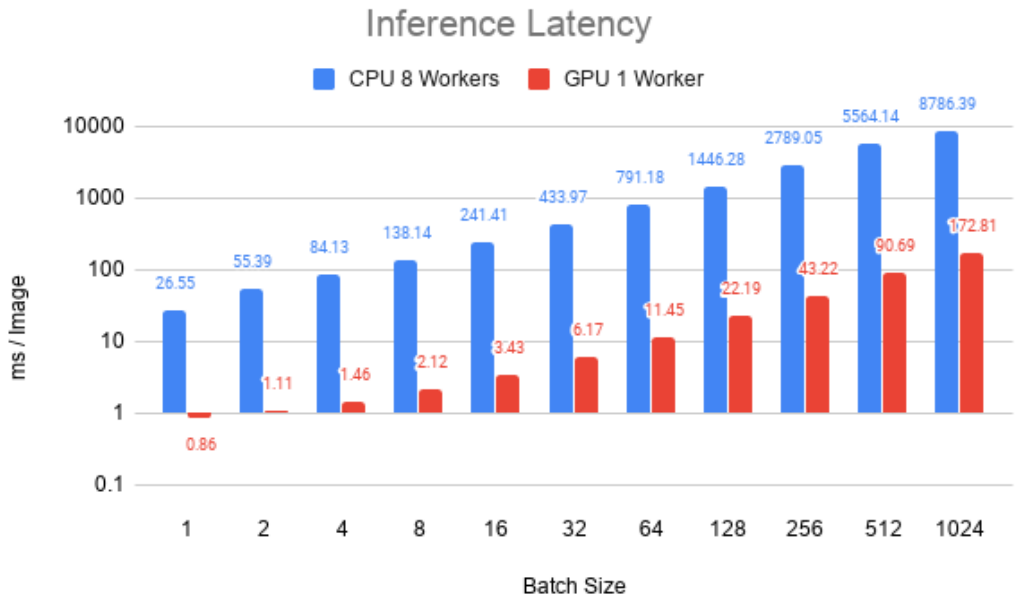

FIGURE 7.1: CPU vs GPU Inference Latency: Latency increases while batch size increases on both platforms.

As expected, the latency increases while the batch size increases. It is remarkable that the GPU achieves almost two orders of magnitude lower latency than the CPU on all batch sizes.

Figure 7.2 depicts the inference throughput on both platforms in images per second. While the CPU's throughput increases as the batch size increases, the GPU's throughput reaches its maximum on 256 batch size, and from then on, it decreases. Almost two orders of magnitude difference between the two platforms are also appearing.

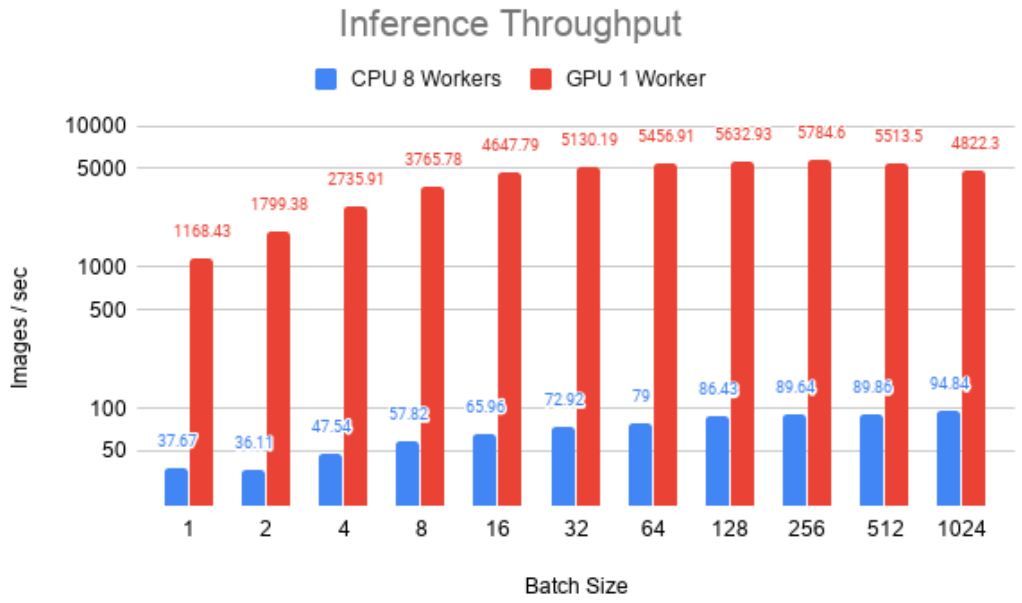

FIGURE 7.2: CPU vs GPU Inference Throughput: Throughput increases while batch size increases on CPU, while the GPU reaches its maximum throughput on 256 batch size.

It should be noted that PyTorch also can configure the number of workers to be used for its inference procedure. A worker can be thought of as an orchestrator, processes that run in parallel to inference images. As a rule of thumb, which has also been tested but not shown on the previous figures, it is best to use the same amount of workers as the number of available threads in a CPU. A single worker should also be used when inferencing with a GPU because multiple workers can create communication bottlenecks and damage performance. Therefore, the aforementioned tests were conducted using eight workers when inferencing on the CPU, and a single worker when inferencing on the GPU.

7.5 Final Performance

The comparisons were conducted using the same dataset and AlexNet hyperparameters across all technologies. The CPU and GPU use floating-point arithmetic for their parameters and activations, while CHaiDNN uses 8-bit quantization, and the proposed platform implementation uses 8-bit fixed-point arithmetic.

Table 7.5 depicts the comparison results of every technology. It should be noted that the CPU and GPU use different batch sizes for their throughput and latency measurements to represent their best performance. In this case, the CPU uses a batch size of 1024 for throughput and a batch size of 1 for latency, and the GPU uses a batch size of 256 for throughput and a batch size of 1 for latency.

The Energy Consumption/Image metric is calculated as shown on equation 7.1.

$$\frac{EnergyConsumption}{Image} = \min\{TotalPower * Latency, \frac{TotalPower}{Throughput}\} \quad (7.1)$$

The Images/Joule metric is calculated, as shown in equation 7.2.

$$\frac{../Images}{Joule} = \max\{\frac{1}{TotalPower * Latency}, \frac{Throughput}{TotalPower}\} \quad (7.2)$$

The throughput and latency speedups, and the power and energy efficiencies for every platform are calculated compared to the CPU.

TABLE 7.5: Performance results

	CPU	**GPU**	**CHaiDNN**	**Proposed Platform**
Clock Frequency (MHz)	3500	1650	250/500	300
Throughput (Images/s)	94.84	5784.6	10.07	0.0927
Throughput Speedup	1x	60.9933x	0.1062x	0.001x
Latency (s)	0.0266	0.0009	0.0993	10.783
Latency Speedup	1x	29.5556x	0.2679x	0.0025x
Total On-Chip Power (Watt)	47	175	19.3	4.559
Power Efficiency	1x	0.2686x	2.4352x	10.3093x
Energy Cons./Image (Joule)	1.2502	0.1575	1.9165	49.1597
Energy Efficiency	1x	7.9378x	0.6523x	0.0254x
Images/Joule	2.0179	33.0549	0.5218	0.0203

The final results of the various performance metrics are also depicted using bar charts in figure 7.3 for better visibility.

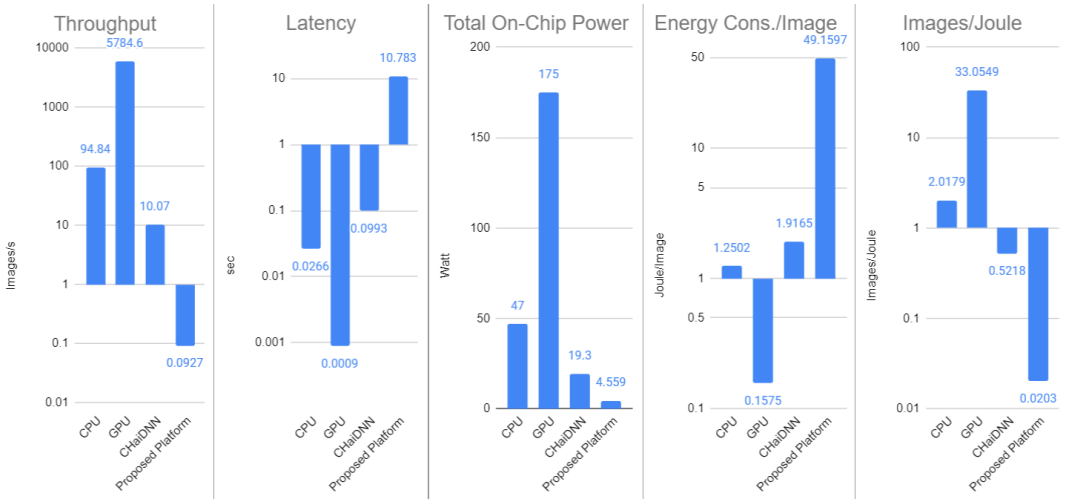

FIGURE 7.3: Final Results Charts: While requiring the highest amount of power, the GPU stands out in every other performance metric.

It can be observed that while the GPU in use requires almost four times the power of its CPU counterpart, it can provide the best performance in every metric, with an almost 61 times throughput speedup and an almost 30 times latency speedup. Although the GPU is the most power-hungry device, it achieves the energy efficiency across all platforms due to its low latency and high throughput capabilities.

Both tested FPGA platforms, the CHaiDNN and the proposed platform, have worse results in every metric compared to the CPU and GPU. However, CHaiDNN and the proposed platform beat their counterparts with more than 2 times and 10 times lower power requirements, respectively. Between the two FPGA platforms, CHaiDNN performs better in terms of throughput, latency, and energy consumption, but requires more than 4 times the proposed platform's power and much more hardware resources, as shown in tables 7.3 and 7.4.

Although the proposed platform's performance is lower than expected for a real-world application, this significant difference in resource utilization between the two platforms creates an opportunity for further development to achieve better results. The priority for further development should be given to the Convolution accelerator, which consumed about 98% of the total inference time in the tests. First of all, the Double-Pumped DSPs technique used by both Xilinx CHaiDNN and Xilinx DPU should also be used by the

proposed platform to create a speedup of up to 2 times. Secondly, the convolution algorithm should be further unrolled to convolve in a single clock cycle the whole kernel, not just row-by-row like the convolution accelerator shown in figure 5.11, or even multiple kernels, striding from left to right. Furthermore, the Max-Pooling accelerator's functionality could be integrated into the Convolution accelerator, applying its operations after creating a single output channel and before the ReLU functionality. This way, the ReLU is also applied to fewer outputs due to the Max-Pooling's down-sampling nature. Additionally, communication is decreased because there is no need for the convolution results to be transferred to the Max-Pooling accelerator, either by MMIO or stream, since they are passed-through directly from the same BRAM instances. Finally, other architectures could also be designed, such as by incorporating a systolic array, and even architectures utilizing multiple FPGA devices.

It is also noteworthy that the RTX 2060 Super is a fairly new GPU, at the time of writing, launched in the mid 2019, while the ZCU102's part, the Xilinx Zynq Ultrascale+ ZU9, launched in 2015, more than 4 years earlier. Newer FPGA parts provide better performance out-of-the-box, with lower power consumption due to better chip lithography, and a lot more hardware resources. Hence, a fairer comparison between a GPU and it corresponding FPGA should be conducted using a modern FPGA part.

Chapter 8

Conclusions and Future Work

In this chapter, this thesis' work is being summed up and evaluated. Also, directions for future work, possible extensions, and optimizations are being given.

8.1 Conclusions

Over the last few years, Convolutional Neural Networks have proved to be capable of tackling complex image recognition problems and sound recognition, security, and data mining problems. The research community continues to surprise the world with new and paradoxical use cases for CNNs, with even more exciting results. With the rise of neural networks, hardware capable of handling high computational complexity in a fast and energy-efficient manner becomes necessary.

This thesis' purpose was to create an FPGA accelerator for CNN inference, using AlexNet as the base network and benchmark. However, a whole platform was created for easy and structured implementation of such accelerators not only for CNNs but neural networks in general. The implementation of this thesis' proposed platform is used to accelerate AlexNet's inference, whose robustness analysis was carried out to investigate the FPGA's strengths and weaknesses. Computational workloads, memory access patterns, memory and bandwidth reduction, as well as algorithmic optimizations, were studied to exploit the FPGA's parallelism capabilities and strengths.

While the proposed platform's implementation was based on the Xilinx ZCU102 Evaluation Kit, it can be transferred and scaled accordingly to other FPGA devices, such as the FORTH QFDB, a custom four-FPGA platform. Although there were no performance benefits using an FPGA over an NVIDIA RTX-2060 Super GPU, a potential for performance improvements appears with

further development, focusing on the convolution accelerator, which exploits the platform's ease of use, extendability, and expandability.

8.2 Future Work

This thesis' proposed platform is by design easily expandable for future use and development, creating several opportunities for its expansion and CNN accelerators' optimization. Some of them are presented below.

- Quantization techniques for both parameters and activations should be further investigated to achieve better classification accuracy using 8-bit or even lower representations. Techniques such as K-Means clustering, Lloyd's, Pair and Quad compression, and SLC found in George Pitsis' thesis [99] could be a great start.
- Similar to integrating the ReLU activation function into the Convolutional and Fully-Connected layers' accelerators, integrating the pooling layer into the convolutional layer's accelerator could be beneficial both latency and throughput by reducing the network's overall memory I/O and avoiding the separate accelerator's initialization.
- The platform's scalability should be exploited not only by implementing it in bigger FPGA devices but also in multiple interconnected FPGAs using platforms such as the FORTH QFDB or CRDB. Multiple accelerator instances could be incorporated using such platforms, creating opportunities for higher throughput and lower latency, as well as new scheduler strategies that might not be presented in the FPGA Implementation chapter.
- Pruning enabled accelerators could bring not only lower latency and high throughput but also higher energy efficiency since there are less required operations for a single inference.
- There are works, such as the Xilinx DPU, which use systolic arrays as their main compute engine. Systolic arrays, while being relatively complicated, could improve latency and memory bandwidth due to their design. They could be used to implement a matrix multiplier for the convolutional layer's operation requirements. Although implementing variable padding and stride is not an obvious task, it should be feasible with careful data scheduling. Also, systolic arrays could be designed to

implement n-dimensional convolutions to expand the accelerator's use cases.

- Monte Carlo Dropout during inference could also be consolidated to increase the confidence of the classification results. Multiple instances of the same network could be run with the same inputs in parallel, using multiple accelerator instances and even multiple FPGA devices. Weight could be zeroed out randomly on each iteration in hardware using a linear feedback shift register as a random number generator.
- Layer-Pipelining, as presented in section 5.6, could further decrease the network's overall latency. While implementing it as presented is a complex task, it could be simplified by implementing a memory address generator that produces addresses in the specified order.
- During some experimentation with Xilinx's tools, it was observed that implementing the same functionality designs using pure VHDL and Xilinx HLS leads to very different performance and especially resource utilization. Although implementing a design using pure VHDL requires much more working hours on its development and testing compared to using Xilinx HLS, it could create better performance results and new opportunities.
- CPU-FPGA partitioning should also be further studied to exploit the CPU's higher clock speeds, avoiding wasting FPGA resources for tasks that CPUs can already handle. In the case of Xilinx ZCU102, all six cores could be utilized to contribute to the overall network inference.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[6] F. Chaix et al. "Implementation and Impact of an Ultra-Compact Multi-FPGA Board for Large System Prototyping". In: *IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)* 3 (2019), pp. 34–41. URL: https://ieeexplore.ieee.org/document/8945720.

[8] Sasanka Potluri. "CNN based high performance computing for real time image processing on GPU". In: (2011). URL: https://ieeexplore.ieee.org/document/6024781/.

[9] Bo Chen-Dmitry Kalenichenko Weijun Wang Tobias Weyand Marco Andreetto Hartwig Adam Andrew G. Howard Menglong Zhu. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: (2011). URL: https://arxiv.org/pdf/1704.04861.

[10] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3.3 (July 1959), pp. 210–229. URL: https://ieeexplore.ieee.org/document/5392560.

[16] Sepp Hochreiter. "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6 (Apr. 1998), pp. 107–116. DOI: 10.1142/S0218488598000094. URL: https://www.researchgate.net/publication/220355039_The_Vanishing_Gradient_Problem_During_Learning_Recurrent_Neural_Nets_and_Problem_Solutions.

[17] Mohammad Rastegari et al. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks". In: *CoRR* abs/1603.05279 (2016). URL: http://arxiv.org/abs/1603.05279.

[20] Christian Szegedy et al. "Going Deeper with Convolutions". In: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842.

[21] Ciresan et al. *Multi-column Deep Neural Networks for Image Classification*. 2012. URL: https://arxiv.org/pdf/1202.2745.pdf.

[22] Andrew D. Back Lawrence Steve; C. Lee Giles; Ah Chung Tsoi. *Face Recognition: A Convolutional Neural Network Approach*. 1997. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.92.5813.

[23] Matusugu et al. *Subject independent facial expression recognition with robust face detection using a convolutional neural network*. 2011. URL: https://www.sciencedirect.com/science/article/pii/S0893608003001151?via\%3Dihub.

[24] Microsoft. *Learning Semantic Representations Using Convolutional Neural Networks for Web Search – Microsoft Research*. 2015. URL: https://www.microsoft.com/en-us/research/publication/learning-semantic-representations-using-convolutional-neural-networks-for-web-search/?from=http\%3A\%2F\%2Fresearch.microsoft.com\%2Fapps\%2Fpubs\%2Fdefault.aspx\%3Fid\%3D214617.

[25] Ronan Collobert and Jason Weston. "A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning". In: *Proceedings of the 25th International Conference on Machine Learning*. ICML '08. Helsinki, Finland: Association for Computing Machinery, 2008, 160–167. ISBN: 9781605582054. DOI: 10.1145/1390156.1390177. URL: https://doi.org/10.1145/1390156.1390177.

[26] Y. Bengio. *Practical recommendations for gradient-based training of deep architectures*. 2012. URL: https://arxiv.org/abs/1206.5533.

[27] D. E. Rumelhart and J. L. McClelland. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. MITP, 1987, pp. 318–362. URL: https://ieeexplore.ieee.org/document/6302929.

[28] Yann Lecun. "Generalization and network design strategies". English (US). In: *Connectionism in perspective*. Ed. by R. Pfeifer et al. Elsevier, 1989. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-89.pdf.

[29] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2 (1991), pp. 251 –257. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(91)90009-T. URL: http://www.sciencedirect.com/science/article/pii/089360809190009T.

[33] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG]. URL: https://arxiv.org/abs/1708.07747.

[37] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2014. arXiv: 1405.0312 [cs.CV]. URL: https://arxiv.org/abs/1405.0312.

[41] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* (2017), 84—90. URL: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[42] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. URL: https://ieeexplore.ieee.org/document/726791.

[43] Matthew D Zeiler and Rob Fergus. *Visualizing and Understanding Convolutional Networks*. 2013. arXiv: 1311.2901 [cs.CV]. URL: https://arxiv.org/abs/1311.2901.

[44] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. arXiv: 1409.1556 [cs.CV]. URL: https://arxiv.org/abs/1409.1556.

[45] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[48] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *CoRR* abs/1408.5093 (2014). arXiv: 1408.5093. URL: https://arxiv.org/abs/1408.5093.

[53] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *CoRR* abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467.

[60] Sharan Chetlur et al. "cuDNN: Efficient Primitives for Deep Learning". In: *CoRR* abs/1410.0759 (2014). arXiv: 1410.0759. URL: http://arxiv.org/abs/1410.0759.

[64] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *CoRR* abs/1704.04760 (2017). arXiv: 1704.04760. URL: http://arxiv.org/abs/1704.04760.

[71] Xilinx. *PG338 - Zynq DPU v3.2 IP Product Guide (v3.2)*. May 2020. URL: https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_2/pg338-dpu.pdf.

[79] Sergei Arthur and David Vassilvitskii. "Dynamically scaled fixed point arithmetic". In: (1991). URL: https://ieeexplore.ieee.org/document/160742/.

[81] Xilinx. *UG1037 - Vivado Design Suite: AXI Reference Guide*. July 2017. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[82] Xilinx. *PG247 - SmartConnect v1.0 Product Guide (v1.0)*. Feb. 2020. URL: https://www.xilinx.com/support/documentation/ip_documentation/smartconnect/v1_0/pg247-smartconnect.pdf.

[83] Xilinx. *PG021 - AXI DMA v7.1 Product Guide (v7.1)*. June 2019. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[84] Xilinx. *PG034 - AXI Central Direct Memory Access v4.1 Product Guide (v4.1)*. Apr. 2018. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf.

[85] Xilinx. *PG078 - AXI Block RAM (BRAM) Controller v4.1 Product Guide (v4.1)*. May 2019. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_1/pg078-axi-bram-ctrl.pdf.

[86] Xilinx. *PG058 - Block Memory Generator v8.4 Product Guide (v8.4)*. Dec. 2019. URL: https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf.

[87] Xilinx. *PG085 - AXI4-Stream Infrastructure IP Suite v3.0 Product Guide (v3.0)*. Dec. 2018. URL: https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf.

[89] David P. Rodgers. "Improvements in Multiprocessor System Design". In: *SIGARCH Comput. Archit. News* 13.3 (June 1985), 225–231. ISSN: 0163-5964. DOI: 10.1145/327070.327215. URL: https://doi.org/10.1145/327070.327215.

[99] Antonios-Georgios Pitsis. "Design and implementation of an FPGA-based convolutional neural network accelerator". Diploma Thesis. Electrical and Computer Engineering, Technical University of Crete, 2018. URL: https://dias.library.tuc.gr/view/79092.

External Links

[2] *Forbes - How Much Data Is Collected Every Minute Of The Day*. Aug. 2019. URL: https://www.forbes.com/sites/nicolemartin1/2019/08/07/how-much-data-is-collected-every-minute-of-the-day/#747555a33d66.

[3] *AMD EPYC 7002 Series Processors*. Feb. 2020. URL: https://www.amd.com/en/processors/epyc-7002-series.

[4] *NVIDIA Titan RTX GPU*. Feb. 2020. URL: https://www.nvidia.com/en-us/deep-learning-ai/products/titan-rtx/.

[5] *Google Cloud TPU*. Feb. 2020. URL: https://cloud.google.com/tpu/docs/system-architecture.

[7] *Foundation of Research and Technology Hellas (FORTH)*. URL: https://www.forth.gr/.

[11] *Machine learning - Wikipedia*. Sept. 2019. URL: https://en.wikipedia.org/wiki/Machine_learning.

[12] *Machine Learning – Applications*. URL: https://www.geeksforgeeks.org/machine-learning-introduction/.

[13] *Top Machine Learning Applications in 2019*. 2019. URL: https://www.geeksforgeeks.org/top-machine-learning-applications-in-2019/.

[14] *Roundup Of Machine Learning Forecasts And Market Estimates*. Feb. 2018. URL: https://www.forbes.com/sites/louiscolumbus/2018/02/18/roundup-of-machine-learning-forecasts-and-market-estimates-2018/#536446aa2225.

[15] *Activation Function - Wikipedia*. Mar. 2020. URL: https://en.wikipedia.org/wiki/Activation_function.

[18] *Types of Artificial Neural Networks - Wikipedia*. Mar. 2020. URL: https://en.wikipedia.org/wiki/Types_of_artificial_neural_networks.

[19] *Convolutional Neural Networks - Wikipedia*. Mar. 2020. URL: https://en.wikipedia.org/wiki/Convolutional_neural_network.

[30] *Udacity Intro to Deep Learning with PyTorch by Facebook AI*. URL: https://www.udacity.com/course/deep-learning-pytorch--ud188.

[31] *The MNIST database of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/.

[32] *MNIST database - Wikipedia*. URL: https://en.wikipedia.org/wiki/MNIST_database.

[34] *Fashion-MNIST - Github*. URL: https://github.com/zalandoresearch/fashion-mnist.

[35] *CIFAR-10 and CIFAR-100*. URL: http://www.cs.toronto.edu/~kriz/cifar.html.

[36] *CIFAR-10 and CIFAR-100 - Wikipedia*. URL: https://en.wikipedia.org/wiki/CIFAR-10.

[38] *Microsoft COCO*. URL: http://cocodataset.org/#home.

[39] *ImageNet Official site*. URL: http://image-net.org/index.

[40] *ImageNet - Wikipedia*. URL: https://en.wikipedia.org/wiki/ImageNet.

[46] François Chollet et al. *Keras - Official site*. 2015. URL: https://keras.io/.

[47] *Keras - Wikipedia*. URL: https://en.wikipedia.org/wiki/Keras.

[49] *Caffe - Official site*. URL: https://caffe.berkeleyvision.org/.

[50] *Caffe - Wikipedia*. URL: https://en.wikipedia.org/wiki/Caffe_(software).

[51] *PyTorch - Official site*. URL: https://pytorch.org/.

[52] *PyTorch - Wikipedia*. URL: https://en.wikipedia.org/wiki/PyTorch.

[54] *TensorFlow - Official site*. URL: https://www.tensorflow.org/.

[55] *TensorFlow - Wikipedia*. URL: https://en.wikipedia.org/wiki/TensorFlow.

[56] *Google Just Open Sourced TensorFlow, Its Artificial Intelligence Engine - Wired*. URL: https://www.wired.com/2015/11/google-open-sources-its-artificial-intelligence-engine/.

[57] *Advanced Vector Extensions - Wikipedia*. URL: https://en.wikipedia.org/wiki/Advanced_Vector_Extensions.

[58] *Streaming SIMD Extensions - Wikipedia*. URL: https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions.

[59] *NVIDIA CUDA*. URL: https://developer.nvidia.com/cuda-zone.

[61] *NVIDIA cuDNN*. URL: https://developer.nvidia.com/cudnn.

[62] *NVIDIA TensorRT*. URL: https://developer.nvidia.com/tensorrt.

[63] *NVIDIA PyCUDA*. URL: https://developer.nvidia.com/pycuda.

[65] *Google opens up about its Tensor Processing Unit*. URL: https://www.datacenterdynamics.com/news/google-opens-up-about-its-tensor-processing-unit/.

[66] *Coral Edge TPU*. URL: https://coral.ai/.

[67] *An in-depth look at Google's first Tensor Processing Unit (TPU)*. URL: https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[68] *Tensor Processing Unit - Wikipedia*. URL: https://en.wikipedia.org/wiki/Tensor_processing_unit.

[69] *Cloud Tensor Processing Units*. URL: https://cloud.google.com/tpu/docs/tpus.

[70] *CHaiDNN - A HLS-based Deep Neural Network Accelerator library for Xilinx Ultrascale+ MPSoC devices*. URL: https://github.com/Xilinx/CHaiDNN.

[72] *Xilinx Vitis AI*. URL: https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html.

[73] *NVIDIA NVDLA*. URL: http://nvdla.org/.

[74] *MATLAB*. URL: https://www.mathworks.com/.

[75] *Kaggle*. URL: https://www.kaggle.com/.

[76] *Xilinx ZCU102 User Guide - UG1182*. URL: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf.

[77] *Xilinx ZCU102 - Product Overview*. URL: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[78] *Intel i7 4710MQ Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/78931/intel-core-i7-4710mq-processor-6m-cache-up-to-3-50-ghz.html.

[80] *Synaptic Pruning - Wikipedia*. URL: https://en.wikipedia.org/wiki/Synaptic_pruning.

[88] *Network Topology - Wikipedia*. URL: https://en.wikipedia.org/wiki/Network_topology.

[90] *Vivado Design Suite - HLx Editions*. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[91] *Xilinx ISE*. URL: https://www.xilinx.com/products/design-tools/ise-design-suite.html.

[92] *Vivado Design Suite User Guide: High-Level Synthesis - UG902*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf#nameddest=xApplyingOptimizationDirecti

[93] *Xilinx Software Development Kit (SDK)*. URL: https://www.xilinx.com/products/design-tools/embedded-software/sdk.html.

[94] *Eclipse IDE*. URL: https://www.eclipse.org/.

[95] *PetaLinux*. URL: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html.

[96] *FreeRTOS*. URL: https://www.freertos.org/.

[97] *GDB*. URL: https://www.gnu.org/software/gdb/.

[98] *NVIDIA RTX 2060 Super*. URL: https://www.nvidia.com/en-eu/geforce/graphics-cards/rtx-2060-super/.