Technical University of Crete

Diploma Thesis

Large Scale Design and Implementation of Convolutional Neural Networks based on Large FPGA Arrays

Author:
Charisios Loukas

Thesis Committee:
Prof. Apostolos Dollas
Prof. Michalis Zervakis
Prof. Eftichios Koutroulis

A thesis submitted in fulfillment of the requirements
for the DIPLOMA of Electrical and Computer Engineering
in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Lab

January 17, 2020

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

DIPLOMA THESIS

Large Scale Design and Implementation of Convolutional Neural Networks based on Large FPGA Arrays

by Charisios Loukas

An international race towards the development of the first exascale supercomputer during the past few years, has created a demand for applications that require comparable amounts of computational work. The superior performance and power consumption that hardware accelerators can achieve, renders their use on such applications inevitable. Convolutional Neural Networks constitute a prime example of a computationally intensive and highly parallelizable system, whose performance can increase significantly, when implemented as such an accelerator, as recent work has shown. This study inherits the hardware accelerator design of a CNN developed by G. Pitsis[37] and attempts to scale it, both horizontally, by incorporating it into the ExaNeSt designs and allowing its use on the QFDB multi-FPGA prototype board, and vertically, by fixing issues with a version of the design that increases the batch size. It also applies a recently published dropout technique on the hardware accelerator at prediction time, demonstrating the capability of trading computational power for increased confidence in the results of the network. The boards used for the purposes of this thesis are the Xilinx ZCU102 and the QFDB, a 4-FPGA prototype board developed in FORTH.

This work is partially supported by the European Union Horizon 2020 program through the EuroExa project.

TECHNICAL UNIVERSITY OF CRETE

Abstract

DIPLOMA THESIS

Σχεδίαση και Υλοποίηση Μεγάλης Κλίμακας CNN βασισμένων σε Ευρείας Κλίμακας Συστήματα Αναδιατασσόμενης Λογικής

by Charisios Loukas

Η διεθνής προσπάθεια για την ανάπτυξη του πρώτου υπερυπολογιστή με δυνατότητα εκτέλεσης πράξεων κλίμακας ενός πεντάκις εκατομμυρίου, έχει δημιουργήσει ζήτηση για εφαρμογές που απαιτούν αντίστοιχο υπολογιστικό φόρτο για την εκτέλεση τους. Η υπερέχουσα επίδοση και κατανάλωση ενέργειας που επιτυγχάνουν οι επιταχυντές υλικού, καθιστά την χρήση τους σε τέτοιες εφαρμογές αναπόφευκτη. Τα Συνελικτικά Νευρωνικά Δίκτυα αποτελούν άριστο παράδειγμα ενός υπολογιστικά εντατικού και άκρως παραλληλοποιήσιμου συστήματος, του οποίου η επίδοση βελτιώνεται σημαντικά, με τη υλοποίηση του σαν επιταχυντή υλικού, όπως έχουν δείξει πρόσφατη βιβλιογραφία. Η παρούσα δουλειά κληρονομεί την σχεδίαση υλικού του επιταχυντή ενός ΣΝΔ, υλοποιημένη από τον Α. Γ. Πίτση και επιχειρεί να την κλιμακώσει, τόσο οριζοντίως, με την ενσωμάτωση της στις σχεδιάσεις του ερευνητικού προγράμματος ExaNeSt και χρήση στην πρωτότυπη πλατφόρμα QFDB, όσο και καθέτως, με την αποσφαλμάτωση μιας εκδοχής της σχεδίασης που αυξάνει τον αριθμό δεσμίδων δεδομένων της εισόδου. Επιπλέον, εφαρμόζουμε μια προσφάτως δημοσιευμένη τεχνική απενεργοποίησης νευρώνων κατά τη διάρκεια της εκτέλεσης του δικτύου, επιδεικνύοντας της δυνατότητα βελτίωσης της εμπιστοσύνης στα αποτελέσματα του δικτύου, θυσιάζοντας επεξεργαστική ισχύ. Οι πλατφόρμες που χρησιμοποιήθηκαν στα πλαίσια αυτής της διπλωματικής εργασίας ήταν οι ZCU102 της Xilinx και το QFDB, το πρωτότυπο που αναπτύχθηκε από το ΙΤΕ.

Η δουλειά αυτή υποστηρίζεται εν μέρει από το ερευνητικό πρόγραμμα Horizon 2020 της Ευρωπαϊκής Ένωσης μέσω του προγράμματος EuroExa.

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Apostolos Dollas, for his constant support and guidance both during my work on this thesis and throughout the course of my studies, as well as for giving me the opportunity to be part of the advanced scientific work taking place in FORTH.

Furthermore, I would like to thank the CARV team in FORTH, for the excellent collaboration we had throughout the course of this work. Especially my fellow student, now PHD candidate, Giorgos Pitsis, for his help in understanding his exceptional work, which was the motivation and starting point for this thesis and PHD candidate Angelos Ioannou, for the time he devoted on guiding me through his work on the Euroexa project. Furthermore I would like to thank Dr. Fabien Chaix for the effort he put into solving any issues that came up with the QFDB. I would also like to express my deepest gratitude to Dr. Christos Kozanitis, for his guidance throughout this endeavour and to Dr. Gregory Tsagatakis for his advice on the mathematical representation of my results.

This thesis could of course not have been completed wihtout the valuable help of the staff of the MHL laboratory in TUC. Most of all I would like to thank Dr. Andreas Brokalakis for his constant guidance over the use of the server and its tools and PHD candidate Pavlos Malagonakis for valuable advice and his insightful input on my work. I would also like to thank Dr. Kostas Georgopoulos for his help with the QFDB and Dr. Dimitris Theodoropoulos for sharing his deep knowledge of the Xilinx design tools. Moreover I would like to thank my fellow student Giorgos Foteinopoulos for his advice and support.

Last but not least i would like to express my deepest gratitude to the people who have always been there for me, my family and friends.

Charisios Loukas,
Chania 2019

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

CNN	**C**onvolutional **N**eural **N**etwork
NN	**N**eural **N**etwork
FC	**F**ully **C**onnected
ReLU	**Re**ctified **L**inear **U**nit
ML	**M**achine **L**earning
HPC	**H**igh **P**erformance **C**omputing
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
CPU	**C**entral **P**rocessor **U**nit
GPU	**G**raphic **P**rocessor **U**nit
TPU	**T**ensor **P**rocessor **U**nit
APU	**A**ccelerated **P**rocessor **U**nit
RPU	**R**eal-time **P**rocessor **U**nit
PS	**P**rocessing **S**ystem
PL	**P**rogrammable **L**ogic
OS	**O**perating **S**ystem
QFDB	**Q**uad **F**PGA **D**aughter **B**oard
CRDB	**C**o-design **R**ecommended **D**aughter **B**oard
HLS	**H**igh **L**evel **S**ynthesis
SDK	**S**oftware **D**evelopment **K**it
RAM	**R**andom **A**ccess **M**emory
D-RAM	**D**ynamic **R**andom **A**ccess **M**emory
B-RAM	**B**lock **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
LUT	**L**ook **U**p **T**able
DDR	**D**ouble **D**ata **R**ate
IP	**I**ntellectual **P**roperty
AXI	**A**dvanced e**X**tensible **I**nterface
FPD	**F**ull **P**ower **D**omain
LPD	**L**ow **P**ower **D**omain

DMA	**D**irect **M**emory **A**ccess
RTL	**R**egister **T**ransfer **L**evel
BSP	**B**oard **S**upport **P**ackage
HSSL	**H**igh **S**peed **S**erial **L**ink
MPI	**M**essage **P**assing **I**nterface
QSPI	**Q**uad **S**erial **P**eripheral **I**nterface
JTAG	**J**oint **T**est **A**ction **G**roup
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter
PCIe	**P**eripheral **C**omponent **I**nterface **e**xpress
SODIMM	**S**mall **O**utline **D**ual **I**n-line **M**emory **M**odule
LVDS	**L**ow **V**oltage **D**ifferential **S**ignaling

Dedicated to my family and friends...

Chapter 1

Introduction

In the recent years a massive effort has been directed towards achieving the development of a supercomputer with exascale capabilities, by states all over the world, separately or as part of scientific cooperation. The advent of exascale computers will usher in a new era in computation, with supercomputers gaining the ability to run vastly larger simulations, such as accurate weather prediction in a global scale[8], search for new molecule structures[35] and even replicate the activities of a human brain at the neuron level[42]. To utilize the capabilities of an exascale supercomputer we need applications that can consume the vast amounts of data at least as fast as the computer can provide them, a task which is challenging on its own. Convolutional Neural Networks, which are the area of focus of this thesis, constitute such an application, being by design computationally intensive and highly parallelizable.

1.1 Motivation

The most significant characteristics of an exascale application are performance, power consumption and scalability. We need good performance in terms of both throughput and latency, in order to utilize the full potential of an exascale supercomputer for lightning-fast communication and computation of large amounts of data. We also need low power consumption, since cooling is a fundamental problem in high performance computing and scalability in order to allow for the parallelization of our computations, making the application suitable for use in a supercomputer. The work of George Pitsis[37] produced notable results in all the aforementioned aspects, thus becoming the main motivation behind this work.

Part of this work, also deals with machine learning techniques, that improve the confidence on the results of a neural network. Such techniques have produced promising results in software implementations of neural networks. Motivated by the work of arin Gal and Zoubin Ghahramani[19], we will attempt to apply Monte Carlo Dropout in a hardware implementation of a convolutional neural network.

1.2 Scientific contributions

The scientific contribution of this thesis revolves around 2 main axes. The first is the use of weight dropout during testing in neural networks as approximate Bayesian inference to model the networks' uncertainty. Standard deep learning tools for regression and classification do not capture model uncertainty. More specifically, convolutional neural networks like the one we will study, use the softmax function to produce the predictive probabilities of each class, which can be falsely misinterpreted as model confidence. To obtain a measure of confidence in the networks results, we use Monte Carlo weight dropout in a trained NN as a Bayesian approximation of a Gaussian stochastic process, with the outputs of the network being the random variables. We can then use the confidence to either evaluate the effectiveness of our classifier or to sort out high uncertainty inputs that need to be classified by a human. This has already been proven effective in software implementations of NNs. Our approach will use the hardware accelerator to test the technique.

The main area of focus of this thesis however, was the large scale design and implementation of a convolutional neural network on a multiple FPGAs, using the hardware design of G. Pitsis[37] as a paradigm. The main contribution of this thesis is the incorporation of this design in the ExaNeSt[13] designs targeted at the Multi-FPGA QFDB prototype board. This achievement gives us the capability of programming multiple QFDB boards with the CNN design, effectively making the scaling of the application possible. The final objective was the use of the ethernet infrastructure of the board to stream data into the accelerators. This objective has not been achieved for various technical reasons, since the software part of this task could not be completed in time, for various technical reasons which we will present. An additional contribution of this work to this area was the debugging of a preexisting dysfunctional CNN design, that increased the batch size of the neural network. This results in a higher degree of parallelism that is also useful when trying to scale an application.

1.3 Thesis Outline

In this section we present an outline of the organisation of this thesis:

- **Chapter 2:** We describe the structure of CNNs and present a number of exascale supercomputer prototypes where they could be used successfully as applications.

- **Chapter 3:** We describe the computational models of the two CNNs used either in the models or the hardware designs of this thesis.
- **Chapter 4:** We present the multi-node system models we created and the results of a Monte-Carlo dropout technique as it was applied in an existing CNN hardware design.
- **Chapter 5:** We describe step-by-step the systemic work that was done to fix the fix an existing architecture and incorporate another in the ExaNeSt project to allow streamingof input values through an ethernet link.
- **Chapter 6:** We present the resource utilization and performance results of this work.
- **Chapter 7:** We conclude this thesis and propose a number of topics suitable for further development.

Chapter 2

Related Work

> One of their most compelling conclusions is that with exascale computing, we are reaching a tipping point in predictive science, an area with potentially major implications across a whole range of new, massively complex problems.
>
> *Irving Wladawsky-Berger*

In this chapter we will present an overview of the different approaches to the design of an exascale supercomputer and the applications such a computer can have on the area of focus of this thesis, namely neural networks.

First we will describe how a convolutional Neural Network (CNN) functions, the modules from which it is made up and how they are structured, followed by a mathematical representation. We will also present the data that are are needed for the computations in every module, their size and when they have to be available to the network. This gives us a basis for the requirements the target platform must fulfill in order to perform optimally in the computations requirements.

In the second part of this chapter we will describe what an exascale computer is and present the massive efforts of numerous research programs around the world to develop such a system. We will also give a brief outline of the architectures of the exascale prototypes designed by these programs and subsequently argue why the platform which was used for the purposes of this thesis is the most suitable for the type of computations that are required for a CNN to function.

Finally we will summarize how this thesis approaches the problem of scaling out a Neural Network application in a large scale multi-node computing system.

2.1 Structure of a Convolutional Neural Network

A typical Convolutional Neural Network[18][41] consists of millions of neurons arranged in several layers. First we have a number convolutional layers separated or not (depending on the problem) by pooling layers. Then we have several fully Connected layers, called hidden layers and finally we have one last fully Connected layer, which is our classifier. We can se the architecture of such a network in figure 2.1

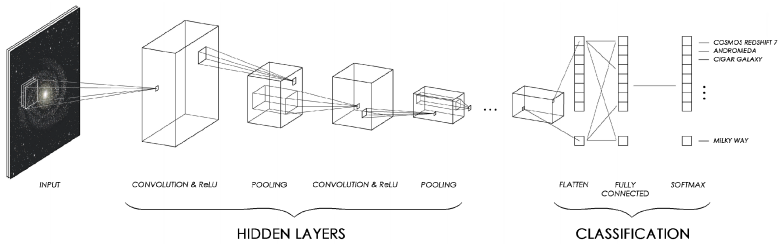

FIGURE 2.1: Architecture of a Convolutional Neural Network
Picture from G. Pitsis thesis

2.1.1 Convolutional Layer

convolutional layers apply a convolution operation to the input, using a kernel matrix and passing the result to the next layer. The number of convolutional layers used in a convolutional Neural Network depends on the complexity of the problem, as a deeper network architecture might be able to extract more local features and thus more detailed characteristics from training examples. Since however matrix multiplication preserves linearity, we need to introduce some form of non-linearity in the form of an activation function f, between the convolutional Layers, to be able to solve complex problems. Typical choices for such a function (know as activation function) are the logistic (sigmoid) function, the hyperbolic function or the Rectified Linear Unit (ReLU) function, which is the most common choice for CNNs, as it is easier to compute, thus rendering the training process much faster and less likely to suffer from the problem of vanishing gradients. The data needed for the computations of a convolutional layer are the kernel matrix and the stride, which is the number of pixels the kernel window will slide after each computation. Assuming a 2 dimensional input[26], let k be the kernel matrix, x the input and y the output of the convolutional layer, we have:

$$y(i,j) = f(\sum_{-a}^{a}\sum_{-b}^{b} k(s,t)x(i-s,j-t)) \text{ with } -a \leq s \leq a,\ -b \leq t \leq b \qquad (2.1)$$

The number of consecutive summation that we have to compute depends on the dimensions of the input data. For example for 2-D a input we compute a double sum and so forth.

2.1.2 Pooling Layer

Pooling layers are applied between convolutional ones to reduce the size of the representation, but not the depth, thus increasing the computation performance of the network. This reduction of the spatial data also leads to a reduction of the number of parameters of the network and as a result to the chance of overfitting it. In CNNs we most commonly use max pooling, seen in figure 2.2, which selects the maximum value from each convolution. There are however more pooling functions, like average pooling[4]. Pooling layers render the network invariant to small changes to the initial input, which is a very important property in image classification.

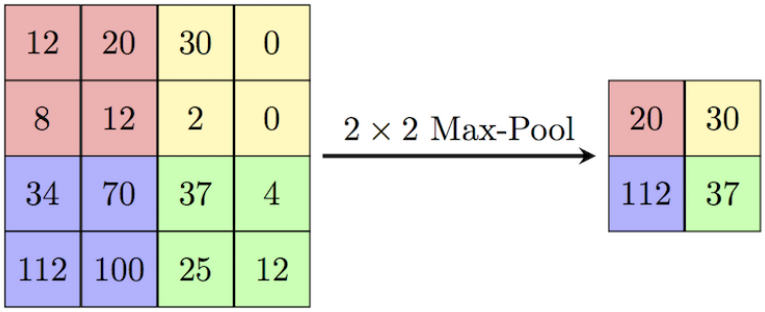

FIGURE 2.2: Max Pooling: https://stackexchange.com

2.1.3 Fully Connected Layer

Fully Connected Layers are layers of nodes each of whom, much like neurons, is connected to all nodes of the previous layer. They are used for the supervised classification of the input and they have a huge number of parameters whose values are determined through the training of the network. The computation of the values of each node, constitutes the lions share of the workload of a classification. For each node we need to know the weights w_j with which we will multiply the values of the nodes of the previous layers x_j and the bias b that will be added to their sum. Finally we need to choose an activation function f.

$$y_i = f(b_i + \sum_{j=1}^{J} w_j x_j) \tag{2.2}$$

2.1.4 Final Classification

The final classification step uses the multi-class generalisation of the Logistic Regression known as Softmax Regression to exploit the probabilistic characteristics of the normalized exponential (softmax) function. Let y_i be the output of i-th node of the last Fully Connected layer. We can then calculate the probability distribution and make our classification by picking the maximum.

$$\sigma(y)_i = \frac{e^{y_i}}{\sum_{j=1}^{J} e^{y_i}} \tag{2.3}$$

2.2 High Performance Computing prototypes

Exascale computing refers to computing systems capable of at least one exa FLOPS, or a billion billion calculations per second and it would be considered a significant achievement in computer engineering, for it is estimated to be the order of processing power of the human brain at neural level [42] (functional might be lower). Most modern countries have been investing heavily in the development of such a system for this past decade, after the first petascale computer first came into operation, in 2008. The computing capabilities of such powerful computers make them ideal for applications in Artificial Inteligence like Neural Networks.

The E.U. has been running High Performance Computing (HPC) projects like CRESTA (Collaborative Research into Exascale Systemware, Tools and Applications), the DEEP project (Dynamical ExaScale Entry Platform), and the Mont-Blanc project since 2011 and has at the moment an entire HPC ecosystem of various development and application projects. Most notably the supercomputer architecture is being designed in the EuroExa and DEEP-EST projects. In the U.S., the Department of energy has announced that their first exascale supercomputer, *Aurora*, will be operational at Argonne National Laboratory by 2021. China has announced that its first exascale computer, Tianhe-3 will enter service by 2020 and Taiwan, Japan and India are also developing their own exascale computers.

In the following chapters we will present the architecture outlines of some of the aforementioned prototypes and argue why the one used during this thesis is the most suitable to the computational needs of a convolutional neural network.

2.2.1 EuroExa and DEEP-EST

Both EuroExa and DEEP-EST are part of the European HPC ecosystem[11], consisting of 19 research projects which cover the entire area of topics related to the development of an exascale computer and its applications.

EuroExa

Originally the informal name for a group of Horizon 2020 research projects, ExaNeSt[13], EcoScale[7] and ExaNoDe[14], EuroEXA[10] aims to develop and deploy an ARM Cortex technology processing system with FPGA acceleration at petaflop level by 2020.

The proposed system[21] is a heterogeneous machine designed to have 19 racks, each consuming 110 KW, for a total consumption of 2 MW of power. Racks consist of 4 network groups, each containing 8 liquid cooled blades connected in a 3D Torus topology. Each blade comprises 16 slots, where computational nodes are placed, divided in 4 quadrants which are connected in a Full Crossbar topology through a switch. The nodes in every quadrant are all connected with each other and their heterogeneous architecture combines a CPU with an FPGA.

In contrast to traditional HPC node architecture, in which the processor is placed between the accelerator and the HPC interconnect, EuroExa, as we can see in figure 2.3, connects the storage to both the CPU and the FPGA accelerator, bringing the data closer to the computation. To make this possible, a large-scale, multi-tiered interconnect and Global Shared Address Space were needed to allow access to the Storage were needed. These goals were achieved through the the development of the UNIMEM (storage) and ExaNet (interconnect) technologies, as part of the EuroEXA project.

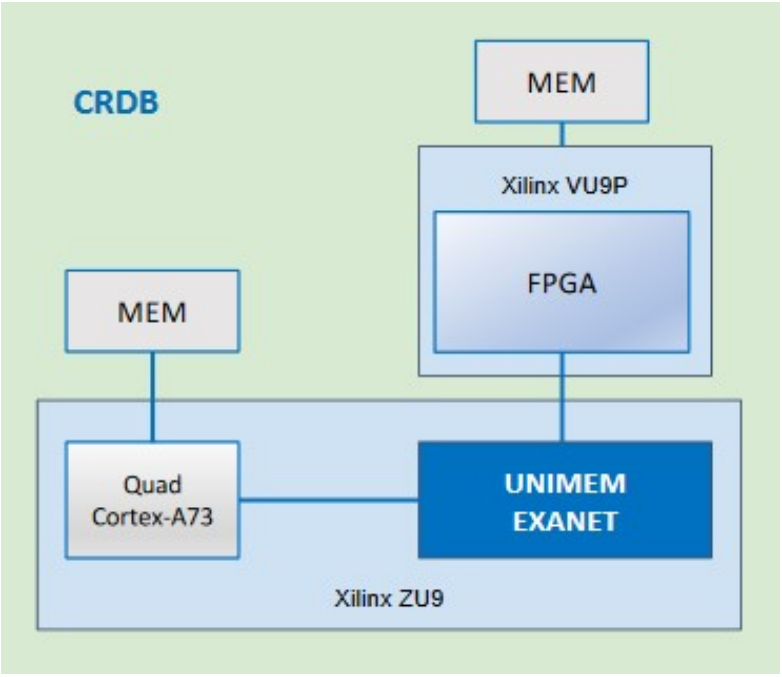

FIGURE 2.3: Co-design Recommended Daughter Board (CRDB) architecture outline: https://events.prace-ri.eu/

The node board implementing this architecture, named Co-design Recommended Daughter Board (Board), incorporates 2 FPGAs, the Xilinx ZU9 and the new Xilinx VU9, on a single board, each with its own memory chip. The ZU9 ARM-v8 Quad Cortex-A73 CPU will be used as the programmer for the VU9 FPGA, with both of them connected to the ZU9 reprogrammable logic implementing the ExaNet and UNIMEM protocols. This way both FPGAs have access to data stored on the shared memory space not only on this node but on all the nodes on the HPC system.

The development of such a system required breakthroughs in several key areas, like communication, hardware design, etc. The solution of these problems is the common goal of the projects which comprise the EuroExa program. More specifically:

- **ExaNeSt:** is responsible for the physical de-ployment characteristics to support the required compute density, along with the storage and inter-connect services.
- **ExaNoDe:** focuses on the delivery of low-energy compute elements for HPC.
- **EcoScale:** focuses on integrating FPGAs and providing them as accelerators in HPC systems.

To test the architecture of the proposed CRDB system, a prototype board was developed, namely the Quad FPGA Daughter Board (QFDB)[3], which was the board used for the purposes of this thesis. The board will be tested in a HPC system[28][47] comprised of racks, with 3 chassis on each rack and 9 mezzanines on each chassis. The mezzanine connects 4 QFDBs in a 3D-Torus topology using 2 16 Gbps HSSL links on each connection.

DEEP-EST

Within the DEEP (Dynamical Exascale Entry Platform), DEEP-ER and DEEP-EST research projects various prototypes have been or are being built, with DEEP-EST[6] being the most advanced of the three. The system implements a modular architecture, with nodes specialized for the computations they will be ordered to perform.

- **Cluster Module (CM):** An Intel Xeon based HPC Cluster with high single-thread performance and a universal Infiniband interconnect, used to process low to medium scalability code parts.

- **Data Analytics Module (DAM):** An Intel Xeon based Cluster with large, fast, non-volatile, byte-addressable memory, one Intel Stratix 10 FPGA and one NVIDIA Tesla V100 card. DAM nodes will be interconnected by 40 Gb/s Ethernet and a 100 Gb/s EXTOLL fabric. DAM nodes are designed for use by data-intensive and machine learning codes.
- **Storage Module (SM):** A storage and service module will provide high-performance disk storage and login nodes. This module will use 40 Gb/s Ethernet, and run the BeeGFS parallel file system.
- **Extreme Scale Booster (ESB):** NVIDIA Tesal-based nodes with a small Intel Xeon CPU and EXTOLL 3D Torus interconnect. ESB nodes are designed to run highly scalable code parts with energy efficiency. This is achieved by running applications from the local V100 memory, and using GPU Direct technology to bypass CPU for network communication.

The prototype is planned to have 50 CM and 75 ESB nodes, both liquid cooled, as well as 16 DAM air cooled nodes. In addition, Network Attached Memory nodes will provide persistent shared memory resources at EXTOLL network speeds, and an experimental Global Communication Engine will drive collective MPI communication on the ESB. Finally, a network federation infrastructure ties all the modules together, supporting MPI and IP communication.

2.2.2 Aurora and Frontier

The US Department of Energy (DoE) and the National Nuclear Security Administration (NNSA) as part of a combined effort named Exascale Computing Project (ECP), have announced the delivery of 2 exascale supercomputers named Aurora and Frontier(OLCF-5) by 2021.

Aurora

Aurora is a planned state-of-the-art exascale supercomputer[23] designed by Intel/Cray for the US DoE Argonne Leadership Computing Facility (ALCF). It is a stepping stone to the fully exascale system, the Frontier supercomputer, as its peak system performance has been announced to be in the range of 180 to 450 petaflops. The final system is planned to have 50000 computing nodes[2], using the Intel Xeon scalable processor. It has not been made public as of yet, whether the node architecture will include some kind of accelerator.

Frontier

Frontier (OLCF-5) is Summit's successor, a planned exascale supercomputer designed by AMD and Cray, that will be operated by the DoE Oak Ridge National Laboratory. The final system aims to achieve a peak performance of 1.5 exaflops using a heterogenous node architecture, Each node will be made up from an AMD EPYC server CPU and 4 purpose built AMD Radeon Instinct server GPU accelerators. Multiple Slingshot NICs will provide 100 Gbps of network bandwidth[17].

2.2.3 Sunway, TianHe-3 and Sugon

China aims to develop an 100,000-node-plus exascale supercomputer based on what appears to be primarily pre-exascale componentry[16]. There are currently three prototypes[48][51][40] for the node architecture, namely Sunway, Sugon and TianHe-3, whose system architectures are presented below.

Sugon

The Sugon prototype, whose proposed architecture can be seen in figure 2.4, is a liquid immersion cooled heterogenous machine comprised of nodes, each outfitted with two Hygon x86 CPU and two DCUs. The prototype numbers 512 nodes in total, connected in a 6D torus network topology. The CPU is a licensed clone of AMD's first-generation EPYC processor, while the DCU is an accelerator built by Hygon.

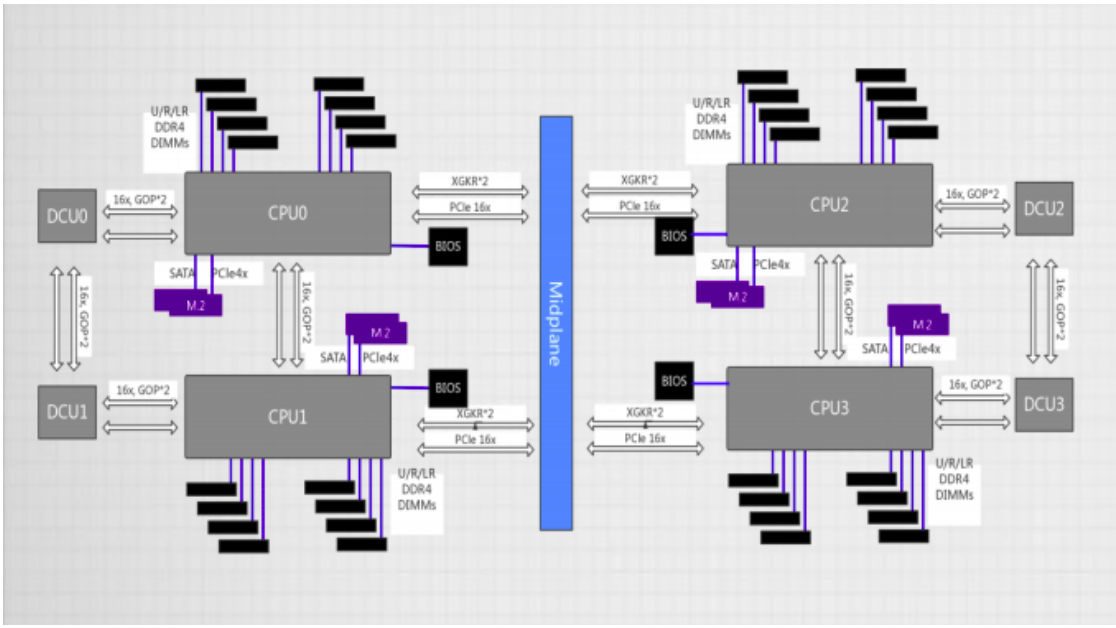

FIGURE 2.4: Sugon Exascale Supercomputer Prototype Node Architecture: https://www.r-ccs.riken.jp/en/

TianHe-3

The TianHe-3 512-node prototype, developed by the National University of Defence Technology, is another heterogenous architecture, with each node consisting of a CPU, presumably Intel Xeon, and 3 Matrix-2000+ DSPs, the successor to the Matrix-2000 accelerator used in the 100-petaflop Tianhe-2 supercomputer. Each Matrix-2000+ accelerator is made up of 128 64-bit cores, operating at 2 Ghz, to deliver a peak performance of 2 teraflops on 130 Watts.

Sunway

The Sunway prototype, developed by the National Research Center of Parallel Computer Engineering and Technology (NRCPC), is a CPU-only machine and as such it is the only non-accelerated architecture currently vying for exascale honors in China. Each of its nodes is equipped with two ShenWei 26010 (SW26010) processors, each with 260 cores arranged in 4 groups of 64 processors each, delivering 3 teraflops of 64-bit floating point performance. Presumably there is a more powerful chip in the works, somewhere in the order of 10 teraflops, to be used in the future exascale system.

2.3 Choice of a Computational Platform

During the past decade there have been major advances in the tools that we use to implement neural networks. Software libraries like Tensorflow, ASIC circuits like the Tensorflow Processing Unit (TPU) or GPU frameworks like the NVIDIA Deep Learning SDK have been designed specifically for optimized neural network performance. And while ASIC circuits are most definitely the best choice, since we can design the hardware in such a way that best suits the NN specific operations, the development of a system with exascale capabilities, designed only for convolutional Neural Network applications is simply not realistic. Which brings us to the next best thing: FPGAs.

As recent work has shown[27][30][43], FPGAs offer significantly better performance than both CPUs and GPUs. Thus, when we deal with exascale computing applications, it makes sense to choose a heterogenous node architecture that combines a CPU with an FPGA. This makes the EuroExa node architecture ideal for this purpose, with the CPU handling the data flow and the FPGA running the application specific accelerator.

2.4 Thesis Approach

This thesis aims to model the performance achieved by a multi-node system conducting the necessary computations for any given CNN design with known throughput and latency. Moreover we want to test the accuracy of this model, by comparing it with actual measurements taken by implementing a QFDB-targeted CNN hardware design and running the resulting system on multiple QFDBs. This could provide meaningful data for the performance gain we can have by running such applications in a future exascale system.

Chapter 3

Computational Model

In this chapter we will present a description of the network architectures of the CNNs, designed and implemented as applications of the technology, that was developed in the Euroexa research program. We will present the motivation behind the design of these CNNs and the problems they can solve. We will then describe their structure, their computational workloads and possible design bottlenecks these may create and finally argue why FPGAs provide such an efficient solution to the challenges they create. Finally, we present an overview of the hardware implementation of these networks.

3.1 1-Dimensional Convolutional Neural Network

The first of the two CNNs that we will discuss, is a simple Convolutional Neural Network that takes an 1-Dimensional input and uses it for classification. Motivation for the design of this network was the problem of spectroscopic redshift estimation in Astronomy [46]. Due to the expansion of the Universe, galaxies recede from each other on average, causing the emitted electromagnetic waves to shift from the blue part of the spectrum to the red part, due to the Doppler effect. Several sources of noise render the estimation process of the initial spectrum far from trivial. Thus, a CNN was designed by Dr. Gregory Tsagatakis and his team, to translate the 1-Dimensional spectroscopic redshift data collected by the Euclid ESA satellite[9] into a reliable classification of galaxies.

To transform Redshift Estimation, which is a regression problem, into a classification problem, which we can solve using CNNs, we focus our study in the redshift range of detectable galaxies. Given the specific characteristics of Euclid, the chosen redshift range is split into evenly sized slots equal to Euclid's required resolution. Each slot represents a class and the problem we now need to solve is the correct classification of our data.

3.1.1 Network Architecture

The CNN we are studying has the structure of a typical CNN, as it was described in the previous paragraph. The network is comprised of 3 convolutional layers, with the use of a non linear activation function interposed between them. After that we have one fully connected Layer and a final Softmax classification stage. The basic structure of such a network can be seen in figure 3.1.

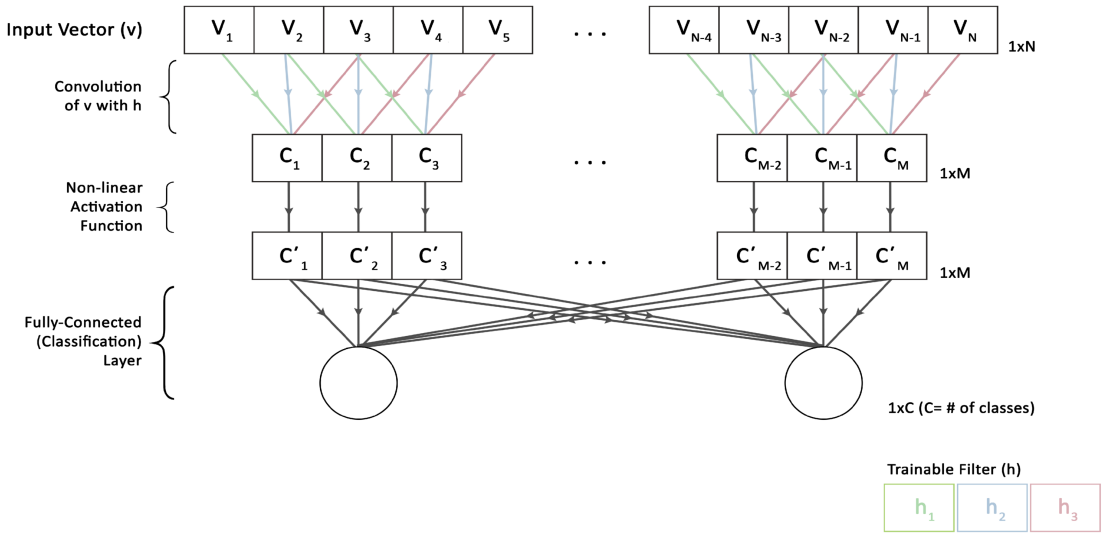

FIGURE 3.1: A simple 1-Dimensional CNN: The input vector v is convolved with a trainable filter h (with a stride equal to 1), resulting in an output vector of size M = N - 2. Subsequently, a non-linear transfer function (typically ReLU) is applied element-wise on the output vector preserving its original size. Finally, a fully-connected, supervised layer is used for the task of classification. The number of the output neurons(C) is equal to the number of the distinct classes of the formulated problem (800 classes in our case).
Picture from G. Pitsis thesis

1st Convolution Layer

The first convolutional layer takes a vector input of 1800 32-bit floating point values, which are the aforementioned redshift data. To calculate the output, we also need the 16x8 kernel matrix and the 16 bias values, summing up to a total of 144 32-bit floating point values. The output of the layer is a 1800x16 matrix, or 28688 floating point values, after removing the margin.

To calculate the output, the kernel matrix is broken into 16 vectors of length 8, each of which is multiplied with a vector of the same size, containing 8 values which were read from the input. This calculation involves 8 Multiply and Accumulate (MAC) operations, one for each kernel-input value pair. This calculation is

repeated 16 times for each of the 16 vectors, into which the kernel matrix has been broken into. The kernel matrix is then shifted by one value, meaning that that the layer has stride value equal to 1 and then process repeats, until the output has been calculated. The stride value is maintained throughout the network.

2nd Convolution Layer

The second convolutional layer takes as input the output of the previous layer. To calculate the output we need the 16×16×8 kernel matrix and the 16 bias values, which sum to a total of 2064 32-bit floating point values. The output of the convolution is again a 1800×16 matrix, which after removing the margin leaves us with 28576 floating point values.

To calculate the output, the kernel matrix is broken into 16 matrices of size 16x8. The values of the input are read in sets of 16 values, which are subsequently multiplied with each of the 8 16-value lines of the kernel matrix fragments. The process is repeated for all 16 matrices, into which the kernel matrix was broken into and the products are then summed, providing the value of the output for the 16 values that were read from the input. To calculate all the values of the output, the process is repeated for each set of 16 values of the input.

3rd Convolution Layer

The third convolutional layer is the same as the second, using 2064 floating point weight values to calculate the output, with the input being again the output of the previous layer. The output is calculated in exactly the same way as before, producing a 28464 floating value matrix.

Fully Connected Layer

The input of the fully connected layer is again the output of the previous layer. The layer itself consists of 800 neurons, with each of them corresponding to one of the output classes. Since every neuron takes as input all 28464 values of the output of the previous layer we need a total of 22771200 weight values to calculate the probability of each class.

Just like in the convolutional layers, only MAC operations are required to calculate the final result. Namely, each input value is multiplied with its corresponding weight and after adding all the products, we apply an activation function on the sum, providing us with the value of the corresponding neuron. After repeating the process for every neuron on the layer, the probability of each class is calculated using the Softmax algorithm.

3.1.2 Hardware Design

The main advantage of using using FPGAs for this type of workload is that we can design a pipeline specifically fitted to the needs of the problem, taking advantage of any parallelism that is available in the algorithm. A detailed description of the hardware design of this CNN can be found in the work of Georgios Pitsis[37].

The calculations required in both the convolutional and the fully connected layers are in essence MAC operations, which can be heavily parallelized. However, we are limited by the memory bandwidth, which only allows us to bring a certain amount of data to the network. Thus, to achieve the best possible level of pipelining, the data are not passed to the next layer after the computation of the output of each layer has been completed. Instead, a FIFO queue is used to stream the output data, only after they suffice for the completion of a single computation of the next layer. This results in the maximum possible utilization of the pipeline, throughout the entire computation.

Convolutional Layers

Like we described previously, the redshift data are read from the input of the 1st convolutional layer in sets of 8 values. The multiplication with the kernel matrix is calculated in parallel for all 8 of them, using the datapath in figure 3.2.

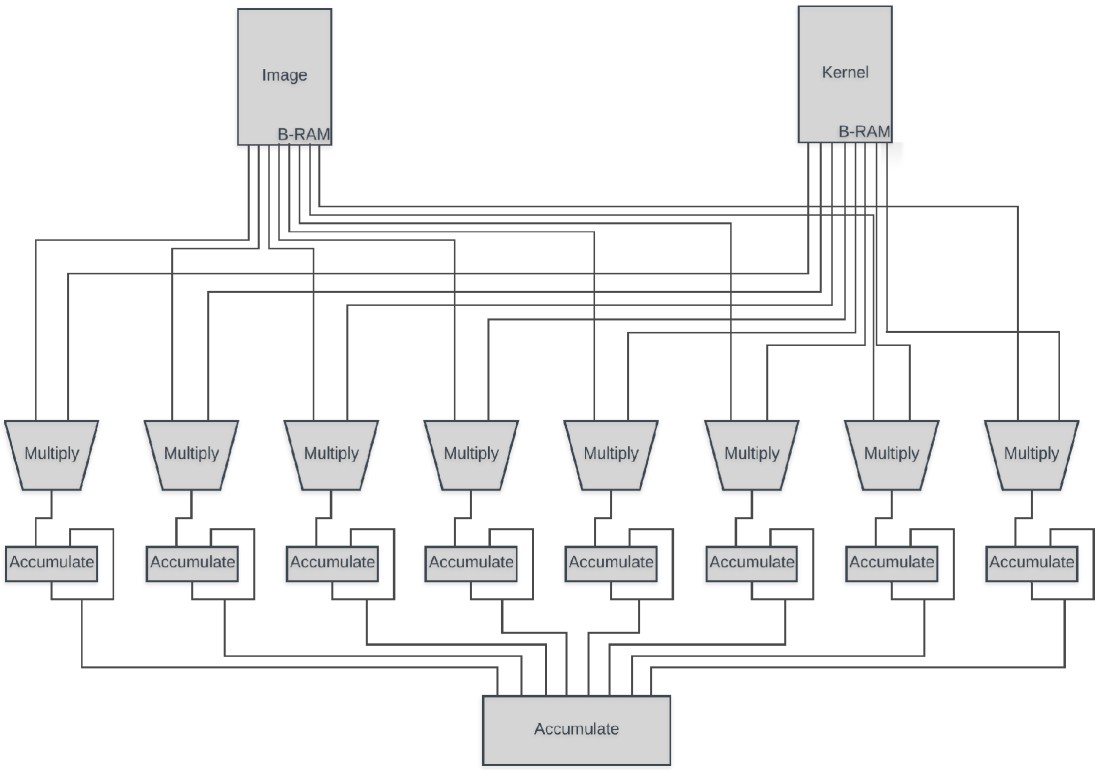

FIGURE 3.2: MAC operation datapat: Picture from G. Pitsis thesis

The input data and the kernel values are streamed in the convolutional layers using a DMA. Every time an output value is calculated in the first convolutional layer, it is forwarded to a FIFO queue. Every time the FIFO queue accumulates 16 output results, it streams them to the next layer. The reason why we need 16 results is that the 2nd convolutional layer requires a matrix input of 16 values to calculate an output value. The process uses the same datapath as before, except now 16 MAC operations are calculated in parallel instead of 8. Once a sufficient number of output values has been calculated, the data are again forwarded to a FIFO queue, just like in the 1st layer. The same process takes place in the 3rd layer. In figure 3.3 we can see an outline of the hardware design of the network including the bus sizes and number of values transferred at each stage, as we described in the previous paragraphs. It should be noted that the figure depicts only the datapath of the input values inside network, omitting the connection to the memory and the datapath of the weight values.

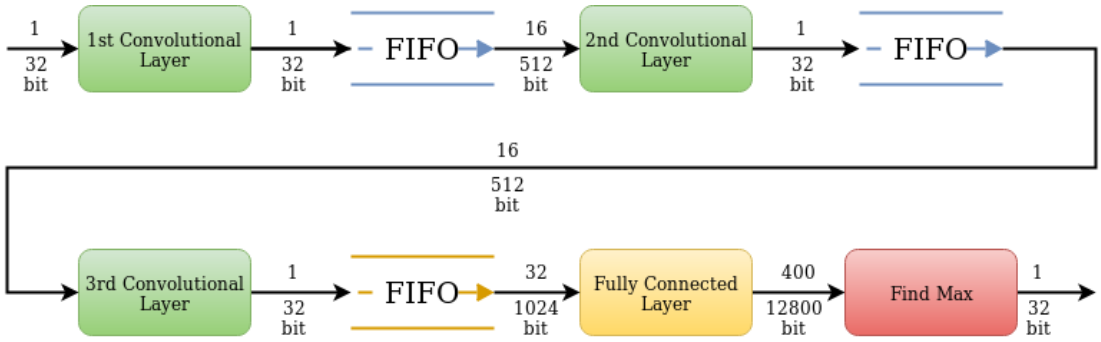

FIGURE 3.3: CNN Datapath

Fully Connected Layer

The fully connected layers present the greatest obstacle in the design of the accelerator. Since every neuron is connected with all neurons of the previous layer, we need to calculate all output values of the 3rd convolutional layer before we can produce the outputs of the fully connected layer. Thus, the rate at which we can transfer data from the 3rd convolutional layer, together with the memory bandwidth, which limits the rate at which we can stream weights from the memory, constitute the bottlenecks of the design. The memory bus size is 128 bits, streaming 32 weight values on a single transfer. Breaking the fully connected layer in 2 identical modules, each calculating the outputs of 400 classes, allows the use of 2 DMAs streaming in total 256 bits or 64 weight values per single transfer. We also need 64 input values, streamed from the previous layer through 2 512 bit channels. As soon as we have the weight values and the corresponding inputs however, we

can start the calculation of the output. This is done with MAC operations, allowing once again the use of the datapath that was previously described. Once the process has been completed for all the classes, we write the output values back in the DDR.

Memory Utilization

From the previous paragraphs, it becomes obvious, that the rate at which we can produce a result of the network, depends upon the rate at which weight values are streamed in the design. Thus it is essential to know the amount of data, that are used in the design and the rate at which the memory can transfer these data. In the following tables, 3.1 and 3.2, we will present the number of weights and data that need to be streamed, their size and the memory percentage they require to be stored.

Table 3.1: Weights Memory Footprint

Layer	#Weights	Footprint	Memory(%)
conv1	144	1.1KB	$6.33 * 10^{-4}$
conv2	2064	16.1KB	$9.2 * 10^{-3}$
conv3	2064	16.1KB	$9.2 * 10^{-3}$
dense	22771200	173.7MB	99.98

Table 3.2: Data Stages Memory Footprint

Stage	#Data	Footprint	Memory(%)
image	1800	14KB	2
conv1	28688	224KB	32
conv2	28576	223.2KB	32
conv3	28464	222.4KB	32
dense	800	6.25KB	0.9

It is obvious that there is not enough space on the BRAMs for all of these data, which necessitates their transfer from the DDR. For this purpose the design makes use of 3 DMAs. As mentioned before, the first DMA streams the kernel matrix values to the convolutional layers of the network and subsequently streams the input values in the first convolutional layer. The other 2 DMAs stream weight values to the fully connected layer. Each DMA has an 1 Gb/s bandwidth.

To reduce the size of the data even further, compression techniques like Pruning[22], Pair Compression and Clustering, can be applied to the data. A detailed

description of how these techniques were used in our CNN, can be found in the thesis work of George Pitsis[30][38].

3.2 Alexnet

The second CNN that we will discuss is called Alexnet and for the purposes of this thesis it is used for object detection in images. Motivation for the design of Alexnet was the ImageNet LSVRC contest of 2010, which posed the problem of classifying 1.2 high resolution images in 1000 different classes.

In the original paper by Alex Krizhevsky[32], the architect of the network, Alexnet has a linear architecture, with one layer after the other. In a more recent paper[31] however, Krizhevsky has presented a different approach with 2 branches of the network calculating the results of separate parts of the input image, introducing a model level parallelism in the design. Each of the two branches has exactly the same structure as the linear network of the original paper. After this improvement the linear architecture was named Caffenet and the paralellized version kept the name Alexnet. The aim of this parallelization is the optimisation of the performance of the network in GPU-based computation platforms. The architecture of the parallelized Alexnet is shown in figure 3.4.

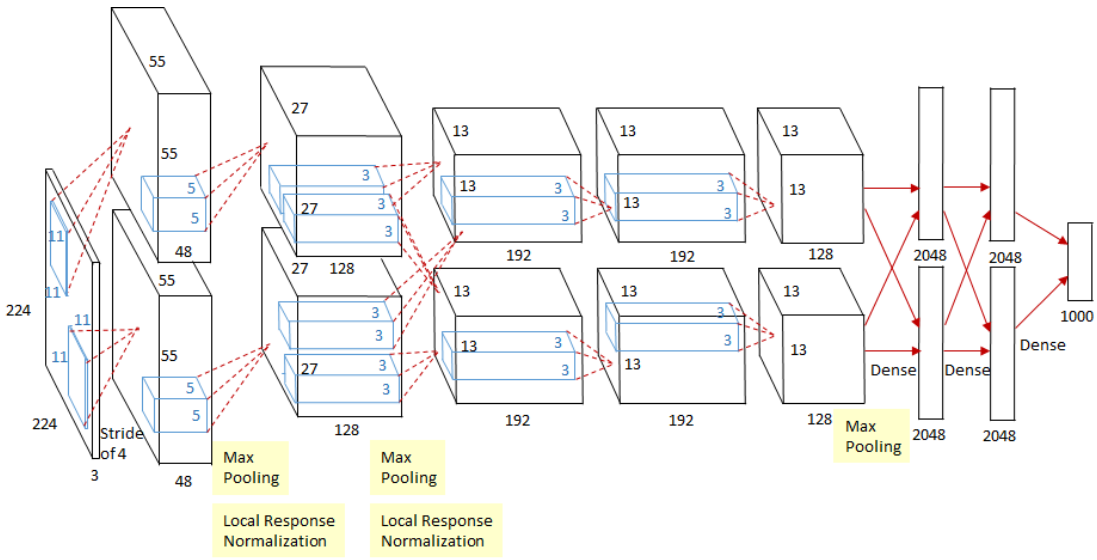

FIGURE 3.4: Architecture of the Alexnet CNN: https://medium.com

At the time this thesis was written, the hardware implementation of the network, which is part of the pending thesis work of Tzanis Fotakis, has not yet been completed. Thus we will only describe the structure of the model created for the

aforementioned work and of course the architecture of the network, as it is described in the original paper. The purpose of this section is to present an even more complex problem than the 1-dimensional CNN presented previously and the computational work that it requires. This information will be used to test the model of the multi-node system in the next chapters.

3.2.1 Network Architecture

Alexnet has the structure of a typical CNN, with convolutional layers followed by fully connected layers, but its architecture is significantly deeper than that of the 1-dimensional CNN we previously described. Its original linear architecture[20] is comprised of 5 convolutional layers with pooling layers interposed between them, followed by 3 fully connected layers. Furthermore normalization is applied between specific layers of the network and neuron dropout is used instead of regularization, to deal with overfitting.

Convolutional Layers

Like in the 1-dimensional CNN we described previously, all convolutional layers essentially apply a single kind of operation upon their inputs. This operation, namely convolution, is the multiplication of a given kernel matrix with a window of input values. The window iterates on the input with a step equal to the stride value, until convolution has been applied to all input data. It might contain only data from the input or it can be zero padded, depending on the information we want to extract. The result of the operation depends on the values of the kernel matrix, its dimensions and the activation function we apply. These characteristics change from layer to layer, as we will see in the following paragraph. The use of a pooling or normalization layer can also have an effect on the output of the network.

- **1st Convolution Layer**

  The 1st layer takes an input with size 227×227×3 and a kernel matrix matrix with size 11×11×96 (width×height×depth) and has a stride value of 4. A Rectifier (ReLU) is used as an activation function and no zero padding is applied, giving us an output of size 55×55×96. The total number of parameters required for the computation of the result of the layer is (11×11×3+1)×96=43944.

  Between the 1st and 2nd convolutional layer we interpose a Max Pooling layer with a filter size of 3×3 and a stride value of 2. The output of the

pooling layer has a size of 27×27×96 and is normalized before being passed to the 2nd convolutional layer.

- **2nd Convolution Layer**

  The 2nd convolutional layer takes as input the output of the previous layer, zero-padded with 2 zeros on every dimension and a kernel matrix with size 5×5×256 and has a stride value of 1. The activation function is again a ReLU. The output has a size of 27×27×256 and the total number of parameters required for its computation is (5×5×96+1)×256=614656.

  We interpose another Max Pooling layer between the 2nd and the 3rd convolutional layer, with a the same filter size and stride like before. The output of the pooling layer has a size of 13×13×256 values and is again normalized before being passed to the 3rd convolutional layer.

- **3rd Convolution Layer**

  The input of the 3rd convolutional layer is, like before, the output of the previous layer and a kernel matrix with size 3×3×384 with a stride value of 1. The ReLU function is used for neuron activation and we apply zero padding with one zero value on each dimension of the input, before calculating the result. The output has a size of 13×13×384 and the total number of parameters required for its computation is (3×3×256+1)×384=885120.

- **4th Convolution Layer**

  The input of the 4th convolutional layer is the output of the previous layer and a kernel matrix with size 3×3×384 with a stride value of 1. It is identical to the 3rd layer so the ReLU is again used as activation function and a zero padding of 1 is applied to the input. The output has a size of 13×13×384 and the total number of parameters for the layer is (3×3×384+1)×384=1327488.

- **5th Convolution Layer**

  The final convolutional layer takes as input the output of the previous layer and a kernel matrix with size 13×13×256 and has a stride value of 1. The activation function is a ReLU and we also use a zero padding of one zero value to the input. The output has a size of 13×13×256 and the total number of parameters required for its computation is (3×3×384+1)×256=884992.

  Before passing the result of the 5th convolutional layer to the fully connected layers we pass it through another max pooling layer, with a filter size of 3×3 and a stride step of 2. The final output has a size of 6×6×256 values, with each value stored in a neuron connected to all neurons of the next layer.

Fully Connected Layers

As we have shown before, a fully connected layer is simply a set of neurons, each connected to all neurons of the previous layer. Every input value is then multiplied with a weight value and the products of all the multiplications are added together along with a bias value. From a computational perspective the layer essentially applies a MAC operation on the input. The sum is then passed to an activation function, which produces the value of each neuron of the layer.

During training we can apply neuron dropout as a means to avoid overfitting[36], a common problem with such large networks. As recent work[19] has shown however, dropout can also be used during testing, to increase confidence in the results of the network, as we will show in the next chapters. In his original paper presenting Alexnet, Krizhevsky uses neuron dropout only during training and only in 2 places in the network. More specifically dropout is applied between the 5th convolutional layer and the 1st fully connected layer and between the 1st and 2nd fully connected layers.

- **1st Fully Connected Layer**

  The 1st fully connected layer has 4096 neurons, each connected with all neurons of the 5th convolutional layer. This means that to calculate the output of the layer we need $6\times6\times256\times4096=37748736$ weight values and 4096 bias values. The neurons are activated by means of the ReLU activation function.

- **2nd Fully Connected Layer**

  The 2nd fully connected layer, just like the 1st, is made up from 4096 neurons, each connected with all neurons of the previous layer, requiring a total of $4096\times4096=16777216$ weight values and 4096 bias values to produce its result. Once more, ReLU is used as the activation function for the layers neurons.

- **3rd Fully Connected Layer**

  The 3rd fully connected layer is the final layer of the network. It is comprised of 1000 neurons, one for every class of the LSVRC-2010 contest problem. Each of them is connected with all neurons of the previous layer, requiring a total of $1000\times4096=4096000$ weight values and 1000 bias values to produce the result of the probabilities for each class.

Chapter 4

Modeling and Robustness analysis

In this chapter we will try to establish the necessity of the work carried out during this thesis, by using software models to create a rough approximation of an FPGA network and the workload it can handle. Two models were built for this purpose, one in MATLAB, in the form of a graph, presenting us with a very rough approximation of the computing capabilities if our multinode system and one in Python presenting a more elaborate, discrete event simulation.

In the second part of the chapter we will use runtime neuron dropout on the CNN hardware accelerator designed by G. Pitsis and G. Kalomoiris to improve its robustness. We use the outputs of multiple runs of the CNN for the same input with different neuron dropouts as a random variable to improve our confidence in the networks prediction.

4.1 Multi-node System Dataflow Analysis

The main objective of this thesis is to run a CNN accelerator on multiple computational nodes of a supercomputer. This of course requires a constant flow of data to these nodes. The rate at which the data are streamed, together with the rate the nodes can process the data, determines the number of nodes required for the computation. It is obvious that, if the data flow is small enough, even a single node would be sufficient for the corresponding application, rendering the use of a supercomputer unnecessary. On the other hand, if the system does not have enough computational power, additional nodes would have to be added, to avoid an increase in latency. Our goal is, to create a roughly approximate model of our multi-node system and dismiss both these possibilities for the system we will use for this thesis.

The computational power of our system can be expressed in terms of latency and throughput, of both the nodes and the connections between them. In our case, the latency and throughput of each node are measured by running the accelerators

as a standalone design on a single node while corresponding values for the inter-node connections have been measured in FORTH as part of the development of the computational platform, the QFDB.

The basic layout of the computational platform used for the purposes of this thesis can be seen in figure 4.1. A Manager PC is connected to a number of computational platforms, streaming to them data which either comes from an external input of is stored on the computer itself. Each platform has four computational nodes, each equipped with its own CNN accelerator. Although all nodes all connected to each other, we are only interested in their connection with the first node, which is the node to which data is streamed from the Manager PC. The data to the remaining nodes is streamed from the first node through a set of links with known throughput and latency.

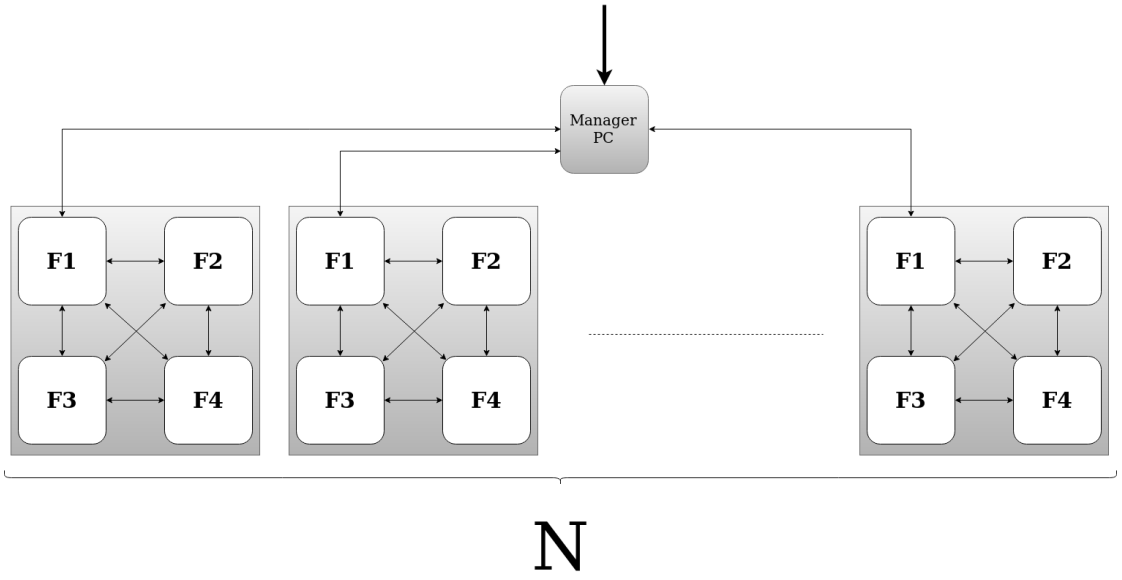

FIGURE 4.1: Multi-node System Layout

Adopting this model requires certain assumptions, which are not strictly true in the actual computing platform we use, in order to achieve. The first assumption is that the data on the Manager PC can be accessed simultaneously by all QFDBs and that there is a continuous flow of information. The second assumption is that the latency of the software application running on the PS side of each MPSoC is negligible, which is most definitely is not. Finally we assume that the latency and throughput measurements for each link include the part of the transmission protocol that takes place inside the FPGA, particularly the R-DMA. This potential bottleneck cannot be represented in this model without the addition of a considerable amount of complexity to it.

4.1.1 Graph Model

From the aforementioned assumptions it becomes obvious that the layout of the system we presented in the previous paragraph is not an exact representation of the real system. This is even more obvious in our model, which is simply a graph, whose edges and nodes represent groups of design modules and have their own latency and throughput values.

Each node represents an FPGA on a QFDB board and its latency and throughput values are the values measured for the accelerator with which that FPGA is programmed. The node models the part of the design where data is streamed from the DDR to the accelerator and back, once the processing is finished. Each edge represents a link between two FPGAs or a QFDB and the Manager PC and its latency and throughput are values measured by the CARV laboratory team. The nodes model the part of the design where data is transferred from one physical memory to another, including all the packaging and depackaging actions needed. It should be noted that we do not include the connections between pairs of QFDBs in the model since we do not make use of that infrastructure in this work. The model is created using algorithms 1 and 2, presented here in pseudocode.

Algorithm 1 Find number of QFDBs

```
1:  procedure FIND NUMBER OF QFDBS(Link, qfdbLat, qfdbThr, linkLat, linkThr)
2:      Cluster = digraph();
3:      Cluster = addnode(Cluster, 2);
4:      Cluster.Nodes.Throughput = [Link; Link];
5:      Cluster.Nodes.Latency = [0; 0];
6:      Cluster = addedge(Cluster, 1, 2);
7:      Cluster.Edges.Throughtput = Link;
8:      Cluster.Edges.Latency = 0;
9:      Thr = 0;
10:     fpga = 0;
11:     numOfQFDBs = 0;
12:     while (Link/numOfQFDBs)>Thr do
13:         Cluster = addQFDB(Cluster, qfdbLat, qfdbThr, linkLat, linkThr);
14:         thr1 = Cluster.Nodes(fpga+1,1));
15:         thr2 = min(Cluster.Edges(fpga + 2, 2), Cluster.Nodes(fpga + 2, 1));
16:         thr3 = min(Cluster.Edges(fpga + 3, 2), Cluster.Nodes(fpga + 3, 1));
17:         thr4 = min(Cluster.Edges(fpga + 4, 2), Cluster.Nodes(fpga + 4, 1));
18:         Thr = Thr + min(Cluster.Edges(fpga+1,2),thr1 + thr2 + thr3 + thr4);
19:         fpga = fpga + 4;
20:         numOfQFDBs = numOfQFDBs +1;
21:     return numOfQFDBs;
```

Algorithm 2 Add QFDB to the graph model

```
1: procedure ADD QFDB(g, qfdbLat, qfdbThr, linkLat, linkThr)
2:     n = numnodes(g);
3:     e = numedges(g);
4:     g = addnode(g, 4);
5:     g.Nodes(n + 1 : n + 4, 2) = qfdbLat;
6:     g.Nodes(n + 1 : n + 4, 1) = qfdbThr;
7:     g = addedge(g, [2(n + 1)(n + 1)(n + 1)], [(n + 1)(n + 2)(n + 3)(n + 4)]);
8:     g.Edges((e + 1) : (e + 4), 2) = linkThr;
9:     g.Edges((e + 1) : (e + 4), 3) = linkLat;
10:    return g;
```

At this point an important question is posed. Why would we use such a simple model to represent a much more complex system? The answer is that we are only using this model to prove that the amount data streamed to the system through the input link of the Manager PC requires more than one QFDB for its processing. If for a given input link bandwidth running the model returns a number of required QFDBs equal to or bigger that one, this means that the application we implemented on our hardware is fitting for use in a larger scale system.

4.2 Robustness Improvement through Dropout

In this section, we will present a paper which applies a dropout technique at prediction time, in order to increase confidence in the results of a neural network designed in software. We will subsequently use the knowledge gathered from this paper and use it to replicate the results not in our hardware accelerator. Finally we will present our results using a suitable metric.

4.2.1 Monte Carlo Dropout

The technique we will describe is a form of regularization that prevents overfitting during training[44], called Dropout. During dropout a certain number of randomly selected neurons are simply turned off, which in practise translates to setting their weight and bias values to zero. The model is then left to train without these nodes, which eliminates the risk of overfitting. However as more recent work has shown, a form of dropout can also be applied during prediction time, as we will show. Figure 4.2 demonstrates the basic functional principle of the Dropout technique.

In a paper titled *Dropout as a Bayesian Approximation*[19], Yarin Gal and Zoubin Ghahramani show that the use of dropout in neural networks during prediction, can be interpreted as a Bayesian approximation of a Gaussian process, which

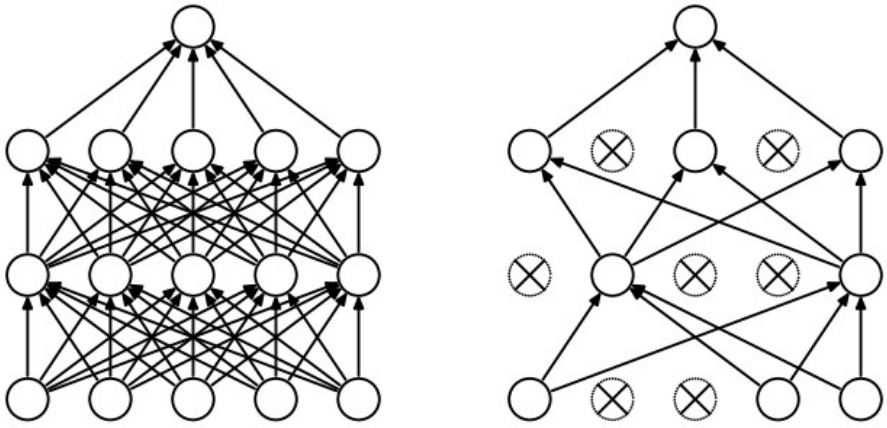

FIGURE 4.2: Weight Dropout
Dropout: A Simple Way to Prevent Neural Networks
from Overfitting

has a well known probabilistic model. In practise all we do is to use dropout on the network over multiple prediction runs, maintaining the same neuron dropout percentage, but randomly selecting a different set of neurons to be turned off every time. We will refer to this set as the dropout mask.

This of course has an effect on our predictions, causing the correct detection rate for each particular run to fall. However, after several runs, we can use the prediction results as random variables. If a prediction remains the same, no matter which neurons are dropped and that prediction coincides with the ground truth result, we can use a suitable metric to calculate how much our confidence in that prediction is increased. In essence we simply cause a disturbance in the system and use its stability to increase our confidence in the networks results.

4.2.2 Application on hardware accelerator

Using the hardware accelerator as a Gaussian process requires a number of different dropout masks to be applied on the weights. However, before when can create those masks we need to make sure that we have a set of data that we can process after the classification is done.

The input data we were given as part of the deliverables of G. Pitsis' thesis lacked labeling, since the results were compared with a software implementation of the CNN and not the ground truth. Without labels we could not use these data for post processing, as we will show in the next paragraphs. Thus a new labeled dataset was requested from Dr. Tsagatakis. It should be noted here that running this new dataset in the hardware accelerator involves the risk of a slightly lower resulting performance, since it was trained for a different dataset. This difference

however, should be fairly small and with performance being of little significance in this chapter, we can consider it as negligible.

The new dataset came in the form of a MATLAB array, whose elements had to subsequently be stored in a binary file as 32 bit integers, in order to be in a format compatible with the once used in the hardware accelerator. With this task completed we then had to create the dropout masks that we would use. This was achieved with algorithm 3, seen here in pseudocode. The algorithm makes use of the *datasample* function of MATLAB, which randomly samples a number of the elements of the array it is given.

Algorithm 3 Dropout Masks

```
1: procedure CREATE DROPOUT MASKS(filename)
2:     FCbias = fread('filename.txt','int32');
3:     FCidx = 1:len(FCbias);
4:     i1 = datasample(FCidx, len(FCbias)/2);
5:     i2 = datasample(FCidx, len(FCbias)/2);
6:     i3 = datasample(FCidx, len(FCbias)/2);
7:     i4 = datasample(FCidx, len(FCbias)/2);
8:     FCbias1 = FCbias;
9:     FCbias2 = FCbias;
10:    FCbias3 = FCbias;
11:    FCbias4 = FCbias;
12:    for (int i=0; i<len(FCbias)/2; i++)
13:        FCbias1(i1(i)) = 0;
14:        FCbias2(i2(i)) = 0;
15:        FCbias3(i3(i)) = 0;
16:        FCbias4(i4(i)) = 0;
17:    fwrite('FCbias1.txt', FCbias1,'int32');
18:    fwrite('FCbias2.txt', FCbias2,'int32');
19:    fwrite('FCbias3.txt', FCbias3,'int32');
20:    fwrite('FCbias4.txt', FCbias4,'int32');
```

As it is obvious, only 4 dropout masks were created for use on the hardware accelerator, which one could argue is a small number, when the intent is to use the NN as a Gaussian stochastic process. However, we only need a small number of masks because the confidence in the results of the classifier is already high and we only need a small improvement to prove that the technique works in practice. The second and more important reason is that four is also the number of FPGAs in a QFDB board. The obvious implication of course is, that in the scaled-out system we can reduce the throughput of each QFDB, by processing one input at a time on four accelerators with different dropout masks, in exchange with a higher confidence in the results of the network.

Once we have created the dropout masks, we need to run the same input through our CNN accelerator once for each of the dropout masks. To do this, all the different masks were stored in the SD card and the only change that had to be made for each run, was the filename of the weights on the SDK code. The results of each run were also stored on the SD card of the ZCU102. For post processing, the output data was first read in MATLAB and then exported as a .m file to be compatible with the Jupyter Notebook in Python.

4.2.3 Performance Measurement

After running the network for different neuron dropouts, we need to process the obtained data to calculate the confidence in the classification results. We can then compare the confidence before and after the application of the Monte Carlo dropout and demonstrate that we can trade computational power for better confidence in our results.

Confidence Metrics

At this point it is meaningful to differentiate between the accuracy of a prediction and the confidence in that prediction. Accuracy defines the skill of the learning algorithm in predicting accurately by measuring the percentage of correct predictions. Confidence on the other hand defines the probability that the model has made the correct predicting given an input.

There are several approaches when it comes to calculating confidence in the predictions of a classifier, some of which we will present here. One way to determine confidence is to measure uncertainty using the Predictive Posterior Mean[45], as can be seen in equations 4.1 and 4.2.

$$Predictive\,Posterior\,Mean : p = \frac{1}{T}\sum_{i=0}^{T} f_{nn}^{d_i}(x) \tag{4.1}$$

$$Uncertainty : c = \frac{1}{T}\sum_{i=0}^{T} \left(f_{nn}^{d_i}(x) - p\right)^2 \tag{4.2}$$

where $f_{nn}^{d_i}(x)$ is the prediction of the network with dropout mask d_i and T is the number of predictions with different dropout masks. We can also calculate the 95% Confidence Interval directly, assuming a 2σ range around our predictions for each class[24]. We can calculate σ, which is of course the standard deviation, using equation 4.3.

$$\sigma = \sqrt{\frac{1}{T}\sum_{i=0}^{T}\left(f_{nn}^{d_i}(x) - p\right)^2} \tag{4.3}$$

Another approach to the measurement of confidence is the use of the Confusion Matrix[15]. To calculate the confusion matrix we need to calculate the true and false positive as well as the true and false negative results of the network. We do this by comparing the networks' predictions with the ground truth. We can then use the values of the matrix to derive some valuable information, using the equations 4.4 through 4.7.

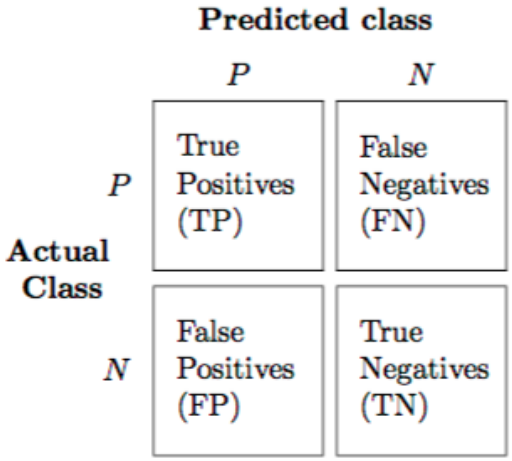

FIGURE 4.3: Confusion Matrix: https://quora.com

$$Recall: TPR = \frac{TP}{TP + FN} \tag{4.4}$$

$$Sensitivity: ACC = \frac{TP + TN}{TP + FP + TN + FN} \tag{4.5}$$

$$Positive\,Predictive\,Value\,(Precision): PPV = \frac{TP}{TP + FP} \tag{4.6}$$

$$Negative\,Predictive\,Value: NPV = \frac{TN}{TN + FN} \tag{4.7}$$

To extract information about the confidence in our true positive predictions we use the Positive Predictive Value, while for true negative predictions we can use the Negative Predictive Value. It should be noted here that the confusion matrix in figure 4.3 represents a binary classifier. The CNN we study has 800 classes, which means that the confusion matrix is an 800×800 matrix and the aforementioned values have to be calculated for each of them.

Chapter 5

Design and Implementation

5.1 Tools

The tools used for the hardware implementation of this thesis, are all part of the Xilinx Vivado Design Suite, a software suite produced by Xilinx for synthesis and analysis of HDL designs. The 2017.1 Edition was used for the editing of the standalone designs of the CNN and the 2017.2 Edition was used to incorporate the CNN IPs into the ExaNeSt design. The use of newer versions was avoided, to ensure compatibility with the existing projects. Vivado has 3 main tools: Vivado HLS, Vivado IP Integrator and Vivado SDK, each fulfilling the needs of different stages of the development process.

5.1.1 Vivado HLS

Vivado High-Level Synthesis (HLS) is a synthesis tool, whose compiler enables C, C++ and SystemC programs to be directly targeted into Xilinx devices, without the need to manually create RTL. The tool allows developers to take advantage of the capabilities of high level programming languages and then handles the task of synthesizing the programs into Intellectual Property (IP) blocks, by automatically generating the appropriate VHDL or Verilog code, according to user-determined design constraints. The generated blocks can then be integrated into a real hardware system, using the remaining tools on the Xilinx toolchain. The high level of abstraction it offers, is widely reviewed to increase developer productivity.

To create an IP block we first need to create an HLS project, which contains the libraries included in the project, the source files and the testbench files. The developer can then design the hardware using C/C++ or SystemC language. Special commands called pragmas can be used, to modify or optimize the resulting VHDL or Verilog code that will later be generated. Once we are done, we can run a simulation on the code and identify possible errors using the integrated debugger. With the code working as intended, we can now proceed to synthesis, which

generates the corresponding VHDL or Verilog code, based on a clock period and target device provided by the user. The ensuing RTL design is summarized in a report, which includes the following information:

- **Timing:** The report presents information about the estimated clock frequency the design will run on and an estimation for the latency of each module and of the entire design.
- **Utilization:** The report presents the percentage of LUTs, FFs, DSPs and BRAMs of the device that are estimated to be utilized by the design.

Vivado HLS also allows the user, to edit the generated HDL code or to design the IP directly, using only HDL code, thus requiring no synthesis step. Finally, before we package the IP, we can use a testbench, to run a co-simulation of the hardware design with a software program and verify that it produces the correct output.

Optimizations in Vivado HLS

As we mentioned in the previous paragraph, HLS uses special commands called pragmas, to modify and optimize the way the design works. We can for example reduce latency, improve throughput, reduce the utilized resources of the device, etc. Here we present a few pragmas crucial for the design used for the purposes of this thesis.

- **pipeline:** The pipeline pragma reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations. It effectively pipelines the function, allowing it to process new inputs every N clock cycles, with N being the II.
- **array_partition**: The array partition pragma partitions an array into smaller arrays or individual elements. This results in the use of multiple smaller memories instead of a large one in the RTL.
- **unroll:** The unroll pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel.
- **stream:** By default, array variables are implemented as RAM. The stream pragma replaces the RAMs with FIFOs, creating a more efficient communication mechanism for producing and consuming the data stored in the array in a sequential manner.

- **dataflow:** The dataflow pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation and the overall throughput of the design.

5.1.2 Vivado IPI

The Vivado IP Integrator is the main design tool in the Xilinx Vivado Design Suite. It allows hardware developers to create complex system designs by instantiating and interconnecting IP cores, either through its design canvas GUI, in the case of this thesis, or programmatically, using a Tool Command Language (Tcl) programming interface. IP cores can either be designed and packaged by Vivado HLS, as mentioned in the previous section, or they can be part of the Vivado IP catalog. The tool also has the capability of converting MathWorks Simulink designs into IPs, using Xilinx's System Generator.

The design process begins by creating a new block diagram and adding to it the IPs that will be used in the design. A base address, clock signal and reset signal are then assigned to each IP, either automatically, using the block and connection automation, or manually. Then the IP I/O signals are manually connected and the settings of each IP can be configured, according to the needs of the design. Once we are done, a design validation is run to check for errors in the design.

With the design validated, we can now proceed to synthesize and implement the design. Synthesis is the process of turning an RTL design into a logic gate schematic, while implementation is the task of optimally placing and routing the synthesized design on a predetermined FPGA. The user has the option of choosing between different strategies for the completion of these tasks, based on the needs of the problem at hand.

After the completion of each of these two stages, the tool provides a report, containing important information and possible error and warning messages. More specifically, each of the 2 reports contains information about the following aspects of the design:

- **Timing:** The report indicates whether the clock signal reaches the entire design in time and if not, it reports back the delays this causes.
- **Utilization:** The report presents information about the percentage of LUTs, FFs, DSPs and BRAMs that are utilized by the design.
- **Power:** The report presents information concerning the projected power consumption of the design and the thermal effects it will have on the device.

With the implementation process completed without errors and the implementation characteristics presented on the report meeting requirements, we are now ready to generate the bitstream file which we can use to program our device using the Hardware Manager.

Finally, it is important to mention the hardware debugging features included in the Hardware Manager tool of the IPI. If a design flaw has been detected, we can use an Integrated Logic Analyzer (ILA), to record signal values in the design. We can then view and analyze these signal values through the Hardware Manager to determine the location of the design flaw.

5.1.3 Vivado SDK

Vivado uses the Xilinx Software Development Kit (XSDK) as an Integrated Design Enviroment (IDE) for creating embedded applications on any of the following Xilinx microprocessors: Zynq UltraScale+ MPSoC, Zynq-7000 SoCs, and the MicroBlaze soft-core microprocessor. It is based on Eclipse 4.5.0 and CDT 8.8.0 and thus includes a C/C++ code editor and compiler, as well as application build tools and a debugging environment. It interfaces directly to the Vivado embedded hardware design environment, providing the user with features like flash memory management or scripting tools like the Xilinx Software Command Tool (XSCT). SDK can be launched either from the Xilinx IP Integrator or separately as a standalone program, but for the purposes of this thesis we use the former option.

Once the design implementation has been completed and the bitstream file generated and exported, we open the SDK, which automatically imports the necessary output product files and device drivers of the hardware design. We then create an application project, causing the tool to automatically import the Board Support Packages (BSP) and libraries corresponding to the board used in the design. We can then proceed to develop the application software programs, whose role is to initialize and activate IPs, DMAs, memories and other instances of the design, as well as to coordinate data transactions between them.

With the application software ready, we can configure the connection with the target device and run the application on the Processing System (PS), to ascertain the correct function of the hardware design, with which we have programmed the Programmable Logic (PL). The PL can either be programmed by the Hardware Manager tool of the IP Integrator or directly through the SDK, by setting the path to the bitstream file in the Run Configurations window. To confirm the proper function of the hardware design, we need to be able to communicate data both from and to the device. We can configure the I/O port to use either the JTAG

cable, which is already connected for the programming of the device, or the UART cable for that purpose.

5.2 FPGA platforms

The architectures implemented for the purposes of this thesis were targeted to two platforms, namely the Xilinx ZCU102 and the QFDB node prototype. In this section we will present a few basic facts about the platforms and a set of important figures to get a quantitative view of their capabilities.

5.2.1 Xilinx Zynq UltraScale+ MPSoC ZCU102

The ZCU102, seen in figure 5.1, is a general purpose evaluation board for rapid-prototyping based on the Zynq UltraScale+ XCZU9EG-2FFVB1156E MPSoC. High speed DDR4 SODIMM and component memory interfaces, FMC expansion ports, multi-gigabit per second serial transceivers, a variety of peripheral interfaces, and FPGA logic for user customized designs provides a flexible prototyping platform. The board uses a Quad-core Cortex-A53 MPCore as an APU, a Dual-core Arm Cortex-R5 as a RPU and a Arm Mali-400 MP2 as a GPU.

FIGURE 5.1: The ZCU102 evaluation kit: ZCU102 User Guide

The following tables present the main features of the board. More specifically, table 5.1 offers a summary of the resources available on the FPGA and table 5.2 presents the sizes of the various memories available on the board.

Table 5.1: ZCU102 block features

Logic Cells	Flip-FLops	LUTs	DSP Slices	B-RAMs
599550	548160	274080	2520	912

Table 5.2: ZCU102 memory sizes

PL DDR (GB)	PS DDR (MB)	Distributed RAM (Mb)	Block RAM (Mb)
4	512	8.8	32.1

5.2.2 Quad FPGA Daughter Board

The QFDB, seen in figure 5.2, is the HPC Testbed Prototype built for ExaNeSt [3] project, which is now a part of the EuroExa research project. It is equipped with 4 Zynq UltraScale+ XCZU9EG-FFVC900-2-e MPSoCs, a part of the same family as the one used on the ZCU102 board, making the conveyance of a hardware design from one platform to the other a relatively easy task. Every FPGA node is connected to a 32MB QSPI memory and to a 16GB DDR4 SODIMM, able to transfer data at a rate of 160 Gbps, with the total available memory of 64 Gb on the board. The board is also equipped with a 256 GB SSD/NMVe (2 TB devices are available today), connected to one of the MPSoCs. The FPGA nodes are connected in an all-to-all intra-node topology with 2 High Speed Serial Links (HSSL) using GTH transceivers as well as 24 Low-voltage differential signaling (LVDS) pairs. Finally one of the MPSoCs is connected to the outside world with 10 HSSLs at 10.3125 Gbps, also using GTH transceivers.

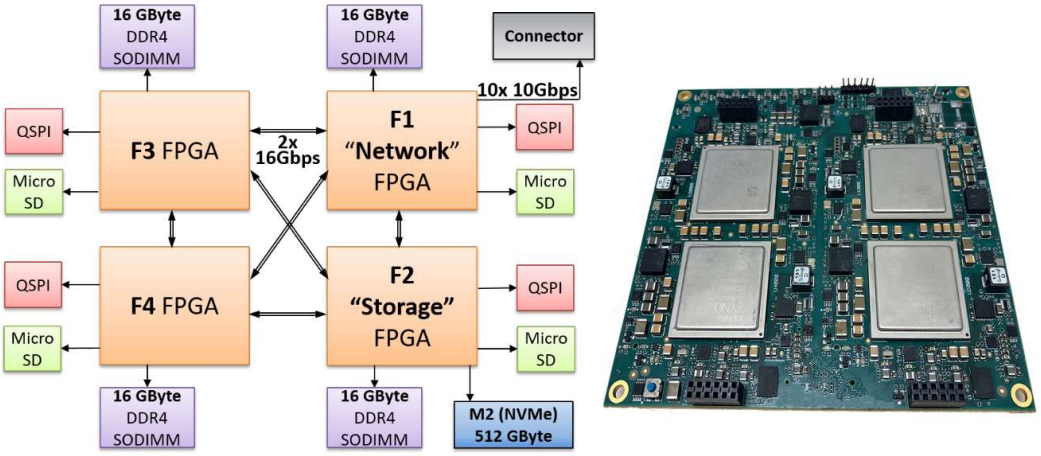

Figure 5.2: QFDB architecture (final version) and actual board

In table 5.3 we present the high speed signals on the QFDB and the nominal values of their bit rates. The QFDB has a flexible infrastructure, in order to support versatile research activities. Thus, the values included in the table represent the peak performance of the system and were measured under very specific conditions. As a result, there is a marked difference between these values and the values in our results.

Table 5.3: High Speed Signals in the QFDB

Type	**Count**	**Maximum Speed**
DDR4	532	1200 MHz / 2400 Mb/s
LVDS	288	800 MHz / 1600 Mb/s
HSSL	88	16.375 Gb/s
PCIe	16	5 Gb/s

For the purposes of this thesis we used 2 QFDB boards, one of which was one of a few prototypes not equipped with the PCIe port connecting to the SSD/NMVe storage. Since the board itself was at the time in the development stage, neither QFDB had access to micro SD memory cards and the actual measured performance of the HSSL links, was significantly inferior to the one in the final version of the board, as can be seen in table 5.4.

Table 5.4: Measured HSSL performance

External HSSL Throughput (Gbps)	Internal HSSL Throughput (Gbps)	Internal HSSL RTT Latency (ms)
3	3	0.048

The utilization of a prototype consisting of several HPC multi-node boards is usually tedious, requiring the development of software tools to solve the associated problems. To accommodate the needs of the platform, a Linux distribution called Carvoonix was created, based on Gentoo Linux. A trimmed Linux kernel, retrieved from the local QSPI flash memory, is used to boot the full Linux system. The trimmed kernel, nicknamed *loby*, mounts a root NFS storage, common to all boards of the prototype. Once the booting process is complete, files can be transferred from the manager PC to the board, through the EtherNet links mentioned previously.

5.3 Batching - Case study of a persistent bug

The first task of this thesis was to review the work of G. Pitsis and G. Kalomoiris[27], focusing on a variation of their published architecture, that increased of the original batch size from 1 to 2 input samples. This architecture was not published due to an issue causing the network to produce inaccurate results. Just like in the original architecture, the network outputs are the index of the class holding the maximum neuron value and the maximum value itself. Since however the batch size is now 2, the network outputs those values for both input samples. The issue at hand was, that all 4 output values produced for each input pair, were incorrect. In this section we will give an outline of the "Batch 2" architecture and describe how this issue was resolved.

5.3.1 Batch 2 hardware design

The "Batch 2" design is based on the original published design with batch size 1. Figure 5.3 shows the Vivado block diagram of the design. For the sake of perspicuity, we have omitted the clock and reset signals and have merged together sets of signals that carry a single value.

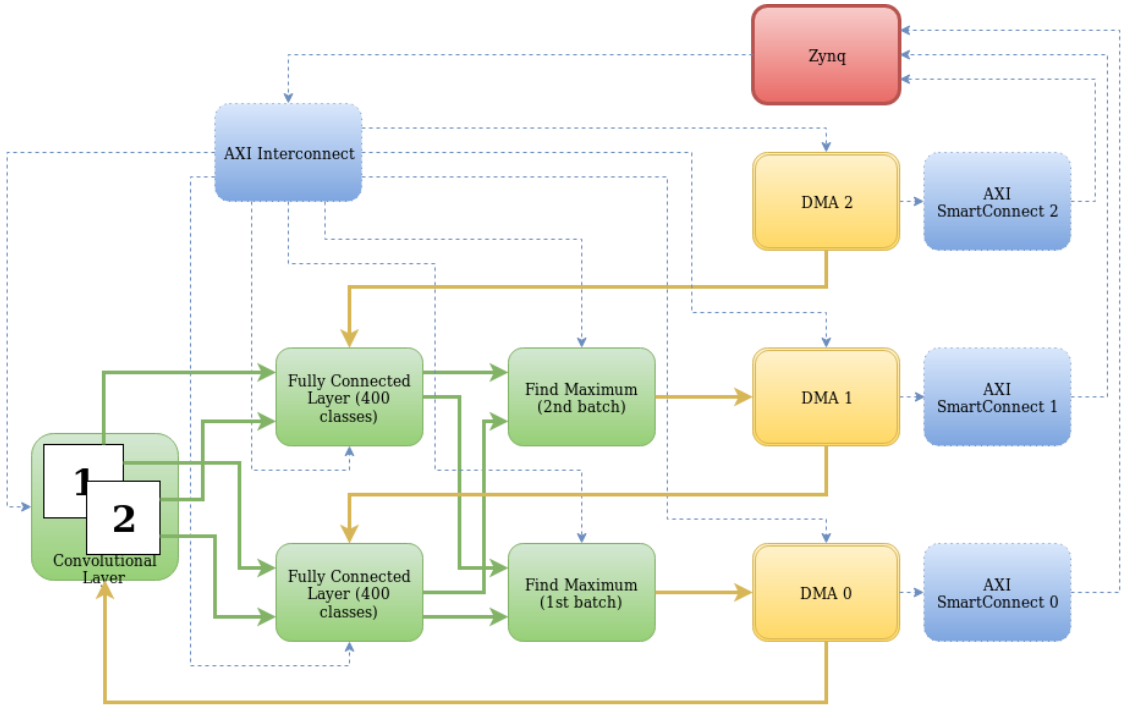

FIGURE 5.3: The Batch 2 Vivado design

Communication between the Zynq Ultrascale+ processor and the IP cores, including the streaming of the data, is done with the use of the AXI4 protocol.

We use 3 DMA IP cores, one for the kernel and input values streamed to the IP containing the convolutional layers and two for the weight values streamed to the 2 IP cores, each containing 400 nodes of the fully connected layer. In order to achieve the maximum possible performance the design also makes use of 3 SmartConnect IPs to link the DMAs with the slave port of the PS. Finally, just like in the original "Batch 1" design, we use an AXI Interconnect IP core in the master port of the PS.

CNN Architecture

The architecture of the CNN is also based on the original "Batch 1" architecture, the only difference being the fact that we have two pipelines, processing twice the amount of input values concurrently. The basic principle is, that we trade away board resources to double the throughput. To that end, the IP cores containing the convolutional and fully connected layers have been redesigned, while the IP that finds the maximum neuron value is the same as in the original architecture.

To allow the concurrent processing of the two input values, we used the surplus of resources on the target board, by executing the MAC operations for each input value in the same loop iteration. Another parallelization strategy was to use loop unrolling. This was done in both the convolutional and the fully connected layers, which meant that the size of the input and output ports of each IP, had to be double the size of that in the "Batch 1" architecture. We can see this in the datapath of the design in figure 5.4. The solid lines follow the input data stream and the dashed lines follow the weight and kernel values stream.

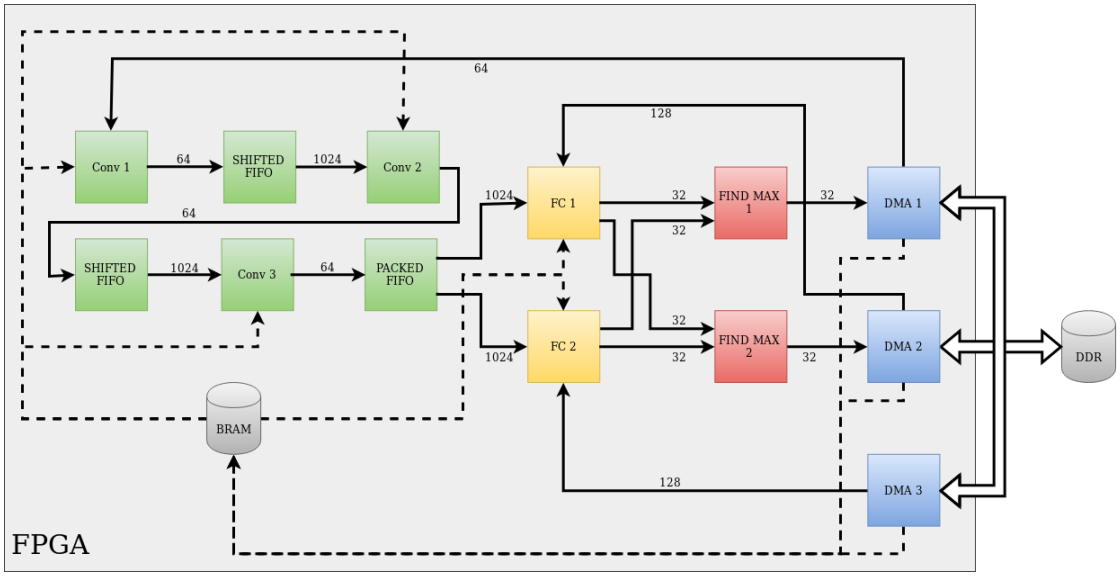

FIGURE 5.4: The "Batch 2" architecture datapath

5.3.2 Vivado SDK code review

Since the SDK code is simply an application running in the PS, changing it requires no intervention in the hardware design, thus the developer simply has to rerun the amended code and check the correctness of the results. This means, that an error in the SDK code requires the least possible amount of work and time to be corrected.

The SDK code of this design can be divided into 3 main sections, each completing a certain task. The first section initializes the DMAs that will be used by the design and allocates the necessary memory space in the DDR for the data to be read in. The second section reads the data from the SD card into the DDR of the ZCU102 and rearranges them, so that they may be transferred in the B-RAMs in the sequence required by the design. Finally, the third section sets up the hardware accelerator and streams the data into it, using a dedicated function of the BSP, which carries out simple transfers from the DDR to the B-RAMs through DMAs.

Memory allocation

Since there are no error messages and data are passed through to our design, it is safe to assume that the DMAs have been initialized properly. After checking that the space allocated for the data is consistent with the requirements of the design, we can also assume that we have enough space in the DDR for all the data to be read. This is confirmed by the outputs of the program, as we described them in previous paragraphs, since we receive the exact same number of results, as the number of inputs we streamed into the design. Thus, the possibility of a code error existing in this section of the code was quickly discarded, simply by taking a closer look at the output data.

Data streaming

The next step was to look for an error in the third section. This was decided so, because reading the data from the SD card into the DDR is a simple task and thus the existence of an error in that section of the code was considered improbable at the time. Reviewing the code of the third section however, proved to be a time-consuming task, since it required an understanding of the inner workings and correct use of the simple transfer function of the BSP. Searching for an error in this section proved fruitless and the only option left was to check for an error in the reading of the data.

Data reading

As mentioned above, the second section of the code, is where the data are read from the SD card into the DDR and rearranged, in a way that is compatible with the function of the hardware design. The reading of the data from the SD is done through a parsing function, whose implementation is a trivial task, especially for a high level programming language like C. After that, a series of for loops rearrange the data in a new variable, by a simple index translation. At first glance there seemed to be no errors, but after reviewing the code several times, an indexing error was found in the for loop rearranging the sequence in which the input values were passed in the design. More specifically the existing code read input values from the DDR using algorithm 4.

Algorithm 4 Original SDK code

```
1: for (int i=0; i<NUM_TESTS/2; i++)
2:     for (int i=0; i<DIM3; i++)
3:         images_for_batch2[2*j+i*DIM3]=tinput[i*DIM3+j];
4:         images_for_batch2[2*j+i*DIM3+1]=tinput[(i+1)*DIM3+j];
```

In this case, NUM_TESTS is the number of inputs, while DIM3 is dimension of the kernel matrix adjacent to the input. The problem was located in the sequence the inputs were stored read from the DDR, as can be seen in the amended code of algorithm 5.

Algorithm 5 Amended SDK code

```
1: for (int i=0; i<NUM_TESTS; i=i+2)
2:     for (int i=0; i<DIM3; i++)
3:         images_for_batch2[2*j+2*i*DIM3]=tinput[2*i*DIM3+j];
4:         images_for_batch2[2*j+2*i*DIM3+1]=tinput[(2*i+1)*DIM3+j];
```

Correcting the indexing error yielded immediate results. After running the revised code, both index values of the class, that holds the maximum neuron value came up correct and so did the maximum value of the 1st batch. The maximum value of the second batch however, continued to deviate from the expected numbers. Even so, there was a marked improvement, with the new maximum values for the second batch lying significantly closer to the expected ones and certainly in the same order of magnitude, unlike before. This pointed to the existence of a second error, which after several additional reviews of the SDK code, was assumed to be in other stages of the design.

5.3.3 Vivado IP Integrator design review

Detecting and rectifying any errors in the IP integration stage of the design is a much more straightforward task, since there is no code to review, but confirming that the error has been eliminated is much more time consuming. Every time a change is made in the design, the synthesis and implementation steps need to be reset and run again. The completion of this task lasts a couple of hours, even in modern computers, like the one hosting the TUC Zeus server, on which this work was done.

Interrupt connection

The first approach was to confirm, that all IP settings were correct and all connections between the various IPs of the design used the correct IP interface ports. A deviation to the original "Batch 1" design was found, since the interrupt signals of the HLS generated IPs were not connected to the PS. Even though this should not affect the design, since we eventually do not make use of these interrupts, a second interrupt port was enabled on the Zynq processor IP and the interrupts were connected there, just to preclude all possibilities. As expected, this approach did not yield any results and since no further errors were found, we proceeded with a second approach: the use of an Integrated Logic Analyzer (ILA) IP.

Integrated Logic Analyzer

Introduction of an ILA into the design is done by marking the signals we want to analyze as debug signals and using the Autoconnect feature of the IPI. Once the design has completed the implementation step, we open the Hardware Manager and program the device. Vivado IPI can then record the marked signals for a fixed period of time, beginning from a user-defined trigger event.

This simple task proved to be quite a challenge though. After inserting the ILA IP and implementing the design, we set the arrival of the first bit in the IP of the fully connected layer, as the trigger event for the recording of the signals. However, even though the Hardware Manager seemed to reach the trigger event, no signals were recorded, indicating a problem is the use of the ILA. After several failed attempts, we resorted to the creation of a new project and the design of the entire system from scratch. This approach aimed to eliminate the possibility that the project was stuck in a faulty state after being transferred through several computers and uploaded on the server. This brought no results and consequently it was decided, that investing more time in this approach, before first exploring the possibility of an error in the HLS code, would be counterproductive.

Reviewing the HLS code is a meticulous, time-consuming task, thus, it would be meaningful to narrow down our search domain. As mentioned in the description of the design, the two modules we use to find the maximum values of each batch, are two instances of the same IP and their only inputs are the values of the neurons of the fully connected layer. Since the values of the neurons of the first batch produce correct results when passed through this module, we can deduce that the problem lies in the values of the neurons of the second batch. In actuality however, we had to review the code of this IP too, to rule out all possible causes of the problem, since the initial review of the code of the other IPs was not fruitful.

5.3.4 Vivado HLS code review

A modification of the design in the HLS step, is the most time-consuming task, since we have to rerun synthesis, testing and packaging and then we have to integrate the new IP in the design and repeat all implementation steps. As we mentioned, we logically excluded the possibility of the error lying in the IP which located the maximum value. Following the same logic, we can narrow down our search in the code of the fully connected layer.

Fully Connected Layer

Since the modules of the fully connected layers are two instances of the same IP and the results of the first batch are produced correctly, there is either a problem with the inputs related to the second batch or the code describing the hardware design of the first batch, is different to the code of the second batch. Since the weights used to calculate the neuron values for both batches are the same, we only needed to examine the second case, in order the ensure that the IP of the fully connected was working as intended. This was done by a comparison of the code used for each of the two batches, with special focus given to the index values in the multiplication of the weights with the input values. After several reviews of the code, no errors were found and we moved on to the review of the code implementing the convolutional layers.

Convolutional Layers

Following the same logic as before, the cause of the problem has to be a difference in the manner in which the inputs of the second batch are read or manipulated. We know this because the first batch produces correct results. This proved to be exactly the case, with the error finally being located in the second convolutional layer of the second batch.

As we described in the 3rd chapter of this thesis, the second convolutional layer reads 16 input values at a time, which are then multiplied with the values of the kernel matrix and added together. The addition however does not take place in a single step. In order to have a smaller fan-in we we add the products together in groups of four and then these 4 partial sums are added together to produce the final sum. In our case, the error was that the first partial sum of the second batch actually added together the first 4 products of the first batch. This was caused by a wrong index in these 4 values, resulting in the hardware design seen in figure 5.5.

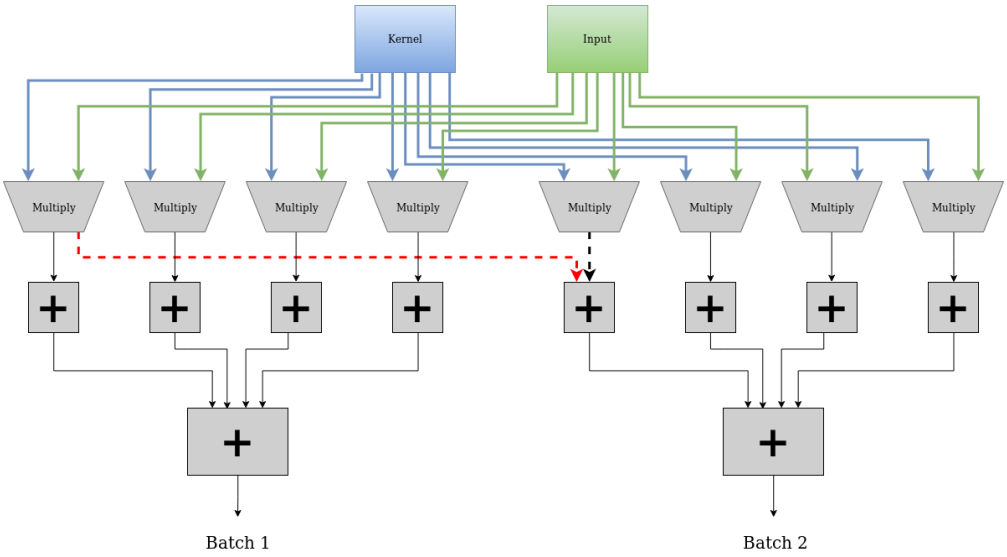

FIGURE 5.5: HLS error: The red dashed line is sum of the first 4 products of the 1st batch connected where the black dashed line of the first 4 products of the 2nd batch should be

As we have demonstrated in previous paragraphs, the deviation from the actual maximum value of the 2nd batch was small and within the same order of magnitude. This is consistent with the results this type error would produce. The values of the partial sum came from the multiplication of weights with actual inputs, except in this case it was the wrong set of inputs. Thus it is safe to assume that the design is now fixed. With the error corrected, we needed to synthesize and package the IP, upgrade the design and repeat all implementation steps, to ascertain whether our system is now working as intended.

5.3.5 DMA Configuration Bug

Before we could confirm the design was working properly, we encountered one final problem, that had be resolved. We packaged the new IP and replaced with it the old IP, which was deleted from the design. After implementing and downloading

the design to the device however, we realised that there was still a problem, since we could not obtain any results in the output.

Inserting breakpoints in the SDK code revealed that the problem was caused by a failure in the DMA initialization function, caused by a reset signal. We had previously encountered this problem, while incorporating the original design of G. Pitsis and G. Kalomoiris in the ExaNeSt architecture. In this case however, the cause of the problem was completely different, as we mentioned in a previous section.

Workaround attempts

The first approach to the resolution of this problem, was to assume that the IPI had assigned a base address to the new IP, without updating the value of the old IP in the memory maps of other IPs connected to it. To avoid such memory mapping issues, we returned to the previous version of the design, which produced false maximum values for the second batch and instead of deleting the old IP, to replace it with the new one, we repackaged the new IP under a different version index and used the IP status report to detect the new version and automatically upgrade the IP. This surprisingly yielded no results, even though the convolutional layer index we changed, which was the only difference in the two designs, should have no effect on the initialization of the DMAs.

The next approach was to create a new project and repeat the entire system design process. The logic behind this decision was to make sure that the project was not stuck in a faulty state, unable to apply any changes in the design. The first attempt was not successful, since the DMAs were still not being initialized, but convinced that there was no room for further errors in the design, we designed the system again in a new Vivado project. This finally brought the desired results.

Bug report

This seems to be a recurring bug in the Vivado toolchain, with reports recorded in the Xilinx forums[49], dating as far back as 2011. The bug occurs when the developer tries to initialize the DMAs, which requires their base addresses. Each DMA has a device ID, recorded in the xparameters header file of the BSP. This ID is given to the XAxiDma_LookupConfig function of the BSP, which returns the base address of the corresponding DMA. With this address we can call the XAxiDMA_CfgInitialize function of the BSP to initialize the DMA. Before returning the function calls the XAxiDma_Reset function also included in the BSP, to reset the engine, so that the hardware can start from a known state. The reset function however does not return, thus stopping the execution of the program.

The execution of the function itself never finishes, stopping after calling of the XAxiDma_WriteReg function. Why this function never returns is not clear, since it is only declared as a define directive in the BSP files. Since the bug has not been addressed, the only workaround we could find, was to start the design process again from a clean project, in our case more than once.

5.3.6 Optimizations

With all the errors now fixed, only the task of optimizing the performance of the design now remained. We implemented the design for several clock frequencies, with various performance results that will be presented in the respective chapter. The goal was to achieve the 300 MHz of the original design with batch size equal to one, or at least the 250 MHz, at which the design was running before the design errors were found.

Starting at 100 MHz, we gradually increased the frequency by 50 MHz at a time. The design worked with no issues at 150 MHz, but at 200 MHz we started encountering some timing violations. Eventually this was resolved by using different implementation strategies, which optimize the area of the physical chip of the FPGA that the design utilizes. This allows the clock signal to reach all slices quicker, enabling the use of a higher frequency clock.

At 250 MHz however, simply changing the implementation strategy did not resolve all timing violations. We had to take another approach which required the synthesis of new IP cores in Vivado HLS, this time with a higher target frequency for the clock signal. The IP cores of the CNN were already synthesized for a target clock of 250 MHz, so we tried synthesizing them for a 300 MHz and a 350 MHz clock. The 350 MHz RTL design however did not meet the timing requirements and thus the highest frequency we achieved for the design was at around 320 MHz. The next step was to export the IP cores and implement the design.

5.4 Incorporation into the ExaNeSt Design

The main objective of this thesis is to run multiple instances of our CNN design on multiple FPGAs and multiple QFDBs, using data streamed from the outside world through an EtherNet link. To achieve this objective, we first need to incorporate our design into the ExaNeSt[28] architecture of each FPGA and run it concurrently. We can confirm the correct function of the design resulting from the combination of the two projects, using a fixed data set imported through JTAG, since the micro SD cards are not available on the device. Once we have achieved this, we can boot

a specially designed Linux kernel on the processor of each MPSoC and use a virtual machine to transfer data to the board through an EtherNet link.

The ability to pass data to the QFDB through an EtherNet cable and stream these data to all FPGAs, allows the concurrent processing of large amounts of information and renders the application scalable by enabling the concurrent use of multiple QFDBs. In the following sections we will describe the merge process of the two designs, the problems that occurred and their solutions and finally we will present an performance overview of the final system.

5.4.1 The ExaNeSt Design

A significant part of this work was to gain a fundamental understanding of the ExaNeSt hardware design, the function of the IP cores contained in it and how to properly use them. The ExaNeSt design is comprised of 4 separate hardware designs, one for each of the Zynq Ultrascale+ MPSoCs on the QFDB, to which we will refer to using the names F1 to F4, with the number being the FPGA index. As we have mentioned in previous paragraphs, the F1 design implements the network infrastructure for external communications, while the F2 design implements the connection to the SSD/NVMe storage. Apart from that however, all 4 designs share a basic architecture that can be divided into 3 main IP groups.

Transceivers

This is an IP core developed in FORTH, that provides chip-to-chip communication through a high-speed low-latency communication channel between the FPGAs of the testbed. The IPwas first designed as part of the EuroServer[12] project, implementing algorithms[5] for low-latency communication protocols. The blocks were redesigned for the EuroEXA project, to allow for advanced stability and resiliency. Algorithms for link bundling and hence higher channel bandwidth were also added, resulting in a latency measurement 3x lower than the Xilinx Aurora IP. The ExaNeSt design contains 2 instances of this IP, one to support the use of ExaNet protocols and one to allow AXI-streaming.

The transceivers constitute the backbone of the interconnect between the FPGAs of the QFDB. As a result, the initial approach to the streaming of data for our CNN, was to utilize them to that end. The data would arrive to the network FPGA through an EtherNet interconnection and would then be distributed to the other FPGAs through the transceivers. This idea however was later abandoned in favor of faster solution, which provided EtherNet access to all 4 FPGAs, as we will describe in the next paragraphs.

ExaNet Protocol

ExaNet is a network architecture responsible for data communication at Tier 0/1/2 of the network interconnect of the ExaNeSt project[1], providing switching and routing features and managing the communication over the High Speed Serial links of the following interconnect levels:

- the high-throughput intra-QFDB level (Tier 0) for data transmission among the four FPGAs of the ExaNeSt node;
- the intra-Mezzanine level (Tier 1) directly connecting the network FPGAs of different nodes within the same mezzanine;
- inter-Mezzanine communication level (Tier 2) based on SFP+ connectors, allowing the implementation of a direct network among QFDBs on a rack.

In the ExaNeSt design a group of IP cores implements the ExaNet communication protocol[39]. Each node design contains a transmitter DMA and a receiver DMA[50][33], both connected to an ExaNet switch IP which routes ExaNet packets among QFDBs and their FPGAs[29]. A dedicated IP core converts the ExaNet packets into AXI4 write transactions in the receiver node and another IP core performs the opposite conversion on the transmitter node. A similar process takes place for AXI4 stream packets, using a different set of IP cores, which besides transactions has to handle the fragmentation and reassembly of the stream into discrete packets that can be transferred using the ExaNet protocol. Figure 5.6 presents the basic concept of the communication between QFDBs.

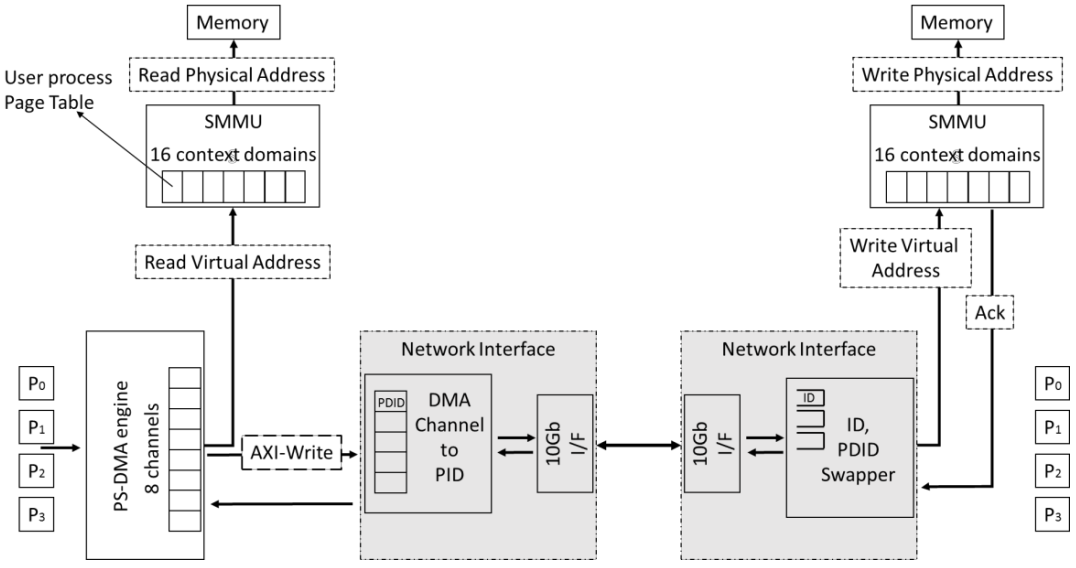

FIGURE 5.6: Inter-node communication: Picture source paper

Ethernet Infrastructure

This is a group of IPs which allow EtherNet access to all FPGAs of the QFDB[34]. Since each FPGAs traffic needs to pass through F1 in order to communicate information outside the QFDB, a custom reduction algorithm was developed in FORTH, implementing a client-server protocol between F1 and the other FPGAs on the board. A client IP core responsible for sending data to F1 for reThus we have to map our data to physical memory addresses and exclude the memory space they occupy from the address range of the OS. The DMAs of the CNN design can then access that space and stream the data.

duction, was instantiated in each MPSoC, providing the PS with an acknowledgement, once the reduction had been completed. A server IP was also instantiated in F1, responsible for the actual reduction of the data and the external communication.

This algorithm can operate for any given number of QFDBs, as long as all FPGAs participate in the reduction. We achieve this with the use of an AXI-stream switch for EtherNet frames' routing, capable of routing broadcast/multicast EtherNet packets. This 10G EtherNet switch can also be configured to support software-bridged EtherNet functionality by routing traffic to F1. It is exactly this function that will allow us to pass data to all four FPGAs using an EtherNet link.

Linux environment and Virtual Machine

Apart from the hardware design, a software environment is necessary for the use of the EtherNet infrastructure of the Exanest project. First, a Virtual Machine is booted and connected to the EtherNet network. It hosts various network services such as SSH, DHPC, DNS and two Network File Systems. Second, on each FPGA, a trimmed Linux kernel is retrieved from the attached QSPI memory. Then, the shared root Network File System is loaded, and a second, more complete, kernel is booted. We use a Linux distribution called CARVoonix, which is specifically designed for the QFDB platform. Users can then connect to the Linux instances running on the MPSoCs using command-line shell.

File sharing between the PS and the Manager PC is achieved through the use of a specific folder in the NFS, which can be accessed by both the OS of the Manager PC and by the OS on the board. However, in order to read the data into the FPGA, we need to store them in the DDR memory of the PS. Thus we have to map our data to physical memory addresses and exclude the memory space they occupy from the address range of the OS. The DMAs of the CNN design can then access that memory space and stream the data.

5.4.2 Design Restrictions

As was made clear in the previous paragraphs, the ExaNeSt design is a complex architecture comprised of numerous IPs developed by separate research teams. Thus, it is crucial to make sure that any changes required for the incorporation of the CNN into the design, do not interfere with other functions of the architecture. This imposes certain restrictions to both the available resources and to our design approach. In the following paragraphs we will describe these restrictions as they manifest in the F2 hardware design.

We choose to present the design process of the F2 over the other architectures for several reasons. First of all, the design process is similar in all four architectures, with only some minor quantitative differences, like the number of interrupt signals that each architecture uses. Moreover, incorporating our CNN in the F2 was more challenging than in the other architectures, since its connection with the SSD/NMVe storage through a PCIe interface presented a few design problems, as we will show in the next paragraphs.

This section presents the work that has been done towards the final goal of this thesis, which is the streaming of data to our CNN accelerator through an EtherNet link.

AXI ports

The Zynq Ultrascale+ processor is equipped with 2 master and 4 slave AXI High Performance Full Power Domain (HP FPD) ports, 1 master and 1 slave AXI HP Low Power Performance (HP LPD) ports and 2 slave AXI HP Cache Coherent FPD (HPC FPD) ports. The F2 design of the ExaNeSt project uses the LPD and both FPD master AXI HP ports. It also uses both slave AXI HPC ports and at least one of the slave AXI HP ports. The CNN design used for this thesis requires only 3 slave HP ports, which as mentioned are available. However, it also requires at least one master HP port. Since both the FPD and the LPD master AXI HP ports are utilized by the ExaNeSt design, it is obvious that one of them would have to be shared. In this case, we used the LPD port, as we will describe in the following section, to avoid changes in the settings of the PS.

Interrupts

The Zynq Ultrascale+ is equipped with 2 8-bit PL to PS interrupt ports. However, at least one of FPGAs of the ExaNeSt design, only uses of one of them and we were instructed to avoid altering the PS settings, to avoid possible interference with the work of other teams working on the project. Activating the second port

would probably not create any such interference, but its use was not necessary, as we will show. Thus, in at least one FPGA, only one interrupt port could be used, with less than 8 bits available, since the ExaNeSt design already uses some of them for its own modules.

The CNN design, as we know from previous sections, is comprised of 4 HLS generated IPs and 4 DMAs, 1 of which supports both read and write functions resulting in a total number of 8 1-bit interrupt signals. With less than 8 bits available, it was obvious that we could not connect all 8 signals without altering the PS settings. However, the interrupts from the 4 HLS generated IPs containing the CNN itself, are never used in the application and thus we can omit their initialization and leave them unconnected. This left us with the 4 DMA interrupt signals which we can then concatenate with the ExaNeSt interrupts and connect them to the interrupt port of the PS.

Porting Data through JTAG

In the original work of G. Pitsis, the CNN design is implemented with both the ZCU102 and the QFDB as target platforms. Although the final version of the QFDB prototype is designed to provide access to a Micro SD card to each of its 4 FPGAs, at the time of this thesis this has not yet been realised. Thus, to confirm that the CNN design had been successfully incorporated into the ExaNeSt architecture, we followed the same tactic as the original work, using JTAG to port the data into the device.

To achieve this we need to create a set of new files in the SDK application project, containing the data, as well as a set of functions to read the data from these files. We created two files, one containing the weight values of the network and one containing the input values. This of course imposes a restriction on the amount of inputs we can pass to our network, but this is not an issue since our goal, once we confirm the correct function of the design, is to bring the data in through the ExaNeSt EtherNet infrastructure.

It is worth noting the significant impact the porting of network data through the JTAG had on the time needed for the programming of the device. Programming the ZCU102, where data was read from the SD card, required less than a minute, while programming the QFDB, with all weight values as well as 2500 input values included in the application files, required more than 3 minutes. The size of the device programming files increased from 27826 kB in the ZCU102 to 56086 kB in the QFDB, justifying the delay.

Finally we need to point out that since we do not make use of a UART cable, the FPGA output is transmitted to the computer through the JTAG cable and

displayed in the computer screen in a dedicated terminal. This terminal can display data passing through the JTAG by means of the CoreSight technology, a set of tools that can be used to debug and trace software that runs on Arm-based SoCs.

5.4.3 First Approach

As we mentioned before, our first task is to incorporate the CNN design into the ExaNeSt architecture, with the restrictions we presented in the previous section. We aim to achieve the same performance as the standalone version implemented on the QFDB, since for the time being the result does not depend on the performance of other components like the EtherNet infrastructure. In the following paragraphs we will describe our approach to this task for the F2 design. The same process was repeated for the F3 and F4 hardware designs, while F1 was preserved in its original form, containing only the ExaNeSt design, for reasons we have previously explained.

AXI4 protocol

All IPs containing the CNN modules were synthesized in Vivado HLS, as we know from previous chapters and were thus equipped with AXI4 ports. These ports make memory mapping and master/slave communication, between the Zynq processor and the IP instances, possible by means of the AXI4 Interface Protocol. Thus the first IP instance to be added in the design, as a slave device of the Zynq, is the AXI Interconnect, on whose master interface we will subsequently connect the AXI ports of all other IPs.

With both the FPD and LPD master AXI HP ports of the Zynq occupied by AXI interconnects of the ExaNeSt design, it was clear that our interconnect IP had to be connected as a slave to one of them. After being instructed to keep the FPD ports reserved for the IPs of the ExaNeSt design, we were left only with the LPD port. The interconnect was thus connected as a slave device of the interconnect of the LPD port, whose number of master ports was increased by one, to accommodate this change. The number of master ports to the interconnect was set to 7, 3 for the DMAs and 4 for the IPs of the CNN. It has to be noted that, using the LPD instead of the FPD port opens up the possibility for lower performance values, but this will only occur in the highly unlikely case where the LPD port is saturated. The architecture of the network, seen in figure 5.7 remained as it was described in the original work of G. Pitsis, since our goal is to maintain the same standard of performance in our design. The IP cores were grouped together in a single hierarchy to become distinguished from the IP core of the ExaNeSt design.

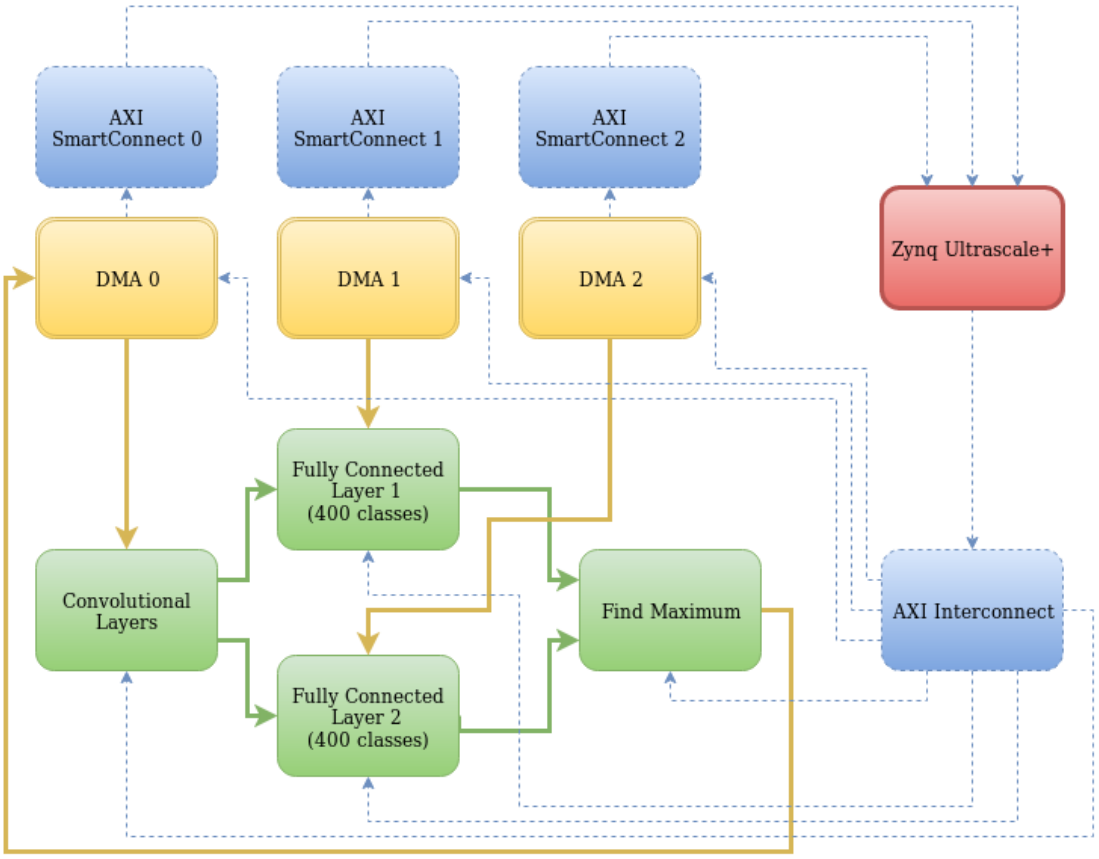

FIGURE 5.7: Batch 1 CNN standalone architecture: clock, reset and interrupt signals were omitted, to emphasize the most structurally important connections of the AXI protocol

The next step is to assign a base address for each IP using the Address Editor of the IPI. This memory mapping is essential for the function of the AXI4 interface protocol. The address assignment was done immediately after connection of the IPs to the interconnect. This way we avoided any automatic address assignment with unpredictable results.

Finally, to complete the interfacing between the PS and our design, we connect the DMAs to the slave AXI HP ports of the Zynq. This is done with the use of an AXI SmartConnect IP core, which, like the AXI Interconnect IP core, connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices. Even though a single SmartConnect should theoretically be sufficient for the design to work, it would connect all 3 DMAs to all 3 HP ports, which not only is a waste of resources but would probably result in an increase in latency.

IP Integration

With our IPs added in the design, we used the autoconnect function of the IPI to automatically provide them with the necessary clock and reset signals. The

next step is to change the IP settings to accommodate the streaming of data in a manner consistent to the architecture of the IPs.

The data, as we know from the computational model, are streamed from the DMAs. More specifically, the first DMA streams the kernel matrix and the input values of the network into the IP core containing the convolutional layers through a 32 bit read channel. It also has a 32 bit write channel which receives the output values from the IP core finding the maximum neuron value of the fully connected layer. As for two remaining DMAs, they each stream the weight values into each of the two IP cores containing the fully connected layers, through a 128 bit read channel, while their write channel is disabled.

As we mentioned in a previous section, we only make use of the interrupts of the DMAs, leaving the interrupts of the CNN IP cores unconnected due to restrictions in our design. The interrupts of the DMAs are concatenated into a single 4 bit signal and subsequently driven to the Zynq. Once we were done with the DMAs, we connected the signals streaming data from one IP of the CNN design to the next, which concluded the integration of the CNN in the ExaNeSt design.

Application Project

Since the aim of this thesis is the use of an embedded OS for the transfer of the input data to the FPGA through an EtherNet link, we cannot use the SDK to run our accelerator. That is because the PS of the MPSoC will be utilized by the embedded OS, thus preventing the Vivado SDK from running on it. However, before we can boot the OS on the PS we must first confirm that the incorporated CNN design is working. To do this, we need to run it as a baremetal application using the Vivado SDK.

The SDK code we will use is the same as the one used in the standalone CNN architecture for the QFDB, as it is described in the thesis dissertations of G. Pitsis and G. Kalomoiris. Much like the SDK code for the "Batch 2" architecture, this code can also be divided in three main structural parts. In the first part we allocate the required memory space in the DDR. In the second part we read the data in the DDR, an operation which in the standalone design targeted to the ZCU102 was carried out with data stored in an SD memory card. In the QFDB however we do not have access to the SD card so we read the store the data as variables in a .c file and transfer them to the QFDB trough the JTAG cable, as part of the software application. Thus in the second part of the code we read the data from the aforementioned file into the DDR. Finally in the third part of the SDK code, we configure and run our hardware accelerator and transfer the data to its

IP cores with the help of a dedicated BSP function, that executes simple transfer operations from the DDR to the PL.

The programming of the FPGAs on the QFDB is done using dedicated commands, which run XSDB scripts created in CARV specifically for this task. Since this is a baremetal application the QFDB is programmed from JTAG using only the .elf file containing the SDK software application, the bistream file and the PSU initializer file.

5.4.4 Troubleshooting

After creating the application project using the code of the standalone QFDB application, several issues, both with the SDK code and the design itself were revealed, either immediately or at runtime. We will describe these issues and how they were resolved, in the following paragraphs.

Address Mapping

The first issue we encountered was a set of compiler errors in the SDK code, which showed that simply using the same code as the standalone QFDB application would not suffice. The root of this issue was the grouping of the IPs of the CNN design into a single block, which caused a different name assignment to the variables containing the base addresses of the aforementioned IPs. Thus when the driver and bitstream files were exported to the SDK and used in conjunction with the SDK code of the standalone QFDB application, the compiler could no longer locate the base addresses of the IPs. The solution to this problem was simple. For every variable containing the base address of an IP, whose name had no definition, we simply needed to locate the name of the variable actually containing the base address of that same IP and substitute the former variable name with the latter. This resolved the compiler errors and allowed the SDK tool to build the application project and produce the files required for the programming of the device.

PCIe configuration

The second issue was revealed while trying to program the device. The tcl responsible for the programming of the FPGAs, yielded an error during the initialization of the board. It was thus obvious that an error, which did not affect the implementation process, existed in the design. After a detailed review of the design and IP settings, we turned to the error code. To determine the source of the problem we had to resort to a form of manual Fault Tree Analysis, using the Xilinx System Debugger (xsdb) scripts developed in CARV to allow the use of the QFDB.

The sequence of successfully executed tcl commands in the script ended when we tried to access an address related to the PCIe interface. With the guidance of Dr. Fabien Chaix, author of the tcl script, it was determined that the address producing the error was connected to the SSD/NMVe storage. At this point a physical check of the board revealed not only that the SSD/NMVe was missing, but there was also was no PCIe port to connect one. A few QFDB prototype boards, like the one we used initially, did not support this feature, leaving the corresponding address unconnected. Since however the PCIe setting on the ExaNeSt design was enabled, when the PS tried to poll the device connected to that address an error was produced.

To remedy this, we disabled the PCIe setting on the PS, a change which remained even after we moved on to working on a board with storage capabilities. This allowed the use of the same bitstream file on both QFDBs used for the purposes of this thesis, making the task of running our application on multiple QFDB boards much less time consuming. Altering the PS settings however, most likely affects other functions of the design. It is thus recommended that the PCIe setting on the PS be enabled for any further application.

Processor System Reset Autoconnect

With the PCIe setting disabled we were able to program the device successfully and run the application. We confirmed that the program was running through a printed message at the start of the code, but we could not see any results printed in the output.

Through a set of breakpoints in the code we discovered that the initialization of the DMAs was never completed, which derived the design of its ability to stream data into the network. During initialization the DMAs are reset so that the hardware can start from a known state. Since the reset of the DMA was never completed we decided to review the source of the reset signals of the DMAs we used.

As presented in a previous paragraph, the ExaNeSt design has 3 main reset sources, namely the Transceiver reset, the ExaNet reset and the PL reset. Since our network is neither part of the ExaNet protocol nor the Transceiver infrastructure, the reset source should be the PL reset module. However the reset signal connected to the DMAs was the ExaNet reset, probably as a result of using the Autoconnect function of the Vivado IPI. Once we rectified this mistake and run the implementation process again, we run our application on the device, which finally yielded the correct output. Having confirmed that our bare metal design is working as intended we proceeded to set up the EtherNet infrastructure.

5.5 Streaming Data through Ethernet

This section presents the work that has been done towards the final goal of this thesis, which is the streaming of data to our CNN accelerator through an EtherNet link.

5.5.1 Ethernet Infrastucture

In previous paragraphs we described the communication protocols of the QFDB implemented as part of the ExaNeSt project. The hardware design alone of course is not enough to achieve this communication. Both in the case where QFDBs need to communicate with each other and the case where they need to communicate with the outside world, a Manager PC is needed to coordinate their actions and manage the boards. The Manager PC is connected to a switch together with the QFDBs, with all connections using the HSS Links we mentioned in previous paragraphs. Access to the embedded OS of each MPSoC on each QFDB is given by a virtual machine, as we will describe in the following paragraphs. In figure 5.8 we can see an outline of our system and its function.

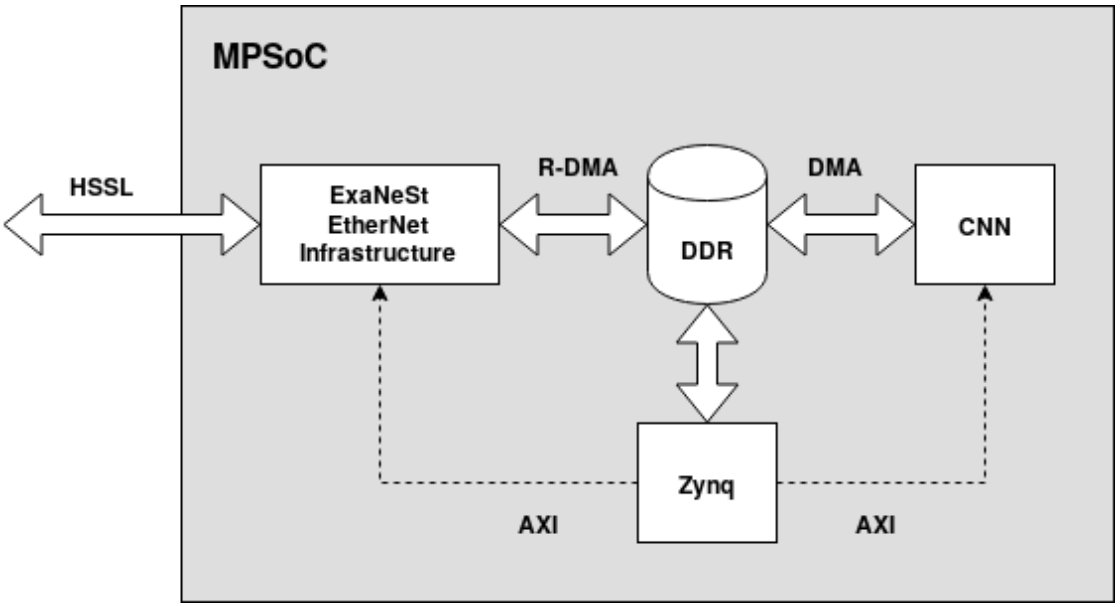

FIGURE 5.8: Streaming data to the CNN through EtherNet

5.5.2 Linux Porting

Since the QFDB is a custom prototype board, the Xilinx tools do not recognise it and thus programming its MPSoCs both as a baremetal application and as an embedded OS application has to be done by some other means. To that end, a series

of programming XSDB scripts were created by the CARV laboratory team, along with other configuration and diagnosing scripts. In our case, we need to configure the Zynq Ultrascale+ as a Linux Operating System Platform, since without an embedded OS we cannot transfer files through the EtherNet link[25].

As we have mentioned before, CARV has developed a Linux distribution called *CARVoonix*, specifically designed to run on the QFDB. A second trimmed Linux kernel, nicknamed *loby* was also developed and stored in the local QSPI flash memory. With the QFDB programmed from QSPI, the full Linux system is booted by loby, which mounts a root NFS storage common to all boards of the prototype[3]. This root FS contains a configuration file that maps each QFDB to a board class. Once the board class and the node ID within the QFDB have been retrieved, the corresponding boot package, which contains a bitstream and a device tree, is loaded.

In practice, most of these actions are carried out automatically by the XSDB scripts mentioned previously. Running these scripts involved the use of only two QFDB-specific commands, which demonstrates how user-friendly the platform is.

5.5.3 Virtual Machine

With the Linux booted on the PS, we now need a UI to provide us with access to the file system. This is accomplished through a virtual machine, provided by the CARV laboratory. The VM is imported on the Manager PC with preconfigured settings to make the connection with the QFDB as effortless as possible. The Manager PC must run on some distribution of Linux, preferably CentOS, since the VM is designed for the Virtual Machine Manager application.

Once a connection has established between the QFDB and the VM, a command line is displayed which can be used to navigate the file system of the embedded OS. Through this terminal we can transfer files from the Manager PC to the DDR memories of the QFDB and vice versa. The files need to be placed in a designated folder, the shared home folder, which provides write priviledges to both the Manager PC and the embedded OS. Furthermore, we can use the terminal to run software applications for our accelerator and configure the PL, as we will show in the next paragraph.

5.5.4 Data Porting

The final step to stream data to our accelerator through an EtherNet link is the software application. Since we are no longer running a baremetal application, we can not use the SDK and its BSP packages to initialize our IPs and stream the

data in the hardware design. The software application now has to be written as a .c program and executed on the embedded OS with the help of the VM.

Before we can write a software application to stream our data, we must first make sure that the PS can communicate with the IP cores of our accelerator. In the SDK this was done almost automatically, with BSP functions that could identify an IP core by using its base address, provided by the bitsream file. Since we can no longer use BSP packages, the base address of each IP must be registered manually on the device tree of the embedded OS.

The next step is to transfer the files containing our data from the Manager PC to the DDR of the PS through the EtherNet connection. We want these data to be accessible by the PL side of the MPSoC, since that is where our accelerator is running. The DDR memory of the PS however is occupied by the embedded OS and is thus unavailable to the PL. To resolve this impasse, we have to exclude a section of the memory from use by the OS and allocate it for PL access. We use the mmap function to write data on physical memory addresses of that section, something which, for security reasons, would not be allowed had we not excluded this section from use by the OS. We also need to use the mmap function to register the base addresses of the IP cores of our accelerator in the memory.

With our data in place and the accelerator IP cores registered in the device tree we are now free to run our software application, which should not differ much from the one written in the SDK code.

It is important to note this work is still in progress and will be included as a proposition for future work. Although editing the device tree and using mmap to access physical memory are fairly well documented tasks and the software application program has been modified to run on an embedded OS, we could not confirm their correct function. This is due to technical issues with the QFDB that prevented us from running the accelerator. As a consequence no results were obtained and as of this time the correct function of the software application has not been verified in practise.

5.6 Overall Design Issues

In this section we will provide a summary of work done on ExaNeSt design, identify some important aspects of this work, explain some design choices that might be suboptimal and propose the necessary optimizations to improve the overall system performance.

5.6.1 Clock Frequency

All 4 ExaNeSt designs were implemented with a clock frequency of 100 MHz, which is the frequency the designs had upon their delivery from CARV. However, we know for a fact that both the CNN accelerator and the ExaNeSt designs were implemented for far higher clock frequencies. For example, in the original work of G. Pitsis, the clock frequency of the CNN accelerator design is 300 MHz, producing a significant speedup over state of the art GPUs.

The designs were not implemented for a higher clock for a number of reasons. First of all, at the time of this work the ExaNeSt project was still under development and it was suggested to us not to change any of the PS settings, to guarantee both compatibility with the QFDB and functionality of the underlying infrastructure. Increasing the clock frequency or making use of a second PL clock would not only require using a second clock source on the PS but a new reset signal generator IP core would also have to be added, which would work on the new PL clock frequency.

Furthermore, the main task of this thesis was to achieve functionality of the merged designs, with optimization being a secondary task. Implementing the design for a higher clock frequency would require more resources and especially routing nets, which could create signal congestion problems and timing violations, that would then have to be resolved. This is a time consuming process than could not have possibly been completed in time.

The actions needed to achieve a higher clock frequency are simple, but need to be carried out in cooperation with the CARV team, to avoid any problems with ExaNeSt design. The first approach is to increase the PL clock frequency which at the moment is shared between the CNN accelerator IP cores and the IP cores of the ExaNeSt design. The second, far safer approach is to introduce a second PL clock on the design and replace with it the clock on all IP cores related with the CNN accelerator.

5.6.2 Full Power Domain Ports

In contrast to the original design in G. Pitsis' work, we use the Low Power Domain Ports on the master side of the Zynq Ultrascale+. Although both the FPD and LPD ports are occupied on the ExaNeSt designs, we were instructed by the CARV team to use the LPD ports, since the CNN accelerator only used the master ports for AXILite connections. However both Xilinx manuals and Xilinx Forum posts suggest that using the FPD instead of the LPD ports will produce better performance results.

This task is also simple, since we only have to transfer and connect the AXI Interconnect of the CNN accelerator with the AXI Interconnect of the FPD ports. We would suggest repeating the address assignment process afterwards, to avoid address conflicts caused by the difficulty Vivado has on automatically updating the base addresses of the IP cores.

5.6.3 SmartConnect IP core Reset signal

Although both the "Batch 1" and the "Batch 2" architectures we inherited, use the same peripheral reset signal as all other IP cores in the design, the correct practise is to use the same interconnect reset signal as the AXI Interconnect IP core in the design uses. In all the designs we kept the reset signal unchanged, treating the design as a black box to avoid changes that might compromise an already functional design. This however could create problems to the R-DMA of the EtherNet infrastructure during data streaming, once the software application is completed. Thus we highly advise changing the reset source on the SmartConnect IP cores.

Chapter 6

Experimental Results

In this capter we will present the results obtained from the hardware designs implemented for the purposes of this thesis. We will describe the resource utilization of each design and the present significant performance metrics. We will focus mainly on the resource utilization and power metrics since the performance measurements of a non-optimized design do not reflect its full potential.

6.1 ExaNeSt

As we know the ExaNeSt project is comprised of 4 hardware designs, one for each MPSoC of the QFDB. In this section we will present a set of resource utilization, power consumption and performance metrics and compare the results of the "Batch 1" architecture both between different platforms and between different hardware designs of the same QFDB. All designs on this section were implemented with the default frequency of 100 MHz, for reasons we have mentioned before.

6.1.1 Resource Utilization

Table 6.1 shows a comparison between the resource utilization of the ZCU102-targeted design, the the QFDB-targeted standalone design and the QFDB-targeted design after its incorporation in the ExaNeSt design. In this example we use the measurements from the F2 design since this is the first of the four designs we incorporated the CNN design into.

As we can see, migrating from the ZCU102 to the QFDB does not affect the resource utilization at all, with the peculiar exception of the decrease in Global Clock Buffer (BUFG) usage. This of course is to be expected as both platforms use the Zynq Ultrascale+. However, the difference between the standalone designs and the design after its incorporation in the ExaNeSt design is obvious and is of course to be expected. We mark an increase in Look-Up Tables (LUT) utilization by 18%, in LUTRAM by 2%, in Flip-Flops (FF) by 10% and in Block RAM by

14%, while BUFG returns to the same utilization percentage as in the ZCU102 design. The reason for this increase of course is the resources required by the other IP cores of the ExaNeSt design.

Table 6.1: Utilization Comparison between platforms

	ZCU102 (%)	**Standalone QFDB (%)**	**ExaNeSt QFDB (%)**
LUT	40	40	58
LUTRAM	2	2	4
FF	39	39	49
BRAM	12	12	26
DSP	19	19	19
IO	0	0	1
GT	0	0	25
BUFG	6	1	6

Table 6.2 presents the resource utilization after the incorporation of the CNN architecture into the ExaNeSt designs, with the exception of the F1. As we can see the values are almost identical, with some minor differences attributed to differences between the ExaNeSt designs and to the non-deterministic nature of the algorithms Vivado utilizes to implement the designs.

Table 6.2: Utilization Comparison between ExaNeSt designs

%	**LUT**	**LUTRAM**	**FF**	**BRAM**	**DSP**	**BUFG**
F1	56	1	30	29	0	8
F2	58	4	49	26	19	6
F3	59	4	50	30	19	7
F4	56	3	46	19	19	6

Of course, as we mentioned in a previous chapter, we did not incorporate the CNN successfully in the F1 design due to timing violations, caused by routing congestion. Table 6.2 compares the utilization of resources before and after the incorporation of the CNN architecture. We notice a particularly high value in LUT utilization, reaching 94%. It has been noted that for particularly dense designs, like the ones used to implement MAC operations, exceeding 75% utilization in any of the significant metrics, like LUTs, FFs, or DSPs, results in timing violations.

Table 6.3: Utilization Comparison

%	**LUT**	**LUTRAM**	**FF**	**BRAM**	**DSP**	**IO**	**GT**	**BUFG**
F1	56	1	30	29	0	3	81	8
CNN-F1	94	4	68	41	19	3	81	8

6.1.2 Power and Performance

Table 6.4 presents the power consumption for each of the four ExaNeSt designs. The differences are caused by both the non-deterministic implementation algorithms used by Vivado and the different functions of ExaNeSt included in each design.

Table 6.4: Power comparison between ExaNeSt designs

	F1	**F2**	**F3**	**F4**
Clock Frequency(MHz)	100	100	100	100
Total On-chip Power(Watt)	10.037	9.345	9.193	8.998
Energy Consumption(Joule)	76.28	71.02	69.87	68.38

Table 6.5: Measurements for 2500 input value dataset

Table 6.6 presents the power and performance measurements of the CNN accelerator incorporated in the F2 design for an 2500 input value dataset, with a total runtime of 7.45 seconds. As we can see the throughput is significantly lower than the that presented in G. Pitsis' original thesis, owing to the fact that we are running the accelerator on a much lower clock frequency.

Table 6.6: Performance of the CNN on the F2 design of ExaNeSt

Clock Frequency (MHz)	Power (Watt)	Energy Consumption (Joule)	Throughput (Images/s)	Latency (ms)
100	9.523	71.02	329	3.037

Table 6.7: Measurements for 2500 input value dataset

6.2 Batch 2

In this section we will present the results obtained from the now functional "Batch2" architecture of the CNN, first described in the thesis of G. Pitsis. We will describe the performance, power and resource utilization measurements for both the 100 MHz and the 150 MHz designs. Since however the quickest clock frequency achieved is 150 MHz, we will focus more on the resource utilization. An optimization of the architecture, with the goal of achieving a higher clock frequency, could however produce significant performance results as well.

6.2.1 Resource Utilization

Table 6.8 presents the resource utilization of the "Batch 2" architecture on the ZCU102 evaluation board for a clock frequency of 150 MHz, the highest frequency for which we managed to successfully implement the design.

Table 6.8: Batch 2 design performance

LUT (%)	LUTRAM (%)	FF (%)	BRAM (%)	DSP (%)	BUFG (%)
65	4	50	17	39	10

6.2.2 Power and Performance

Table 6.9 presents the power and performance measurements of the Batch 2 accelerator running on the ZCU102 platform for a 5000 input value dataset. We present here the results for a 150 Mhz clock frequency, pending results from the 200 MHz clock design.

Table 6.9: Measured HSSL performance

Clock Frequency (MHz)	Power (Watt)	Energy Consumption (Joule)	Throughput (Images/s)	Latency (ms)
150	9.523	96.182	495	2.015

Table 6.10: Measurements for 5000 input value dataset

Chapter 7

Conclusions and Future Work

In this chapter we will sum up and evaluate the work that has been completed in this thesis. We will also describe subjects that this thesis opens up for future work and any aspects of it which merit further optimization.

7.1 Conclusions

With the advent of exascale supercomputers, the demand for applications with a corresponding amount of workload is increasing. This thesis works towards providing such an application, by pairing an existing Convolutional Neural Network hardware design with an existing exascale prototype, all the while demonstrating the challenges of working on a prototype platform. Although the initial goal, which was to run the application on multiple platforms, was not achieved, we managed to obtain functionality with both designs working in tandem and we have set the foundation for the streaming of data to the platform from an external source. Furthermore, part of this thesis was dedicated to fixing an improved version of the application design, paving the way for the use of an even more computationally demanding application on the prototype platform. It is worth noting, that after encountering several issues with the development tools, we have concluded that anyone interested in continuing this work, should migrate the design to a newer version of the Vivado suite. Finally this thesis manages to implement Monte Carlo Dropout during prediction time and use the results to increase the confidence in the results of the network.

7.2 Future Work

This work opens up several possibilities for future work, since its purpose was to achieve functionality on a prototype platform, making optimization much more challenging. We would like to propose the following tasks for further development:

- As we have mentioned before the software application of our design has not yet been tested in an actual QFDB board. Thus they first task for anyone who wishes to continue this work, should be to run the program and resolve any possible errors it might contain. This is a fairly simple software debugging task that will finally allow the use of ethernet for the streaming of data on our design.

- Incorporating our CNN architecture into the successor of the ExaNeSt design, targeting the VU9 FPGA of the CRDB board, currently under development. At this point it is unclear, whether a simple migration process would be sufficient or if a bottom up redesign of the hardware architecture would be necessary, using this work simply as a guide. This will depend highly on the respective changes in the ExaNeSt design.

- Optimization of the "Batch 2" architecture for a higher clock. Although the "Batch 2" architecture is now working, we have not managed to optimize its implementation for the initial target clock frequency of 250 MHz. Achieving this might be as simple as finding the more suitable implementation strategy or it might involve a different approach in the HLS-level design.

- All four ExaNeSt designs used for the purposes of this thesis were running at a clock frequency of 100 MHz, while the CNN design we incorporated in them was designed for a target clock frequency of 300 MHz. Obviously this is a significant difference in performance and since better performance is the main reason for implementing a hardware accelerator, it is important to make this task a priority. It is however not a simple task as it would have to be completed in tandem with the ExaNeSt team, to make sure that a higher clock frequency or the use of a different clock source does not interfere with the function of the QFDB.

- Incorporating the "Batch 2" architecture in the ExaNeSt design, which would result in a 2× speedup with regards to the present design, due to parallelization. Though this task would be fairly simple, optimizing resource utilization especially for higher clock frequencies might prove challenging.

- With the data streamed into the accelerators on the QFDB through ethernet, the next and final step, to achieving a scale out of the application, is to use this design on multiple QFDBs concurrently. This task involves a fair amount of work on the software of the Manager PC, or the implementation of a new QFDB-based design to distribute the external input stream to the QFDB cluster.

References

[1] Roberto Ammendola et al. *Large Scale Low Power Computing System - Status of Network Design in ExaNeSt and EuroExa Projects*. Apr. 2018. URL: https://arxiv.org/pdf/1804.03893.pdf.

[2] *Aurora Fact Sheet*. Argonne National Laboratory. URL: https://en.wikichip.org/w/images/b/b9/aurora-fact-sheet.pdf.

[3] F. Chaix et al. *Implementation and Impact of an ultra-compact multi-FPGA board for Large System Prototyping*. 2019. URL: https://h2rc.cse.sc.edu/papers/academic_2_2_Chaix.pdf.

[4] Dan C. Cireşan et al. "Flexible, High Performance ConvolutionalNeural Networks for Image Classification". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence* 2 (July 2011), pp. 1237–1242. URL: http://people.idsia.ch/~juergen/ijcai2011.pdf.

[5] Caroline Concatto et al. *A CAM-free Exascalable HPC Router*. 2018. URL: https://drive.google.com/file/d/1aQd_K6cmLVMkqTWSLLLjjQ7XxQcbisij/view.

[15] Tom Fawcett. "Introduction to ROC analysis". In: *Pattern Recognition Letters* 27 (June 2006), pp. 861–874. URL: https://people.inf.elte.hu/kiss/13dwhdm/roc.pdf.

[16] Michael Feldman. *China Fleshes Out Exascale Design for Tianhe-3 Supercomputer*. 2019. URL: https://www.nextplatform.com/2019/05/02/china-fleshes-out-exascale-design-for-tianhe-3/.

[17] *Frontier Spec Sheet*. Oak Ridge Leadership Computing Facility. May 2019. URL: https://www.olcf.ornl.gov/wp-content/uploads/2019/05/frontier_specsheet.pdf.

[18] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36 (Jan. 1980), pp. 193–202. URL: https://www.rctn.org/bruno/public/papers/Fukushima1980.pdf.

[19] Yarin Gal and Zoubin Ghahramani. *Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning*. 2016. URL: https://arxiv.org/pdf/1506.02142.pdf.

[20] Hao Gao. *A Walk-through of AlexNet*. 2017. URL: https://medium.com/@smallfishbigsea/a-walk-through-of-alexnet-6cbd137a5637.

[21] Georgios Goumas. "EuroEXA: Co-designed Innovation and System for Resilient Exascale Computing in Europe: FromApplications to Silicon". In: 2019. URL: https://events.prace-ri.eu/event/850/sessions/2573/attachments/883/1564/7_15.05_14.30_EuroHPC_Goumas_EuroEXA.pdf.

[22] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. URL: https://arxiv.org/pdf/1510.00149.pdf.

[23] Nicole Hemsoth. *Details Emerge on Knights Hill Based Aurora Supercomputer*. 2015. URL: https://www.nextplatform.com/2015/04/09/details-emerge-on-knights-hill-based-aurora-supercomputer/.

[24] Charlie Hewitt. *Confidence measures for CNN classification using Gaussian processes*. 2018. URL: https://chewitt.me/Papers/CTH-CNN-Conf-2018.pdf.

[25] P. Hopton et al. "Compact Packaging and Liquid Cooling Technology for Exascale". In: 2018. URL: http://www.exanest.eu/pub/hopton_exaWrksh18manch_liquidCool.pdf.

[26] Jefkine. *Backpropagation In Convolutional Neural Networks*. 2016. URL: https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/.

[27] Ioannis Kalomoiris et al. *An Experimental Analysis of the Opportunities to Use Field Programmable Gate Array Multiprocessors for On-board Satellite Deep Learning Classification of Spectroscopic Observations from Future ESA Space Missions*. 2019. URL: http://users.ics.forth.gr/~tsakalid/PAPERS/CNFRS/2019-ESA.pdf.

[28] M. Katevenis et al. *The ExaNeSt Project: Interconnects, Storage, and Packaging for Exascale Systems*. 2016. URL: http://users.ics.forth.gr/~nchrysos/papers/ExaNeStDSD2016.pdf.

[29] Manolis Katevenis. "Cluster Communication Latency: towards approaching its Minimum Hardware Limits, on Low-Power Platforms". In: 2017. URL: http://www.exanest.eu/pub/katevenis_samos17_stamatisVsymp.pdf.

[30] Kevin Kiningham, Michael Graczyk, and Athul Ramkumar. *Design and Analysis of a Hardware CNN Accelerator*. 2017. URL: http://cs231n.stanford.edu/reports/2017/pdfs/116.pdf.

[31] Alex Krizhevsky. *One weird trick for parallelizing convolutional neural networks*. 2014. URL: https://arxiv.org/pdf/1404.5997v2.pdf.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*. 2012. URL: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[33] J. Lant et al. *Receive-side notification for enhanced RDMA in FPGA-based networks*. 2019. URL: https://www.researchgate.net/publication/333082000_Receive-Side_Notification_for_Enhanced_RDMA_in_FPGA_Based_Networks.

[34] Joshua Lant et al. *Enabling Shared Memory Communication in Networks of MPSoCs*. 2018. URL: https://drive.google.com/file/d/1Ts-5nJkqRDVdPSTRNZK0oXIkXODAKcwx/view.

[36] Sunita Nayak. *Understanding AlexNet*. 2018. URL: https://www.learnopencv.com/understanding-alexnet/.

[37] Antonios Georgios Pitsis. "Design and Implementation of an FPGA-Based Convolutional Neural Network Accelerator". 2018. URL: https://dias.library.tuc.gr/view/79094.

[38] George Pitsis et al. *Efficient Convolutional Neural Network Weight Compression for Space Data Classification on Multi-FPGA Platforms*. 2019. URL: http://users.ics.forth.gr/~greg/Docs/ICASSP2019b.pdf.

[39] Manolis Ploumidis et al. *Software and Hardware co-design for low-power HPC platforms*. June 2019. URL: https://www.researchgate.net/publication/333809437_Software_and_Hardware_co-design_for_low-power_HPC_platforms.

[40] Depei Qian. "China's HPC development: a brief review and perspectives". In: 2019. URL: https://www.hpci-office.jp/conference/201711/invited_talks3.pdf.

[41] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[42] Anders Sandberg and Nick Bostrom. *Whole Brain Emulation - A Roadmap*. 2008. URL: http://www.fhi.ox.ac.uk/brain-emulation-roadmap-report.pdf.

[44] Nitish Srivastava et al. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. 2014. URL: http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf.

[45] Tobias Sterbak. *Model uncertainty in deep learning with Monte Carlo dropout in keras*. 2019. URL: https://www.depends-on-the-definition.com/model-uncertainty-in-deep-learning-with-monte-carlo-dropout/.

[46] Radamanthys Stivaktakis et al. *Convolutional Neural Networks for Spectroscopic Redshift Estimationon Euclid Data*. 2018. URL: https://arxiv.org/pdf/1809.09622.pdf.

[47] Instituto Nazionale di Fisica Nucleare APE lab team. *European Exascale System Interconnect and Storage*. 2019. URL: https://apegate.roma1.infn.it/?page_id=656.

[48] Ruibo Wang. "Tianhe-3 and the exascale road in China". In: 2019. URL: https://www.r-ccs.riken.jp/R-CCS-Symposium/2019/slides/Wang.pdf.

[50] Pantelis Xirouchakis et al. *Low latency RDMA for High Performance Computing on ARM platforms*. 2017. URL: https://figshare.com/articles/Low_latency_RDMA_for_High_Performance_Computing_on_ARM_platforms/8869355/1.

[51] Jingheng Xu. "Tianhe-3 and the exascale road in China". In: 2019. URL: https://events.prace-ri.eu/event/850/contributions/702/attachments/959/1543/3_15.05_Chinese_Update_Jingheng_Xu.pdf.

External Links

[6] *DEEP project prototypes*. URL: https://www.deep-projects.eu/hardware/prototypes.html.

[7] *ECOSCALE*. URL: http://www.ecoscale.eu/.

[8] *Energy Exascale Earth System Model - Long Term Roadmap*. URL: https://e3sm.org/about/vision-and-mission/long-term-roadmap/.

[9] *Euclid Consortium*. URL: https://www.euclid-ec.org/.

[10] *EuroExa*. URL: http://www.euroexa.eu/.

[11] *European High-Performance Computing Handbook 2018*. URL: https://www.etp4hpc.eu/pujades/files/ETP4HPC_Handbook_2018_web_20181210.pdf.

[12] *EuroServer project*. URL: www.euroserver-project.eu.

[13] *ExaNeSt*. URL: http://www.exanest.eu/.

[14] *ExaNoDe*. URL: http:www.exanode.eu/.

[49] *Xilinx Community Forums - DMA bug*. URL: https://forums.xilinx.com/t5/Processor-System-Design/Program-stuck-at-DMA-config/m-p/1045463#M49845.