TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Design and Implementation of an FPGA-Based CNN Architecture for on-Board Satellite Processing of Data from the Euclid Space Telescope

---

Author:
Ioannis KALOMOIRIS

Thesis Committee:
Prof. Apostolos DOLLAS
Prof. Sotiris IOANNIDIS
Asst. Prof. Grigorios TSAGKATAKIS (U.Crete)

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

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

September 15, 2024

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Design and Implementation of an FPGA-Based CNN Architecture for on-Board Satellite Processing of Data from the Euclid Space Telescope

by Ioannis KALOMOIRIS

Convolution Neural Networks (CNNs) have been widely employed for various AI tasks and have demonstrated state-of-the-art performance, especially in complex image recognition problems. The widely used for these tasks GPUs, although having a lot of computational power, come with very high power consumption. This is a deterrent factor for their usage, especially in cases where a small energy footprint is important, like on-board signal processing. In this thesis, we demonstrate an FPGA architecture implemented for the inference stage of a specific CNN, enabling the estimation of the galaxy redshift from spectroscopic observations by dividing the redshift range into 800 Classes. The proposed FPGA architecture achieved an improvement in energy efficiency of up to 11.9x alongside a 2.16x throughput speedup over GPU platforms. The results are from actual executions on FPGAs with space-qualified equivalent parts, enabling performing accurate redshift estimation in space with low energy cost, with no need for raw data transmission to the ground.

This work is partially supported by the EuroExa Project [1].

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Σχεδίαση και Υλοποίηση Αρχιτεκτονικής Συνελικτικών Νευρωνικών Δικτύων Βασισμένη σε Αναδιατασσόμενη Λογική για επί τόπου Επεξεργασία σε Δορυφόρο Δεδομένων από το Διαστημικό Τηλεσκόπιο Euclid

by Ioannis KALOMOIRIS

Τα Συνελικτικά Νευρωνικά Δίκτυα (ΣΝΔ) χρησιμοποιούνται ευρέως για διάφορες εργασίες τεχνητής νοημοσύνης και έχουν επιδείξει κορυφαίες επιδόσεις, ιδίως σε πολύπλοκα προβλήματα αναγνώρισης εικόνων. Οι ευρέως διαδεδομένες για αυτές τις εργασίες κάρτες γραφικών (GPU), αν και διαθέτουν μεγάλη υπολογιστική ισχύ, έχουν πολύ υψηλή κατανάλωση ενέργειας. Αυτό αποτελεί αποτρεπτικό παράγοντα για τη χρήση τους, ιδίως σε περιπτώσεις όπου είναι σημαντικό το μικρό ενεργειακό αποτύπωμα, όπως η επεξεργασία σήματος εντός συστημάτων. Στην παρούσα εργασία, παρουσιάζουμε μια αρχιτεκτονική FPGA που υλοποιήθηκε για το στάδιο συμπερασμού ενός συγκεκριμένου ΣΝΔ, το οποίο καθιστά δυνατή την εκτίμηση της ερυθράς μετατόπισης γαλαξιών από φασματοσκοπικές παρατηρήσεις, διαιρώντας το εύρος της ερυθράς μετατόπισης σε 800 κλάσεις. Η προτεινόμενη αρχιτεκτονική πέτυχε βελτίωση της ενεργειακής απόδοσης έως και 11,9 φορές, παράλληλα με βελτίωση της απόδοσης κατά 2,16 φορές σε σχέση με τις πλατφόρμες GPU. Τα αποτελέσματα προέρχονται από πραγματικές εκτελέσεις σε FPGAs, οι οποίες έχουν ισοδύναμα εξαρτήματα που έχουν πιστοποιηθεί για χρήστη στο διάστημα, επιτρέποντας έτσι την εκτέλεση ακριβούς εκτίμησης της ερυθράς μετατόπισης στο διάστημα με χαμηλό ενεργειακό κόστος, χωρίς την ανάγκη μετάδοσης ακατέργαστων δεδομένων στο έδαφος.

Η συγκεκριμένη δουλειά υποστηρίζεται εν μέρει από το πρόγραμμα EuroExa [1].

Acknowledgements

I would like to thank my advisor, Prof. Apostolos Dollas for his patience, guidance, continuous support, and the opportunity to work on state-of-the-art technology and research.

I would like to thank my co-partner in this project, Giorgos Pitsis, for his excellent collaboration throughout this procedure.

Additionally, I would like to thank Dr. Christos Kozanitis and Dr. Aggelos Ioannou of the FORTH Institute for their support and cooperation during this project and for giving me the opportunity to be part of their team.

Also, I would like to thank Prof. Sotirios Ioannidis and Asst. Prof. Gregory Tsagatakis (U Crete), for their interest in my work and for accepting to be on my dissertation committee.

In addition, I would like to thank Vasilis Amourgianos, Paylos Malakonakis, Kostas Malavazos, and all the members of the Microprocessor and Hardware lab for the time they invested in helping me and for their technical support.

Finally, I would like to thank my parents, George and Maria, for everything they offered me over the years.

This thesis is dedicated to them.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ALU	**A**rithmetic **L**ogic **U**nit
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BRAM	**B**lock **R**andom **A**ccess **M**emory
CPU	**C**entral **P**rocessor **U**nit
CS	**C**omputer **S**cience
DDR4	**D**ouble **D**ata **R**ate type texbf4 memory
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GPU	**G**raphic **P**rocessor **U**nit
HBM	**H**igh **B**andwidth **M**emory
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
HPC	**H**ight **P**erformance **C**omputing
LUT	**L**ook **U**p **T**able
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
PL	**P**rogrammable **L**ogic
PS	**P**rocessing **S**ystem
RAM	**R**andom **A**ccess **M**emory
SDK	**S**oftware **D**evelopment **K**it
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SSE	**S**treaming **S**IMD **E**xtensions
SSD	**S**olid **S**tate **D**rive
TDP	**T**hermal **D**esign **P**ower
URAM	**U**ltra **R**andom **A**ccess **M**emory
USD	**U**nited **S**tates **D**ollar

Dedicated to my family and friends…

Chapter 1

Introduction

In recent years, the rapid advances in sensors, communications, computations, and storage have created vast collections of data, with trillions of bytes being used every day. The benefits for society from the use of these data are immeasurable, having transformed how people interact and make use of information daily. This has led to the emergence of the concept of Big Data and the need for different, state-of-the-art processing and distributing of information. Various approaches and algorithms have been proposed for this new data-driven computing era. Through this process, tremendous improvements have been achieved in Machine Learning.

Specifically, Convolutional Neural Networks have been widely used for various Artificial Intelligence tasks. Being able to achieve high accuracy and frequently outperform other AI approaches, CNNs have been employed in a wide range of applications, with object recognition and classification being among them. However, the great computation and storage complexity of those algorithms is a deterrent factor for their usage.

General-purpose CPU platforms do not offer enough computation capacity, so the main choice for neural network workloads is the power-hungry GPUs because of their high computation power and easy-to-use frameworks. Lately, FPGA and ASIC-based neural network accelerators have become a research topic. Various accelerators for CNNs have been proposed on FPGA platforms because of their advantages of high performance, fast development, and reconfigurability.

This work focuses on improvements in cost-energy-performance for the inference phase of a specific CNN using FPGA platforms. The Convolution Neural Network was originally developed by Dr.Tsagatakis as part of DEDALE

project, which addressed the problem of spectroscopic redshift estimation in Astronomy.

1.1 Motivation

Lately, as the demand for Convolutional Neural Network applications has skyrocketed because of their state-of-the-art performance, especially for complex image recognition problems, the same has happened to the energy costs for running platforms capable of executing those tasks. GPUs, although their high power consumption, are the most widely used platform for implementing such applications due to their computational power and performance. Furthermore, several applications with limited power budgets and tight latency constraints, like mobile embedded platforms, are not suited for the inference to be hosted on GPUs.

As the training of those models usually happens only once in their life cycle, but the inference happens repeatedly, reconfigurable hardware in the form of FPGAs for Convolutional Neural Network inference has become a key research topic [2] [3] because of their advantages, such as high performance, fast development, reconfigurability, and lower power consumption. There are numerous applications for these platforms, from high-performance data centers to low-power onboard image processing.

1.2 Scientific Contribution

The scientific contribution of this thesis is focused on implementing an FPGA architecture for the inference part of the given CNN, which is competitive against the best available platforms both in terms of performance (latency and throughput) and energy consumption, an important aspect since the use-case is focused on the on-board signal processing for astrophysics applications. The Convolutional Neural network has been developed and trained to enable the estimation of spectroscopic redshift by dividing the redshift range into 800 Classes.

We investigated multiple architectures, exploiting various opportunities for parallelism, better resource utilization, and limiting the memory accesses needed for the data transfer, which is a major drawback in the field of FPGA implementations. We propose architectures for both our target platforms,

ZCU102 and QFDB, with a speedup of up to 2.16x in throughput and an 11.9x improvement in terms of energy efficiency.

1.3 Thesis Outline

Finally, we present a brief overview of the content of the thesis:

- In Chapter 2, we state all the background information needed for the completion of this thesis. We give an overview of Convolutional Neural Network(CNN) and Deep learning alongside key terms.
- In Chapter 3, we present the various state-of-the-art software used for CNN and the various hardware platforms used. Furthermore, we present the related work around CNNs on FPGAs.
- In Chapter 4, we present the CNN developed by our collaborators in FORTH for the classification of Euclid data and its structure, as well as the necessary modeling performed to understand the specific parameters of the network to port it to our FPGA platform.
- In Chapter 5, we describe the architectures of the given CNN for our specific FPGA platforms in different stages, the limitations we came across in this task, and the means we used to solve them.
- In Chapter 6, we present and evaluate the performance of our implementations in the FPGA platforms compared to various benchmarks.
- At least, Chapter 7 is the conclusion of this thesis and presents suggestions for future improvements.

Chapter 2

Theoretical Background

2.1 Introduction to Machine Learning

The term Machine Learning was first introduced to Computer Science by Arthur L. Samuel in 1959 and described as "the field of study that gives computers the ability to learn without being explicitly programmed" [4]. Although this is more of an abstract definition, we can easily understand what Machine Learning offers to science, a way closer to answering Alan Turing's initial question in his paper Computing Machinery and Intelligence "Can machines think?".

In 1997, Tom Mitchel proposed a more mathematical definition for Machine Learning: "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E." With the use of Machine Learning algorithms, computers are able to train on data inputs and statistical analysis to output results in a given range.

Because Machine Learning appears in several guises, with many applications and data types, much of the effort is made to reduce a range of diverse problems to a set of definite models and provide good guarantees for their solution. For this reason, Machine Learning is correlated with other fields of study focused on prediction, such as computational statistics.

Machine learning has improved every technology user's experience nowadays. Facial recognition technology helps users tag and share videos and photos of friends on social media. Machine learning-based recommendation engines suggest what books to read or movies to watch next based on user preferences. Web page ranking provides the user with sites relevant to his

search based on a vast number of criteria, all with the use of Machine Learning.

2.2 Deep learning

Deep learning is a subset of machine learning utilizing algorithms inspired by the structure and function of the brain called artificial neural networks. In other words, its goal is to mirror the functioning of our brains. Deep learning algorithms are similar to how each neuron is connected to each other, how information is passed, and how the nervous system is structured.

Deep learning utilizes a hierarchical level of Artificial Neural Networks (ANN) [5] to carry out the process of machine learning. While most traditional programs go over data linearly, the hierarchical function of deep learning systems enables computers to process data with a nonlinear approach.

In Deep Learning algorithms, each level (layer) learns to transform its input data into a slightly more composite representation with the purpose of minimizing the difference between its prediction and the expected output. For example, in an image recognition application that classifies cats and dogs, the input is an image represented as a matrix of pixels; the first representational layer may abstract the pixels and encode edges, the second layer may encode a nose, eyes, and sharp teeth, and the third layer may recognize that the image contains an animal. If the input is a dog, but the deep learning algorithm predicts a cat (inference), the algorithm will learn through training that the features of the given input are associated with a dog label.

2.3 Convolutional Neural Networks

In deep learning, a convolutional neural network (CNN) is a class of deep neural networks with applications in the fields of image recognition and classification, natural language processing, and recommendation systems.

Convolutional neural networks and their architecture are similar to that of the connectivity pattern of the neurons in the human brain and were inspired by the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The fields of different neurons partially overlap to cover the entire visual area collectively.

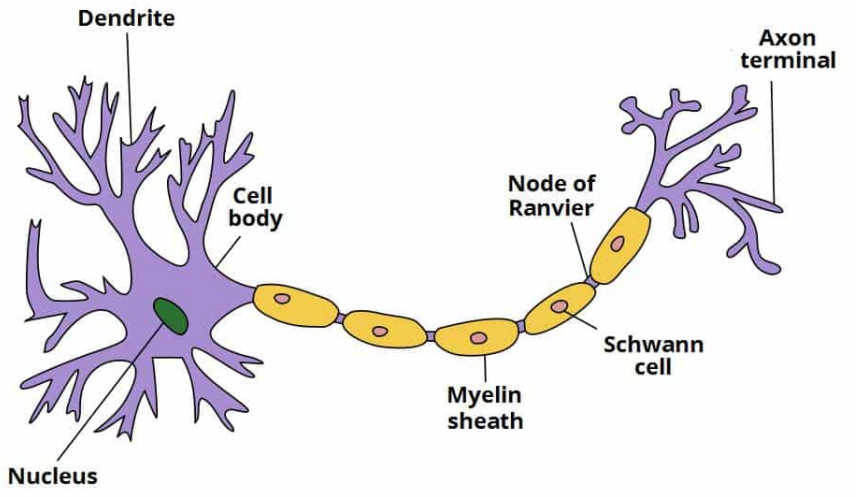

FIGURE 2.1: Basic Structure of a Neuron https://teachmeanatomy.info/the-basics/ultrastructure/nerves/

The pre-processing required in a CNN is much lower compared to other classification algorithms. With enough training, this means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a significant advantage and a big difference from the majority of Machine Learning algorithms.

Each neuron in a neural network computes an output value by applying some function to the input values coming from the receptive field in the previous layer. The function that is used for the input values is specified by a vector of weights and a bias. Learning in a neural network progresses by making incremental adjustments to the biases and weights. The vector of weights and the bias are called a filter and represent some feature of the input. A distinguishing feature of CNNs is the possibility that many neurons share the same filter. This reduces memory footprint because a single bias and a single vector of weights are used across all receptive fields sharing that filter, rather than each receptive field having its own bias and vector of weights.

2.3.1 Convolution Layer

In mathematics, Convolution is an operation in which two sources of information, functions, are intertwined. The result is a third function that indicates how the shape of one of the two inputs is modified by the other.

The first layer of a Convolutional Neural Network is always a Convolutional Layer. In Convolutional Layers, the operation of Convolution is applied to the input and the results are passed to the next layer.

To configure a Convolutional Layer, we can use the following parameters:

- **Kernels**: Size of kernels (or filters) used in the Convolutional Layer.
- **Number of Filters**
- **Stride**: The number of pixels shifts over the input matrix in the convolution operation. For example, if a CNN's stride is set to 1, the filter will move one pixel - or unit - at a time.
- **Padding**: In cases where it is needed to save the information presented in the input's corners or edges, the usage of padding helps by adding extra rows and columns on the outer dimension of the input.

The number of Convolutional Layers in a CNN is directly correlated to the complexity of the problem the network is trying to solve. In tasks where features are relatively simple and distinguishable, a CNN with fewer hidden layers may suffice, and in more complex tasks, a deeper network is required.

2.3.2 Fully Connected Layer

Fully Connected Layers usually are the last or the last few layers in the network.

The input of the Fully Connected Layer is the output from the last Convolutional Layer, which is first flattened and fed into the Fully Connected Layer, where weights will be applied to perform a classification using the transformed features. Usually, the number of neurons in the final Fully Connected layer of a CNN equals the number of output classes in a classification task.

Adding a Fully Connected layer is a (usually) cheap way of learning non-linear combinations from the output of the Convolutional Layer, which represents high-level features. The Fully Connected layer is learning a possibly non-linear function in that space.

2.3.3 Pooling

A limitation of the Convolutional layers' feature map output is that they record the precise position of features in the input. Small changes in the input

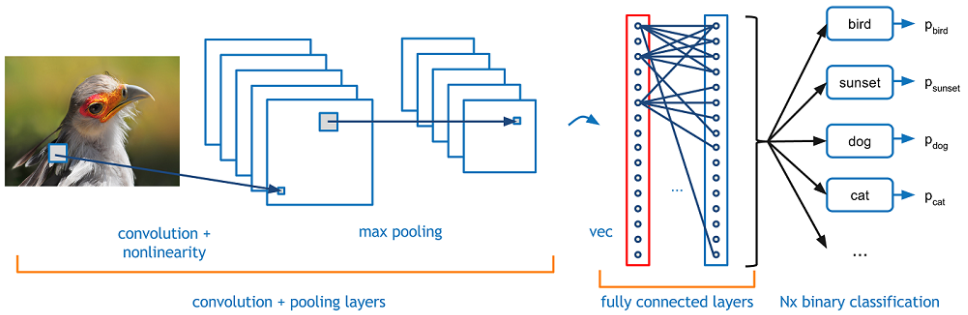

FIGURE 2.2: Structure of a CNN: https://towardsdatascience.com/covolutional-neural-network-cb0883dd6529

image will result in a different output. This can be the result of re-cropping, rotation, shifting, and other minor changes to the input.

A common approach to addressing this issue is called down sampling. A lower resolution version of the input is created, containing the essential features, reducing the spatial dimensions, without keeping the information not beneficial to the task. The down sampling can be achieved by a robust and common approach, adding a pooling layer.

There are two widely used types of pooling: average pooling and max pooling, with the latter being the most common.

- **Max Pooling**: The highest number of the input's area (an n × m matrice) is taken.
- **Average Pooling**: The mean of the input's pooling area is used

FIGURE 2.3: Pooling

2.3.4 Activation Function

The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it.

Non-linearity is not introduced into a network without the use of an activation function. We can use an activation function to model a response variable that varies nonlinearly in relation to its explanatory variables (target variable, class label, or score). The term ”non-linear” refers to an output that cannot be recreated by a linear combination of inputs.

There are different types of activation functions, such as Sigmoid, tanh, ReLU, Soft Max, Leaky ReLU, and more. In this work we will focus on ReLU and Soft Max.

Rectified Linear Unit (ReLU)

The Rectified Linear Unit (ReLU) is a widely used activation function in Convolutional Neural Networks. It is a simple function that outputs the input value if it is positive and zero if it is negative.

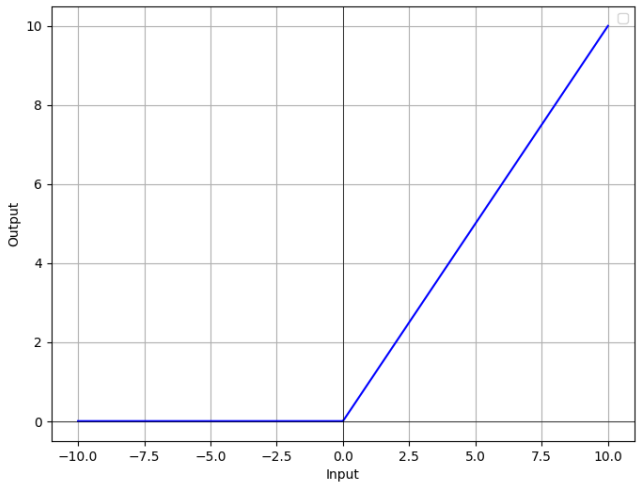

FIGURE 2.4: ReLU

The main drawback of ReLU is that all negative values are turned to zero, which can decrease the ability of the CNN to fit or train properly. This issue is resolved using Leaky ReLU, which allows negative values with a gradient or other activation functions.

Soft Max

SoftMax is usually used in Convolutional Neural Networks to convert the network’s final output into probability distributions with a sum of 1, ensuring that the output values represent normalized class probabilities.

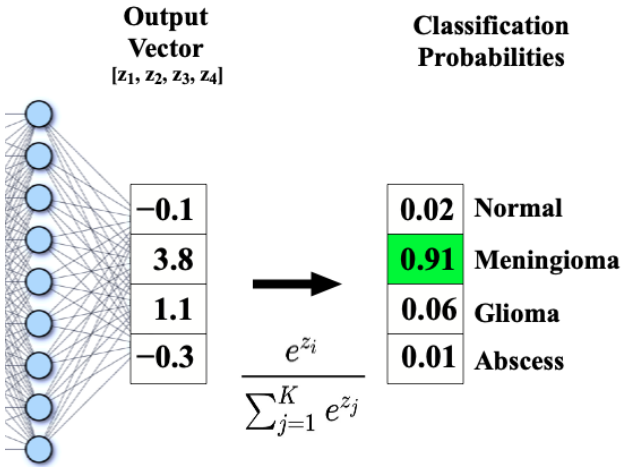

FIGURE 2.5: SoftMax: https://mriquestions.com/softmax.html

2.3.5 Inference

During inference, a CNN takes an input image and passes it through convolutional, pooling, and fully connected layers to produce a final output, such as a classification prediction. The convolutional layers apply a set of filters (or kernels) to the input, producing feature maps that capture local spatial patterns. The pooling layers then reduce the spatial dimensions of the feature maps. Finally, the fully connected layers combine the extracted features to produce the output.

The inference process in a CNN is computationally intensive, requiring millions of multiply-accumulate operations to apply the convolutional filters and propagate the feature maps through the network.

The complexity and depth of the CNN architecture directly impact the inference time, with deeper and more complex models generally taking longer to run the inference and requiring better hardware. Careful optimization of the CNN architecture and implementation is deemed necessary in order to run inference in real-time or in a satisfactory level, specifically when there are hardware constraints, as in our case.

2.3.6 Training

Training in CNNs is the process of optimizing the network's weights and biases to achieve the primary objective, minimizing the loss function. This is typically done using a supervised learning approach, where the network is given a set of labeled training images. The network learns to map the input

images to their correct labels by adjusting the weights and biases to minimize the difference between predicted and actual labels.

A loss function is used to measure how well the network is performing on the training data. The loss function is typically calculated by accounting the difference between the predicted labels and the actual labels of the training data.

2.3.7 Back Propagation

Backpropagation [6] is a technique for calculating the gradients of the loss function with respect to the network's weights and biases in order to update them. The gradients indicate the direction and magnitude of adjustments needed for each parameter to minimize the loss function.

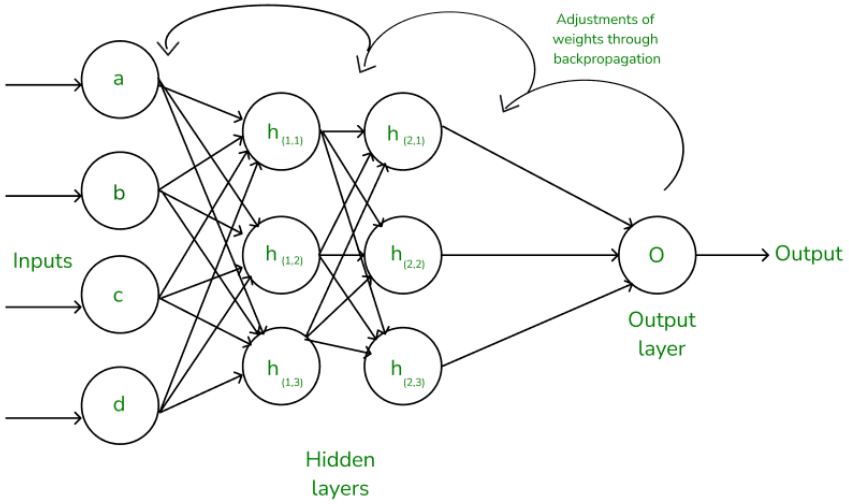

FIGURE 2.6: Backpropagation: https://www.geeksforgeeks.org/backpropagation-in-neural-network/

2.3.8 Classification

As mentioned earlier, the final classification step usually incorporates Soft-Max to exploit its probabilistic characteristics. As we can calculate the probability distribution, we can make our final classification by selecting the class with the highest probability.

Chapter 3

Related Work

3.1 Frameworks for Deep Learning

In recent years, as the interest in developing deep learning applications has increased, several software frameworks have been developed. These frameworks provide the necessary tools to design, train, and deploy CNNs. In this section, we will examine some of those prominent frameworks.

3.1.1 TensorFlow

TensorFlow [7][8] is an open-source deep-learning framework developed by Google and initially released in November 2015, with version 2.0 released in September 2019. TensorFlow is based on Google's earlier deep learning system DistBelief [9], but simplified and generalized to target a wider variety of tasks and is considered one of the most popular and widely used deep learning frameworks with support for multiple platforms (CPU, GPU, TPU) as well as large-scale distributed systems with hundreds machines and mobile platforms [10].

3.1.2 Keras

Keras [11][12], introduced in March 2015, is a library that provides abstract building blocks to build deep learning networks and supports both CPU and GPU. The building blocks are built using Theano and TensorFlow. It allows the user to focus on the main concepts of deep learning, such as creating layers for neural networks, while taking care of the details of tensors, their shapes, and their mathematical details [13].

Keras is divided into two major kinds of framework: the sequential model [14] and the functional API [15]. The sequential API is considered the simplest of the two and is a linear stack of layers, with each layer having precisely one input and one output tensor. The main idea behind functional API is that a deep learning model is usually a directed acyclic graph (DAG) of layers. So, the functional API provides the means to create more complex and flexible models, like models with non-linear topology, shared layers, and multiple inputs or outputs.

3.1.3 Caffe

Caffe (Convolutional Architecture for Fast Feature Embedding) [16] is a deep learning framework originally developed at the University of California, Berkeley, released in April 2017. It is written in C++ and provides bindings to Python and MATLAB. Although Caffe was originally created for vision, it was quickly adopted and improved for various tasks. Caffe is deprecated as Caffe2 was merged into PyTorch in March 2018.

3.1.4 PyTorch

PyTorch [17][18] is an open-source deep learning framework developed by Meta and released in September 2016. Its major advantage is that it uses dynamic computation graphs, also known as define-by-run graphs, allowing users to modify the network behavior on the fly. This provides scalability to different dimensional inputs, flexibility, ease of debugging, and faster prototyping.

3.2 Hardware for CNN

In this section, we will explore the various hardware platforms that have emerged to implement CNN architectures, from general-purpose processors to specialized accelerators designed for inference workloads. CPUs lack the computation power required to run modern CNNs, especially compared to other platforms, so we will not examine them further in this chapter.

3.2.1 GPU

GPUs have traditionally been used to accelerate 3D graphics applications, like games. In recent years, they have also been used for machine learning [19], as the computations in CNNs are inherently parallel and mainly floating-point operations, which makes them well-suited for the GPU's computing model. Critical factors behind the capabilities of the GPUs are the CUDA and Tensor Cores.

CUDA (Compute Unified Device Architecture) Cores, introduced in 2006, are the building blocks of NVIDIA GPUs. They are designed to perform general-purpose floating-point computations in parallel. Tensor Cores, introduced in 2017 with NVIDIA Volta architecture, presented specialized hardware to optimize AI-related workloads. Better utilization of a GPU's available resources, achieved by offloading part of the operations from Tensor cores to CUDA cores, can lead up to a 19% improvement in performance [20].

Furthermore, NVIDIA provides cuDNN (CUDA Deep Neural Network) [21] and CUDA Toolkit [22]. cuDNN is a GPU-accelerated library that accelerates widely used frameworks, like TensorFlow and Keras, by delivering optimized implementations for standard routines such as forward and backward convolution. CUDA Toolkit offers a development environment for creating GPU-accelerated applications to develop, optimize, and deploy applications on GPU systems. CUDA Toolkit and cuDNN are complementary technologies that work together as many deep learning frameworks require both to be installed.

As the research of parallel implementations of CNN on GPUs has flourished, there have been studies regarding the evaluation of the said implementations [23], providing detailed performance analysis and exploring bottlenecks, assisting researchers to find the more suitable implementation for their needs.

3.2.2 FPGA

As a trade-off between performance, energy efficiency, and flexibility, FPGAs offer an interesting design point between GPUs and ASICs and have shown great success in accelerating inference tasks, even while usually running in lower clock frequencies compared to both GPUs and ASICs.

Microsoft has already deployed FPGAs in datacenters both to accelerate generic workloads [24], like using the FPGAs to rank documents for the Microsoft

Bing Search and to accelerate Deep Convolutional Neural Networks [25]. The results in both cases are auspicious, as these architectures can achieve high levels of performance at low power consumption. Furthermore, the GPU solutions require up to 10x power to operate compared to the FPGA solution, making their deployment impractical in power-constrained cases, such as datacenters.

Numerous studies have compared the performance and energy efficiency differences between FPGAs and other platforms for various tasks. In [26], the team compared the two platforms in the case of Sliding-Window applications, like convolution, and the FPGA provided better performance, except for small input sizes, with speedups up to 11x compared to the GPU, while even being in some cases orders of magnitude better in terms of energy efficiency. In [27], the focus is shifted to the trends in next-generation DNNs, like exploiting sparsity irregular parallelism and custom data types. The team proposed a customizable DNN hardware accelerator template for FPGA and presented results up to 5.4x better performance than GPUs in binarized DNNs.

FPGAs have less NRE cost and can be reconfigured to support newer models and different layer types, making it easier to keep up with the continuous advances in the field, especially when compared to ASIC implementations. Also, as FPGAs often offer a variety of I/O, they provide the opportunity to be part of larger systems.

3.2.3 ASIC

In recent years, there have been several applications with limited power budgets and tight latency constraints, like mobile embedded platforms or self-driving cars, that are not suited for the inference to be hosted on GPUs. To achieve the best performance and energy efficiency, the focus has been shifted to building custom application-specific integrated circuits (ASICs) to accelerate CNN inference workloads.

In [28], the team designed a Small-Footprint High-Throughput accelerator for large-scale CNNs and DNNs with a small footprint that, compared to a 128-bit 2GHz SIMD processor, is 117.87x faster and can reduce the total energy by 21.08x. This was achieved by focusing on state-of-the-art networks with large-scale layers, which allowed for the introduction of storage structures designed to take advantage of locality properties. In [29], the team

is focused on accelerating different types of DNNs using a multichip architecture and proved that it is possible to achieve a significant speedup over a GPU while reducing the energy for a subset of the largest known neural network layers. In [30], the team presented EIE, an energy-efficient engine optimized for inference on compressed deep neural networks to make the Fully Connected layers more efficient, achieving a 3.400x and 13x in energy consumption and performance compared to GPU, respectively.

Furthermore, the case of ASIC vs FPGA for CNN inference task has been made [31] where ASIC and FPGA platforms are compared on various Architectures for accelerating inference. The ASIC's performance is reported to be 2.8× to 6.3× faster than that of the FPGA implementations.

ASICs do not offer enough flexibility to accommodate the rapid evolution of deep learning models, as the high non-recurring engineering (NRE) cost and time for the design, verification, and implementation of an ASIC make it challenging to keep up with the field's growth.

3.2.4 TPU

In 2016, Google announced the Tensor Processing Unit (TPU), its own AI accelerator application-specific integrated circuit (ASIC) built specifically for machine learning. TPU is optimized for Google's TensorFlow 3.1.1. TPU is designed to perform the large matrix operations commonly found in machine learning algorithms much more efficiently than general-purpose CPUs or GPUs. Specifically, Google, after a year of using the TPU in their data centers, claims that the TPU delivered 15–30X higher performance and 30–80X higher performance-per-watt using the TOPS/Watt metric than contemporary CPUs and GPUs [32].

The TPU is built on a 28nm process, runs at 700MHz, and consumes 40W when running [33] while connecting to its host via a PCIe Gen3 x16 bus that provides 12.5GB/s of effective bandwidth. It includes 65,536 Matrix Multiplier Units(MXU), an 8-bit multiply-and-add unit for matrix operations, featuring a systolic array, a different architecture than typical CPUs and GPUs. This means that the inputs that need to be used many times as part of producing the output are read only once but used for many different operations without being stored back in a register.

In May 2024, Google announced the sixth generation of TPU, claiming a 4.7X increase in peak compute performance per chip compared to the previous generation while being 67% more energy-efficient [34].

3.3 The FPGA perspective

Although FPGAs have BRAM integrated into the fabric, the limited size is a major bottleneck. Newer FPGAs have big DRAM modules, but compared to GPUs, they still face limitations. Reducing data precision can help get around those restrictions. Several studies have shown that CNN inference does not require floating-point computations and fixed-point arithmetic can be used with less than a 1% drop in accuracy [35] and satisfactory results [36]. In [37], they used fixed-point 16-bit, proposed an architecture based on the Winograd algorithm, and designed a line buffer to cache the feature maps for the Winograd algorithm. The Winograd algorithm improved arithmetic complexity by reducing the required number of multiplications as a tile of elements in the feature map is generated together using the structural similarity among them. This produced a 2.98x energy efficiency compared to GPUs.

Furthermore, as the Binarized Neural Networks (BNN) [38][39] are improving in terms of accuracy [40], the FPGAs can take advantage of the much lower memory footprint of those Networks and be solid platform alternatives. In [41], a roofline model to quantify peak performance for BNNs on FPGAs was developed, showing that the FPGA's performance for binary operations is about 16× higher compared to 8-bit and 53× higher compared to 16-bit fixed point operation. As the focused BNN (binarized AlexNet) could be kept in the on-chip memory, it was almost able to reach the computational peak, while the peak performance of the fixed-point CNN was bound by the memory bandwidth. The FPGA accelerator presented in [42] is 8.3x faster and 75x more energy-efficient than the GPU counterpart when processing online individual requests in small batch sizes and on par with the GPU in terms of throughput while delivering 9.5x higher energy efficiency in the case of static data in large batch sizes. This was made possible by the hardware mapping of convolution kernels using LUTs for bitwise operations, with LUT utilization of 79%, which led to massive computing parallelism.

This wide variety of precisions matches well with FPGAs as they are able

to execute non-standard custom bit-width arithmetic with much higher efficiency and flexibility than GPUs.

In [43], the team developed a deeply pipelined multi-FPGA architecture, which achieved up to 21x and 2x energy efficiency compared to CPU and GPU implementations, respectively. As the different kernels in a CNN model require various compute resources, for example, in the case of AlexNet, Convolution layers are compute-intensive while corresponding only to the 4.31% of the total weights, and the Fully Connect layers are memory bandwidth intensive while containing only 6% of the total arithmetic operations, it is challenging to balance the resource allocation on a single FPGA for different layers in the CNN. Each of the six FPGAs has a computation engine customized for one or more CNN layers with the interconnection between the FPGAs implemented using the Xilinx Aurora protocol [44]. Each link is bidirectional, has a bandwidth of 750MB/s, and is responsible for transferring the output feature maps of the last layer of the transceiver FPGA to the receiver FPGA.

In [45], the team first analyzed the CNN and its characteristics, highlighting, like in [43], the differences between compute-intensive Convolution and memory-centric Fully Connected Layers. This distinction is crucial for designing efficient hardware accelerators. The study introduces a method for dynamic-precision quantization, which allows for reduced precision in data representation, reducing memory footprint and bandwidth requirements without significantly sacrificing accuracy. They also introduce the memory system design, which aims to feed the elements with data efficiently by arranging the data for both the Convolutional and Fully Connected layers to maximize the burst length of each DMA transaction. The final architecture achieved an average performance of 187.80 GOP/s for Convolution Layers and 136.97 GOP/s for the whole network. Although the system is 1.4× faster than the CPU and consumes less power, the performance of the GPU is 13.0× better, with the drawback of 26.0× power consumption.

A detailed quantitative analysis of the design objectives, such as latency and data transfers, of a CNN accelerator and their correlation to the design variables, e.g., loop unrolling, is presented in [46], which resulted in a specific dataflow of hardware CNN acceleration. The dataflow aims to minimize data transfers and memory access while fully utilizing the available computing resources for high performance. The proposed accelerator achieved up to 5.6× throughput and 5.5× latency improvements compared to prior implementations.

Chapter 4

The CNN for Euclid Data Classification

In this Chapter, we will present the Convolutional Neural Network implemented in this thesis, as well as its background, the motivation behind its creation, and the problem that it solves. We will also describe the specific CNN's structure and characteristics.

4.1 Convolutional Neural Networks for Spectroscopic Redshift Estimation on Euclid

As the recent advances in the emerging fields of Big Data and Deep Learning have made the analysis of huge amounts of data from various sources easier, the field of astronomy is one of the fields that stand to benefit from this as satellites and telescopes have been capturing large numbers of images of the sky.

One particular problem in astrophysics is the ability to derive precise estimates of galaxy redshifts. Redshifting is the increase in the wavelengths and the corresponding decrease in the frequency and photon energy of light emitted from galaxies through the Doppler effect. This is the result of galaxies moving away from each other and any given observation point due to the expansion of the Universe according to the Big Bang model. Redshift is the principal way in which galaxies' radial distances can be measured and hence their 3-dimensional position in the Universe, helping to observe the rate of expansion of the Universe and the gravitational lensing of light by the matter distribution.

The Euclid satellite aims to measure the global properties of the Universe and will collect photometric data and spectroscopic data [47]. The latter will help us determine the details of cosmic acceleration through measurements of the distribution of matter in cosmic structures. However, redshift estimation from spectroscopic observations is not a simple task as there are various sources of errors, both instrumental and astrophysical, that are usually taken care of through human evaluation. This approach is possible only with small datasets, but as Euclid will obtain a massive amount of spectra, new automated processes need to be developed that will achieve high accuracy with minimum human evaluations.

Estimation of galaxy redshifts is usually perceived as a regression procedure because a galaxy redshift can be measured as a non-negative real value number, but it can be transformed into a classification task without the loss of essential information. Due to the characteristics of Euclid, the study will be focused on the redshift range of detectable galaxies, and it is possible to restrict the precision of each estimation to match the resolution of the spectroscopic instrument. Therefore, the chosen redshift range can be divided into evenly sized slots equal to Euclid's required resolution. As a result, the task is transformed into a classification task which uses a set of ordinal classes, each one corresponding to a different slot.

As a classification model can be utilized, a Convolutional Neural Network was designed by Dr. Gregory Tsagatakis and his team to translate the spectroscopic redshift data collected by the Euclid satellite into a reliable classification of galaxies [48].

For the implementation and training of the model, TensorFlow 3.1.1 and Keras 3.1.2 were used, and a simulated dataset was modeled after the Euclid satellite galaxy survey [49], in terms of quality, veracity, and volume.

The team trained and evaluated multiple Convolutional Neural Networks with different depths, including 1, 2, and 3 Convolutional Layers, in order to examine the impact of the different depths on the network. In all cases, the final Fully Connected layer was used for the classification. The shallowest, with only one Convolutional Layer, was the slowest in terms of epochs needed to converge and significantly underperformed compared to the deeper models.

Furthermore, the team investigated how the size of the training set affects the network's behavior, again starting with networks of various depths. The

size of the training sets scales from 40.000 to 100.000 and, finally, to 200.000 and 400.000 observations. As easily perceived, using the training set with the smallest number of observations introduces overfitting, with the network performing well in the training set but not well enough in the validation and testing examples. Also, the bigger the training set is, the better the model will be able to generalize in the long term, and overfitting will decrease.

4.2 Our Network

4.2.1 Network Architecture

The final model designed is able to classify images created from Euclid data into 800 Classes and was trained using 400.000 examples. Each image, input of the network, has a total of 1800 pixels, comprising the features of the model. Three Convolution Layers make up the network, followed by the classifier, a Fully Connected Layer, and a SoftMax stage. After each Convolution Layer, there is a non-linear activation function (ReLU).

In Figure 4.1, we present a simple 1-dimensional Convolutional Neural Network, which takes a one-dimensional array X with size p. The input is convolved with the kernel W of size 3 with stride 1, resulting in the array C of size $g = p - 3 + 1$, with 3 being the kernel size. A non-linear function is applied to C, resulting in C', which is the input of the Fully Connected layer.

1st Convolution Layer

The first Convolution Layer of the network gets as input the image, a one-dimensional vector of size 1800x1 32-bit floating-point values. Furthermore, for the convolution, we need the pre-trained kernel matrix and the bias. The total input of the first layer is 1944 32-bit floating-point values, 1800 pixels from the image, and 144 values from the kernel and bias. The output of this layer is a two-dimensional array.

In algorithm 1, we present a MATLAB-style pseudo-code for the first layer of our CNN. The inputs are:

- image, a one-dimensional array with a size of 1x1800
- kernel, a two-dimensional array with a size of 16x8
- bias, an array of size 16

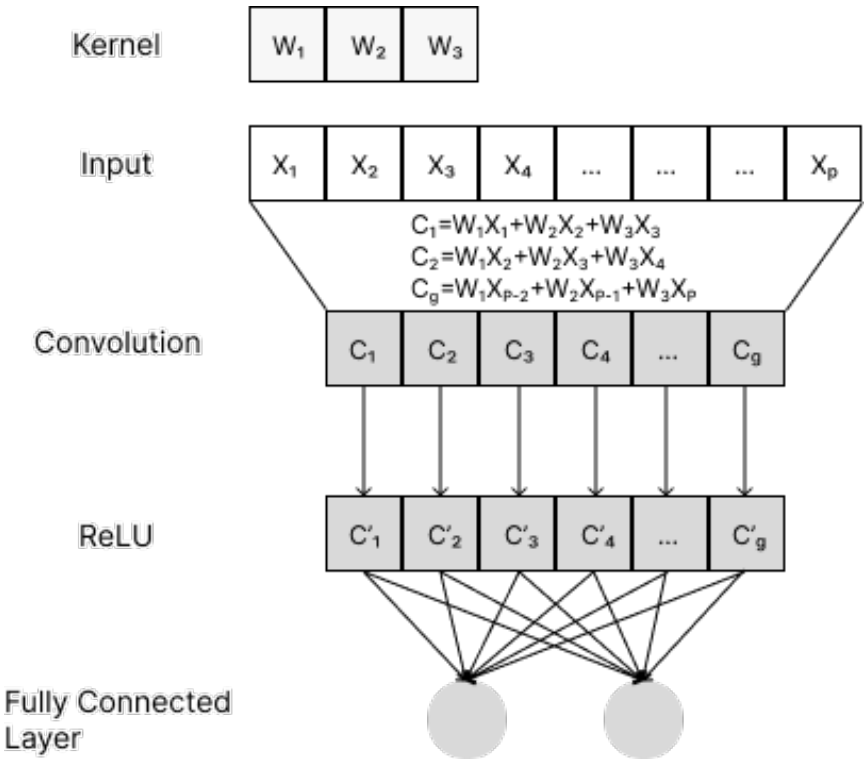

FIGURE 4.1: A simple 1-Dimensional CNN: The input X is convolved with a pre-trained filter W, with stride one. A non-linear activation function (ReLU in our case) is applied to the output. Finally, there is the Fully Connected layer of the classification, with output size equal to the number of classes (800 in our case)

Algorithm 1 1-D Convolution

procedure CONVOLUTION 1 D($image, kernel, bias$)
  $ImageSize \leftarrow size(image)$
  $KernelDim1 \leftarrow size(kernel, 1)$
  $KernelDim2 \leftarrow size(kernel, 2)$
  $ConvOutput \leftarrow zeros(ImageSize - KernelDim2 + 1, KernelDim1)$
  for $i \leftarrow 1$ to $KernelDim1$ **do**
    $ConvOutput(:, i) \leftarrow conv(image, kernel(i, :), valid)$
  for $i \leftarrow 1$ to $KernelDim1$ **do**
    $ConvOutput(:, i) \leftarrow ConvOutput(:, i) + bias(i)$
  return $ConvOutput$

The output is a two-dimensional array of size 1793x16. As this is based on MATLAB, the : in matrices refers to entire rows or columns; for example, $ConvOutput(:, i) \leftarrow conv(image, kernel(i, :), valid)$ means: For i, perform a 1D convolution between the input and the $i - th$ row of the kernel ($kernel(i, :)$) and store the result in the corresponding column i of the output matrix. The *valid* value returns only those parts of the convolution that the function

computes without zero-padded edges.

2nd Convolution Layer

The second Convolution Layer of the network gets as input the two-dimensional output of the first Convolution Layer. Furthermore, for the convolution, we need the pre-trained kernel matrix and the bias values, summing to 2064 32-bit floating-point values. The output of this layer is a two-dimensional array.

Algorithm 2 2-D Convolution

```
1:  procedure CONVOLUTION 2 D(input, kernel, bias)
2:      InputSize ← size(input)
3:      KernelDim1 ← size(kernel, 1)
4:      KernelDim2 ← size(kernel, 2)
5:      KernelDim3 ← size(kernel, 3)
6:      ConvOutput ← zeros(InputSize − KernelDim3 + 1, KernelDim1)
7:      for j ← 1 to KernelDim1 do
8:          for i ← 1 to KernelDim2 do
9:              ConvOutputTemp(:, i) ← conv(input(:, i), kernel(:, j, i), valid)
10:         ConvOutput(:, j) ← sum(ConvOutputTemp, 2)
11:     for i ← 1 to KernelDim1 do
12:         ConvOutput(:, i) ← ConvOutput(:, i) + bias(i)
13:     return ConvOutput
```

In algorithm 2, we present a MATLAB-style pseudo-code for the second and third layers of our CNN. The inputs are:

- input, a two-dimensional array, the output of the previews layer (1793x16 and 1786x16)
- kernel, a three-dimensional array with a size of 16x16x8
- bias, an array of size 16

While the output is a two-dimensional array of size 1786x16 for the second layer and 1779x16 for the third.

3nd Convolution Layer

The third Convolution Layer of the CNN is the same as the second one. The inputs are:

- the output of the previous layer (two-dimensional array)
- the pre-trained kernel of size 16x16x8 (2048 32-bit floating point values)

- the bias with size 16 32-bit floating point values

The output is also the same as the second layer, a two-dimensional array.

Fully Connected Layer

The last layer of the network is the Fully Connected layer. It takes as input the output of the last convolution layer after it has been flattened, effectively transformed from a two-dimensional vector to a one-dimensional one. The layer consists of 800 neurons, each responsible for the result of one of the output classes. It consumes a total of 22771200 weights, 28464 for each class, to calculate the final output, a one-dimensional array of size equal to the number of Classes, in our case 800.

Algorithm 3 Fully Connected

1: **procedure** FULLY CONNECTED($input, kernel, bias$)
2: $InputSize \leftarrow size(input)$
3: $NumberOfClasses \leftarrow size(bias)$
4: $Output \leftarrow zeros(NumberOfClasses)$
5: $Output \leftarrow kernel * input$
6: **for** $i \leftarrow 1$ to $NumberOfClasses$ **do**
7: $Output(i) \leftarrow Output(i) + bias(i)$
8: **return** $Output$

In algorithm 3, we present a MATLAB-style pseudo-code for the Dense Layer of our network. The inputs are:

- input, the flattened output of the third Convolution Layer, a one-dimensional array of size 1x28464
- kernel, a two-dimensional array with a size of 28464x800
- bias, an array of size 800

The output is a one-dimensional array of size 800, one for each of the Classes

A SoftMax function 2.3.4 is applied to the output of the Fully Connected Network, transforming it into probability distribution with a sum of 1.

Activation function - ReLU

After each of the Convolution Layers, an activation function, ReLU 2.3.4 in our case, is applied to the layer's output, ensuring that values less than zero

do not pass to the next layer. In algorithm 4, we present the steps for the ReLU, which takes as input the two-dimensional vector, the output of the previous layer.

Algorithm 4 ReLU

```
1:  procedure RELU(input)
2:      InputDim1 ← size(input, 1)
3:      InputDim2 ← size(input, 2)
4:      for i ← 1 to InputDim1 do
5:          for j ← 1 to InputDim2 do
6:              if input < 0 then
7:                  Output(j, i) ← 0
8:              else
9:                  Output(j, i) ← input(j, i)
10:     return Output
```

4.2.2 Memory Footprint

An important aspect of the performance of the network is the amount of data, mainly weights and bias, that need to be streamed in the design. Specifically, to implement the inference of the network in the FPGA, where memory is a scarce resource as BRAMs are small, it is essential to map the amount of data that is used, the distribution of the data in the different stages of the network, and the rate we can access that data. In the following table 4.1, we present the number of weights and data for the different layers that need to be accessed, along with their size and memory percentage.

Layer	#Weights	Footprint	Memory(%)
Convolution Layer 1	144	1.1 Kilobytes	6.33×10^{-4}
Convolution Layer 2	2064	16.1 Kilobytes	9.2×10^{-3}
Convolution Layer 3	2064	16.1 Kilobytes	9.2×10^{-3}
Fully Connected	22771200	173.7 Megabytes	99.98

TABLE 4.1: Weights and Memory Footprint of the Network

It is obvious, as reported in the related research, that the Dense Layer is memory bandwidth intensive as its weights are the significant majority of the total weights of the network, 99.98% of the total data. Furthermore, it is apparent that it is not possible to fit the data in the BRAM, as the data are in the hundreds of MB (173.7 MB), and there is not enough space to store them in the 4MB BRAM.

4.2.3 Weight Distribution

As mentioned before, it is important to explore, analyze, and understand the distribution of the model's weights for each layer. This is possible to provide valuable insights that can be leveraged for weight pruning, effectively reducing the memory footprint of the network without affecting performance. This also leads to reducing the network's complexity, which may result in faster inference and reduced computational resource requirements. Furthermore, analyzing weight distribution is a necessary step when considering quantization, converting weights to lower precision formats, which again leads to faster inference and reduced requirements for computational resources.

The following figures present the weight distribution of the 3 Convolutional Layers alongside the Fully Connected layer in the histogram charts.

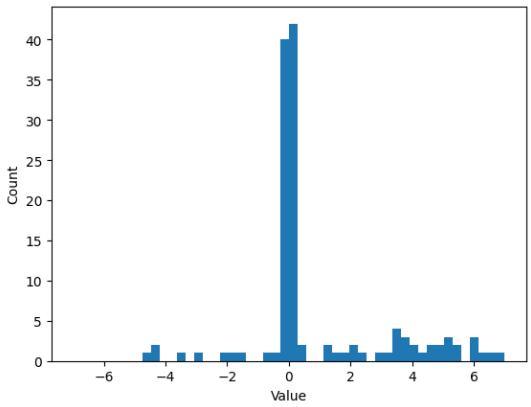

FIGURE 4.2: Weight Distribution - 1st Conv Layer

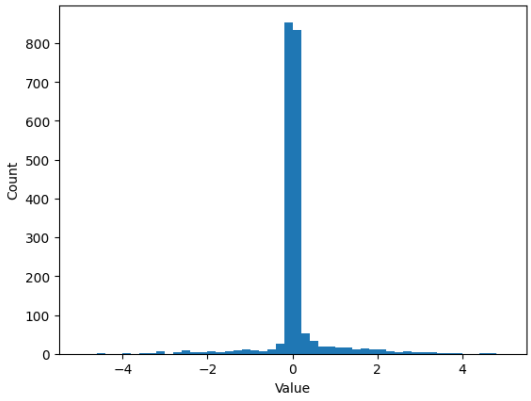

FIGURE 4.3: Weight Distribution - 2nd Conv Layer

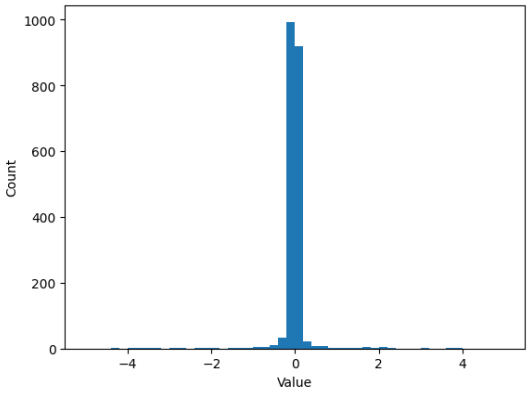

FIGURE 4.4: Weight Distribution - 3rd Conv Layer

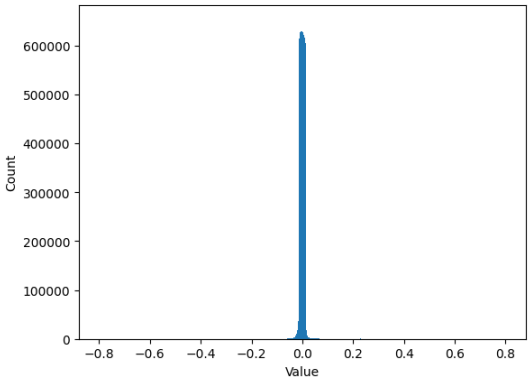

FIGURE 4.5: Weight Distribution - Dense Layer

4.2.4 Reduction of Memory Footprint

As mentioned before, the network's data can not be stored in the FPGA's BRAM, and memory constraints are a significant performance bottleneck of many applications. So, it is essential to reduce the overall memory footprint, either so the network's parameters can fit in the BRAM or to reduce the amount of data streamed to the FPGA. A more detailed description of this process can be found in the thesis of G. Pitsis [50].

One of the first options is to use Fixed Points, either static or dynamic, which will significantly improve the memory footprint. The main issue with this option is that as the bit-width reduces, the network's accuracy also decreases. So, a hybrid solution is proposed, where the weights are grouped according to their values in k centroids, and their new values are placed in a codebook. This means that there is no need to store the value of each weight, only the corresponding centroid's index, providing the opportunity for the kernels to be floating points with reduced memory footprint. The clustering can be achieved using several algorithms, such as Lloyds or K-means.

A drawback of Lloyds algorithm is that it tries to group weights without understanding the importance of different values. It can be easily perceived that larger weights are more important for the outcome of the network compared to smaller weights due to the mathematical operations, but their density is far smaller compared to that of smaller weights 4.2.3.

In order to cope with this, two new steps were proposed: Inverse Density and Hierarchical Clustering. We propose to initialize the codebook starting from the minimum value and ending up with the maximum as we try to provide

a higher resolution to the larger absolute values and a lower to the smaller values. Furthermore, using a larger number of centroids will increase the resolution across all values, and the algorithm is forced to pay more attention to higher values. This will enable us to apply the clustering algorithm hierarchically, which will result in a better resolution for the important weights. Those steps are possible because the analysis of the weights shows that density is inversely proportional to the importance of weights.

The steps mentioned above were applied to the Fully Connected layer's weights as this is the main bottleneck of our system. Furthermore, it is important to note that, unlike [51][52], the model was not retrained.

Chapter 5

Design and Implementation of the FPGA Architecture

This Chapter focuses on the implementation of different architectures of our application. We propose FPGA-based hardware designs for our specific CNN with respect to the constraints presented by our platforms.

5.1 FPGA Platforms

5.1.1 ZCU 102

In hardware, most of the experiments were conducted on ZCU102 UltraScale+ evaluation [53] board, which consists the xczu9eg-2ffvb1156e MPSoC (multiprocessor system-on-chip). The evaluation board provides a rapid prototyping platform. It contains many useful features, including a power processor system (PS) hard block peripherals exposed through the Multi-use I/O (MIO) interface and a variety of FPGA user-programmable logic (PL).

LUT	274080
FlipFlop	548160
DSP	2520
BRAM	912

TABLE 5.1: Available Resources on ZCU102

5.1.2 QFDB

Quad-FPGA Daughter Board (QFDB) is a custom PCB platform designed and created by FORTH. Each board carries four Xilinx Zynq Ultrascale+ FPGA with xczu9eg-2ffvc900, with the computational power provided by the board

being 4x compared to ZCU102 and xczu9eg-2ffvb1156e. The FPGAs on the QFDB are connected between them with Xilinx Aurora Protocol.

Our architectures are fully compatible with both platforms because they both carry type Xilinx Zynq UltraScale+ (ZU9EG) FPGAs.

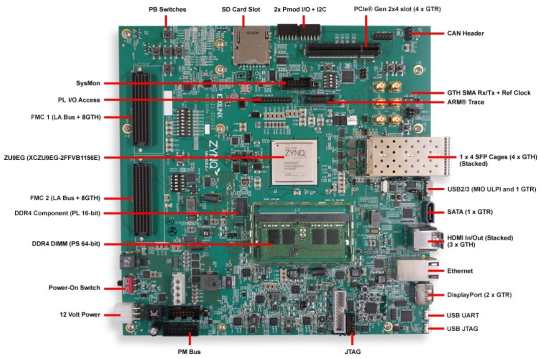

FIGURE 5.1: ZCU102 Evaluation Kit https://www.mouser.co.uk/new/xilinx/xilinx-zynq-ultrascale-zcu102-eval-kit/

FIGURE 5.2: QFDB

5.2 Xilinx Tools

For the hardware implementation in this thesis, the tools used were part of the Xilinx Vivado Design Suite, a software suite developed by Xilinx for synthesizing and analyzing HDL (Hardware Description Language) designs. The Xilinx Vivado Design Suite provides a robust set of tools that cater to different stages of the hardware development process from which Vivado HLS, Vivado IDE, and Xilinx SDK were used.

5.2.1 Vivado HLS

Vivado High-Level Synthesis (HLS) is a synthesis tool that enables developers to target C, C++, and SystemC programs directly to Xilinx devices, eliminating the need for manual RTL creation. By leveraging the capabilities of high-level programming languages, HLS automates the synthesis of programs into Intellectual Property (IP) blocks. These blocks are generated as VHDL or Verilog code and can be seamlessly integrated into hardware systems using other tools within the Xilinx Vivado Design Suite.

An HLS project with the necessary files must be created to create the IP block that will be integrated into our system. The tool offers an integrated debugger (C/RTL Co-simulation) to verify the RTL output prior to synthesis. After verifying that the code is working as expected and producing the correct results, we can proceed with the block's synthesis, which will generate the HDL code and a detailed report. The report contains useful information about the design's timing (frequency, latency) and resource utilization (estimated).

Furthermore, Vivado HLS offers pragmas that enable developers to optimize designs by reducing latency, enhancing throughput performance, and minimizing the area and resource usage of the generated RTL code. These pragmas are directly integrated into the kernel's source code.

array_partition: This pragma partitions an array into smaller arrays or individual elements. This results in RTL designs with multiple small memories or registers rather than a single large memory. This approach boosts the number of read and write ports for storage, potentially improving the throughput of the design. This pragma requires the use of additional memory instances or registers to accommodate the partitioned data structures.

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

pipeline: The pipeline pragma reduces the initiation interval for a function or loop by allowing the concurrent execution of operations.

stream: By default, array variables are implemented as RAM. Using the stream pragma replaces the RAM with FIFOs. This substitution enhances data communication efficiency, especially when data needs to be produced or consumed sequentially. FIFOs are particularly effective for managing continuous data streams.

unroll: The unroll pragma allows some or all loop iterations to occur in parallel by creating multiple copies of the loop body in the RTL design. The developer has the option to either fully or partially unroll the loop, allowing some decision-making in the trade-off between improving performance and increasing resource utilization.

5.2.2 Vivado

The Vivado IP integrator lets the developer create complex system designs by instantiating and interconnecting IP from the Vivado IP catalog or custom IP designed and synthesized using Vivado HLS on a design canvas. You can create designs interactively through the IP integrator canvas GUI or programmatically through a Tcl programming interface. Designs are typically constructed at the interface level but might also be manipulated at the port level.

The user must create a block diagram where the IPs will be placed. Every IP will be assigned a base address, clock, and reset signal. After that, the user must connect each IP's ports according to the design and validate that there are no errors. If there are errors, the user can use the Integrated Logic Analyzer (ILA) to record signal values to help him identify the location of the error.

The next step is to proceed with the synthesis and implementation of the design. Synthesis is the process of transforming an RTL-specified design into a gate-level representation. Implementation includes all steps necessary to place and route the netlist onto device resources within the design's logical, physical, and timing constraints, making the netlist ready for bitstream generation. Both synthesis and implementation provide the options to choose among several pre-configured strategies based on the needs of the design or create new ones for a more customized approach. A strategy is a set of switches to the tools, which are defined in a pre-configured set of options for the synthesis application or the various utilities and programs that run during implementation.

After synthesis and implementation are completed, comprehensive reports are generated to provide insights into various aspects of the design. These reports are crucial for evaluating the design's performance, power, resource utilization, and meeting timing constraints. Finally, the bitstream can be generated, provided no errors exist and the characteristics from the reports meet the requirements.

5.2.3 Vivado SDK

The Vivado Software Development Kit (SDK) is the Integrated Design Environment (IDE) for creating embedded applications targeting Xilinx ARM processors, and it is based on Eclipse 4.5.0 and CDT 8.8.0. Vivado SDK

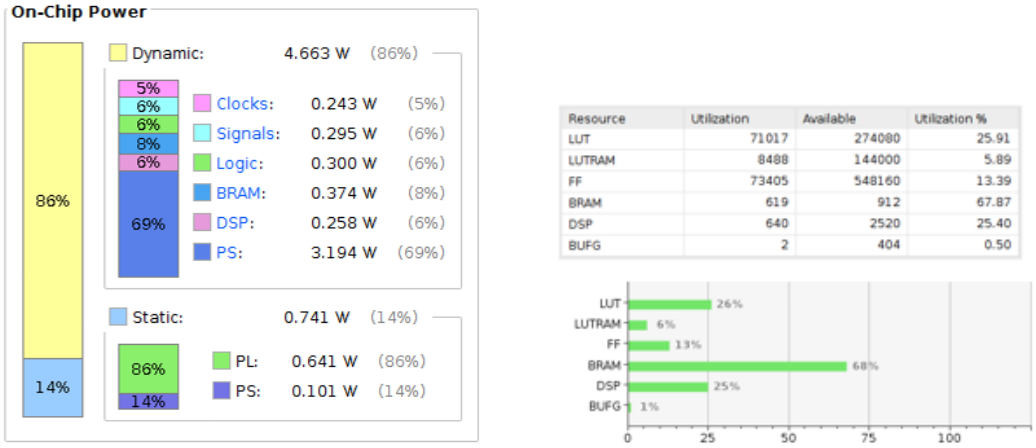

FIGURE 5.3: Vivado Power and Resources Utilization Reports

understands the custom embedded hardware design that has been defined in the Vivado IDE. Based on this design, several key parameters are auto-configured, including memory maps, peripheral register settings, tools and library paths, compiler options, JTAG and flash memory settings, debugger connections, and Linux and bare-metal Board Support Packages (BSPs). This means that as the implementation has been completed and the bitstream is generated, the necessary files will be imported upon opening the Vivado SDK.

Then, the developer is able to create the application's software program that will initialize and activate the IPs, DMAs, and other parts of the design based on their base address given in Vivado IPI. Furthermore, the data transactions between the different parts of the design must be defined during this procedure, as well as the data reading and writing from the SD card and time-measuring functions.

5.3 Architecture 1 - 1DMA

Our first novel approach to implementing a custom FPGA-specific architecture for our Convolution Neural Network was to break the CNN into smaller IP blocks, with the first being the three Convolution Layers and the second one the Fully Connected. This two-stage architecture was designed to be implemented on 2 FPGAs, and the output of the Convolution Layers stage will be communicated to the Fully Connected.

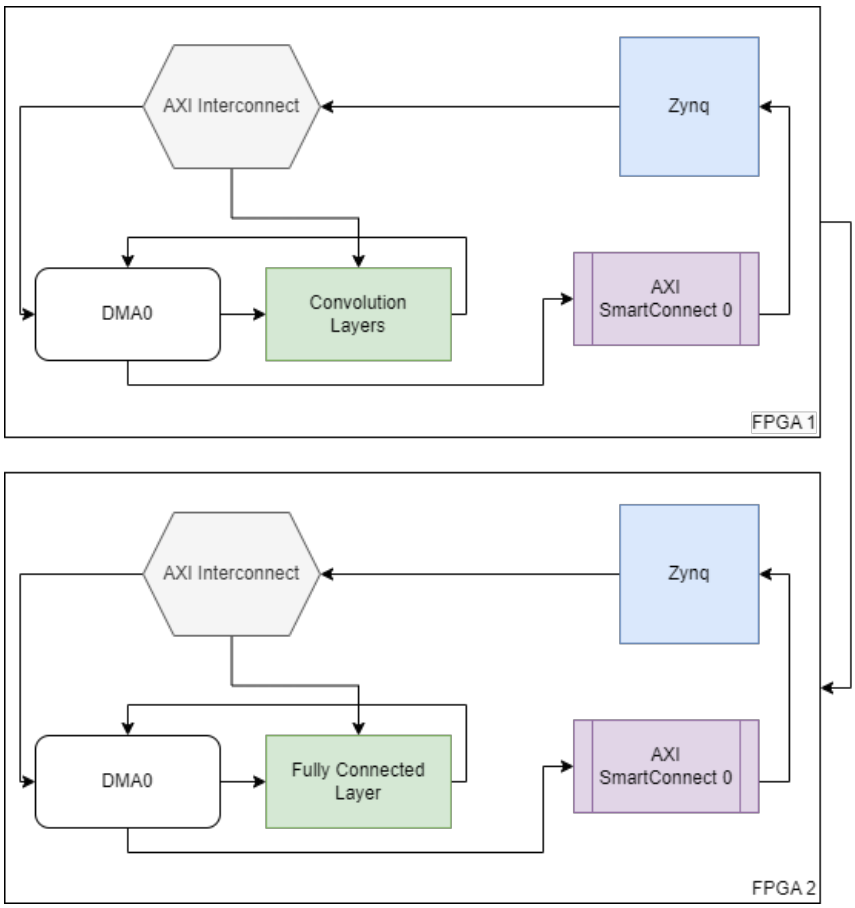

FIGURE 5.4: Two Stage Architecture

5.3.1 Convolutional Layers

As mentioned before, our CNN consists of three Convolutional Layers. The kernel of the first layer is significantly smaller than the second and third by a factor of 16. Specifically, the first convolution layer's kernel is 16x8 floats, with its input being 1800x1 and the output 1793x16, while the second and third are 16x16x8.

Each convolution layer is fully pipelined with interval 1, meaning that in each cycle, the layer consumes one input data and transmits one result per cycle after completing the first convolution operation. To achieve this, an interval array partition was used for the input and kernel filters. Furthermore, we made eight instances of the input of each convolution layer because the stride of the operation is one, and the size of the filter is 8x1, so we will be able to access consistent data from the input.

The input and output data transfers are implemented with AXI4 Stream using one DMA. Upon receiving the data, we store it in the binded BRAM and advance to making eight instances, as mentioned before.

We experimented with several clock periods for the first architecture, from 10ns to 3.3ns. Reducing the clock period further would not imply any improvements, as we would be bonded by the max frequency of the High-Performance Ports of the Zynq Ultrascale+.

Clock Period	100MHz	215MHz	300MHz
LUT usage	26%	25%	25%
FlipFlop usage	14%	17%	18%
DSP usage	26%	25%	25%
BRAM usage	68%	69%	70%

TABLE 5.2: Resources Utilization for the Convolution Layers

Clock Period	100MHz	215MHz	300MHz
Latency Cycles	596776	598181	600734
Latency	5.967ms	2.751ms	1.982ms

TABLE 5.3: Latency for different Clock Periods

Clock Period	100MHz	215MHz	300MHz
Total On-chip power	5.405W	8.428W	9.759W
Total On-chip Energy	0.032J	0.023J	0.0193J

TABLE 5.4: Power and Energy for the Convolution Layers

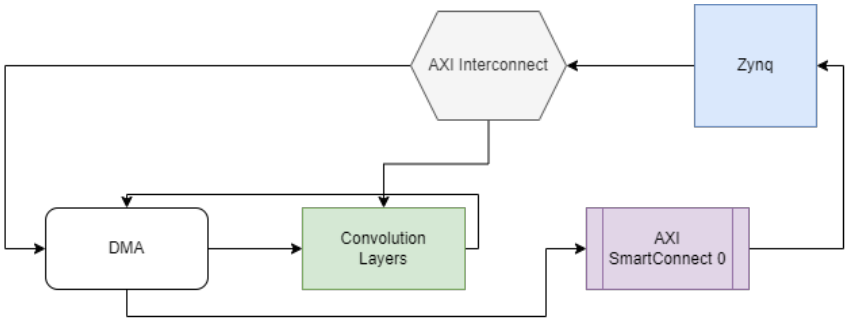

FIGURE 5.5: Architecture 1 Convolution Layers Design

5.3.2 Fully Connected Layer

After the three Convolution Layers, the next step is the Fully Connected layer, and the output of the final Convolutional Layer needs to be formatted from 1779×16 to 28464×1. As described, the Dense Layer consists of 800 kernels, one for each of the 800 classes. Each kernel is composed of 28464 floats, meaning that, at this point, this is the main bottleneck of the architecture performance-wise.

The Fully connected layer is fully pipelined with interval 1, meaning that in each cycle, the layer consumes one input data and transmits one result per cycle after the completion of the first matrix-matrix multiplication. The weights are 32-bit long because, as was previously explained, they are of type float. A 22 million cycle delay occurred before all the data (28464x800 floats) could be accessed. Furthermore, as the weights of the dense layer were not able to be stored in the B-RAM, we streamed them from the DDR.

For this reason, we experimented with different streamed data widths in the DMA from 32 bits (one float per cycle) to 64 bits (two floats per cycle) and later to 128 bits (four floats per cycle).

	32bits	64bits	128bits
LUT usage	3%	4%	5%
FlipFlop usage	2%	2%	3%
DSP usage	1%	1%	1%
BRAM usage	4%	4%	5%

TABLE 5.5: Resources Utilization for the Dense Layers for different Bus Width

	32bits	64bits	128bits
Latency	76.1ms	38.7ms	20.3ms
Bandwidth	1.1GB/s	2.19GB/s	4.18GB/s

TABLE 5.6: Latency and Bandwidth with different Bus Width

	32bits	64bits	128bits
Total On-chip power	4.236W	4.298W	4.343W
Total On-chip Energy	0.33J	0.16J	0.088J

TABLE 5.7: Power and Energy for the Dense Layers with different Bus Width

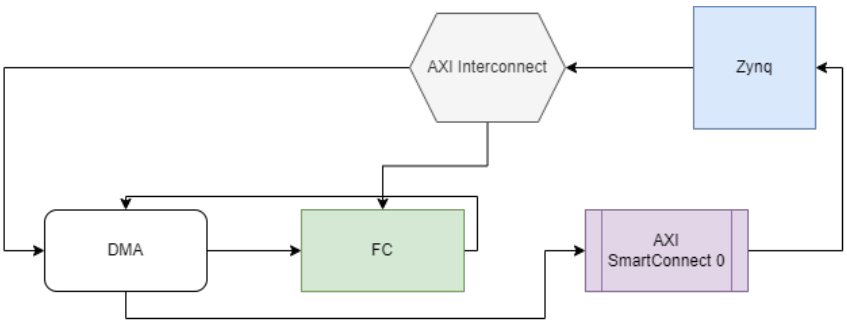

FIGURE 5.6: Architecture 1 Dense Layer Design

5.4 IO Problems - Memory BW

In our first architecture on the FPGA, the performance is sub-optimal compared to the GPU numbers. As can be easily perceived from the results of the Dense Layer, adding more bandwidth by making the memory bus of the DMA larger, from 32bits to 128bits, yielded significant results in terms of latency and total energy with an insignificant hit on the available resources and the total on-chip power. Specifically, accessing four floats in a single cycle improved the latency by a factor of 3.74x and the energy by 3.75x by adding 1% of usages on LUTs, FlipFlops, and BRAM. This is due to the fact that the Fully Connected Layer's weights were streamed through the DMA and could not be stored.

The next step is to study the bandwidth of the FPGA and the PS DDR and consider using the PL DDR memory to accommodate the bandwidth requirements and the different methods for data transfer. We will also add more DMAs to our architecture, which will significantly impact the performance of the dense layer, the main bottleneck of the first Architecture.

Simple projects that read and write randomly generated values from and to the DMA, respectively, were created to test the bandwidth of both channels of the DMAs, read and write, and the potential difference using cache can make to the bandwidth.

	Read	Read/Write
With Cache	8.44 GB/s	17.54 GB/s
Without Cache	6.22 GB/s	11.65 GB/s

TABLE 5.8: Max Bandwidth of DMAs

5.5 Architecture 1 - 4 DMAs

At this point, we realized that the next step was to add more DMAs into the bottleneck of our architecture, the Fully Connect layer, so we could utilize the available bandwidth we had calculated on section 5.4 until we reached the theoretical bandwidth of the memory 16GB/s.

In this version of our architecture, we modified the Fully Connected Layer FPGA of the two-stage architecture Architecture to use four DMAs with a bus width of 128bits in order to utilize the full potential bandwidth of the HP Ports on the ARM processor on the ZCU102. The necessary modifications

were made to the accelerator to compute one-fourth of the final output, and we proceeded with four instances of the modified accelerator.

As each accelerator instance is now responsible only for 200 out of the 800 original classes, the input is adjusted to include the weights for the classes each instance is responsible for and the corresponding biases alongside the feature map produced from the Convolution Layers.

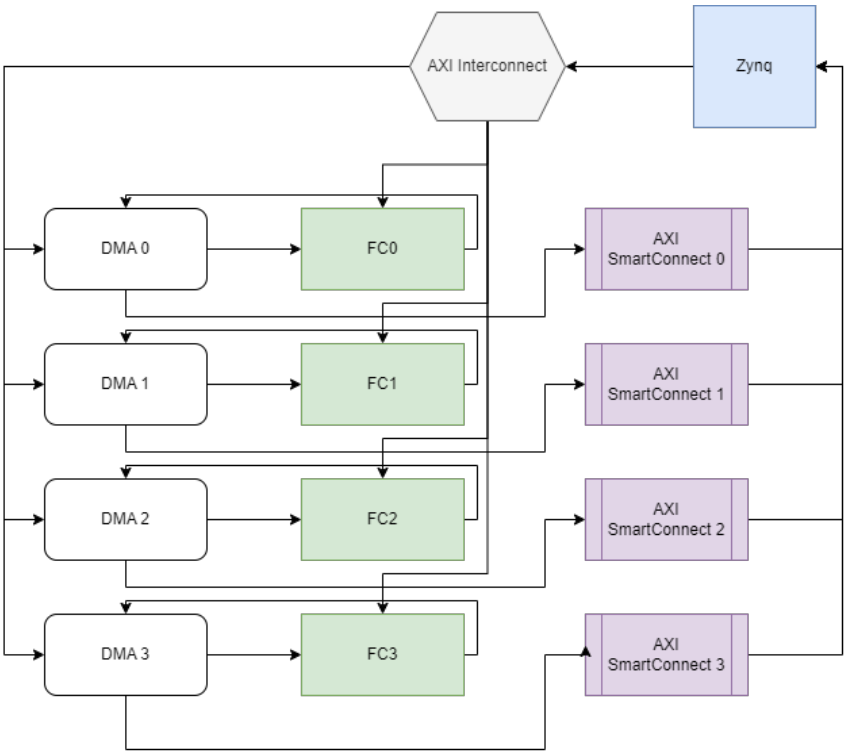

FIGURE 5.7: Architecture 1 Dense Layer with 4 DMAs Design

	32bits	128bits	4DMAs 128bits
LUT usage	3%	5%	15%
FlipFlop usage	2%	3%	11%
DSP usage	1%	1%	3.2%
BRAM usage	4%	5%	16.4%

TABLE 5.9: Resources Utilization for the Dense Layer with 4 DMAs

	32bits	128bits	4DMAs 128bits
Latency	76.1ms	20.3ms	9.1ms
Bandwidth	1.1GB/s	4.18GB/s	9.23GB/s

TABLE 5.10: Latency and Bandwidth with different Bus Width

In the following table 5.11, we compare the total On-chip power and energy of the new Fully Connected Layer Architecture using 4 DMAs to the first Architecture with only one DMA for the Dense Layer.

	32bits	128bits	128bits 4DMAs
Total On-chip power	4.236W	4.343W	5.33W
Total On-chip Energy	0.33J	0.088J	0.048J

TABLE 5.11: Power and Energy for the Dense Layers with different Bus Width

By using four 128-bit width DMAs instead of one, we were able to improve the latency of the Fully Connected Layer by a factor of 2.2x and the energy required to produce the results of one image by a factor of 1.8x.

5.6 Final Version

In this Architecture, we integrated the Robustness and Sensitivity Analysis made by Giorgos Pitsis [50], effectively reducing the memory footprint of the weights with minimum error. This allowed us to improve the speed of accessing and processing our data.

In this architecture, as the memory footprint has been reduced significantly, we are able to fit the entire CNN into a single FPGA. We have also changed the CNN's output to only the selected class and its value, in contrast to the previous architectures, where we outputted all the classes with the corresponding values.

We are using 3 DMAs to transfer data to and from the CNN. The first DMA's, DMA 0, read channel has a 32-bit width and is used to transfer the weights, biases, and images to the Convolution Layers Accelerator. The write channel is responsible for writing the final classification for each image to the DDR. The read channels of DMA 1 and DMA 2 have a 128-bit width. They are responsible for transferring the Fully Connected Layer's corresponding biases, codebook, and weights to each instance of the Fully Connected Accelerator. The write channels of both DMA 1 and 2 are disabled.

As we are using the compressed weights for the Fully Connected layer, and each weight now has a 4-bit size compared to the 32-bit before the compression, we are able to stream combined 64 weights per cycle through the DMA 1 and DMA 2 to the two Accelerators.

Each Fully Connected Accelerator is responsible for calculating the values for 400 of the total 800 Classes. The results for each class are sent to a newly developed IP called Find Max, which is tasked with identifying the highest neuron value. The new IP's input is separated into two streams, one from each instance of the Fully Connected Accelerator. During each iteration, we compare the two floats from the streams representing the values of the classes against each other and the larger of those two with the previous max value, updating and storing the highest value and its corresponding class. After all the values for a single image have been calculated and compared, the selected Class and its value are written to the DDR.

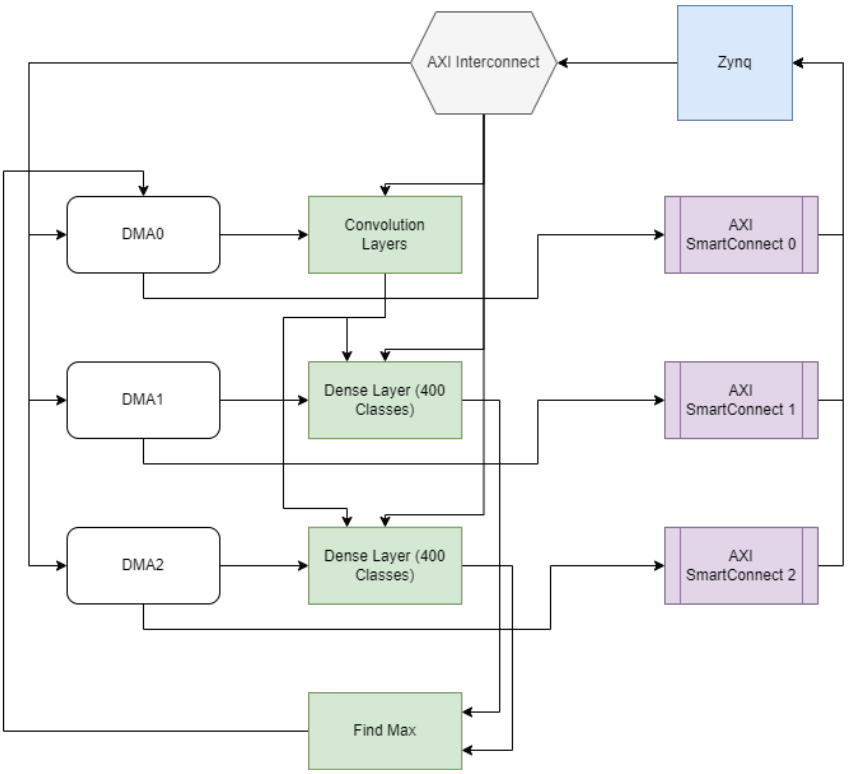

FIGURE 5.8: Final Architecture Design

In table 5.12, we present the Resources Utilization for Architecture 1.

LUT usage	39.4%
FlipFlop usage	38.6%
DSP usage	19.4%
BRAM usage	11.5%

TABLE 5.12: Resources Utilization for Batch 1

5.6.1 ZCU102

As with the previous Architectures, we targeted the ZCU102. In table 5.13, we present the Total Power and Energy needed to produce the classification of the entire 10k dataset.

Total Power On-chip	10.69W
Total Energy On-chip	170J

TABLE 5.13: Energy and Power for the Batch 1 on ZCU102

5.6.2 QFDB

For this version of the Architecture, we also targeted the QFDB. We created four instances of the Architecture, one for each FPGA of the QFDB. Each instance calculates the results of one-fourth of the entire 10k dataset, meaning each FPGA is responsible for classifying 2500 images. As we are calculating simultaneously four results, we have effectively batch 4.

Porting the Architecture from the ZCU102 to the QFDB did not require major changes to the architecture as both platforms carry FPGAs of the same family. The only significant change is that the FPGAs on the QFDB, at the time of this thesis, did not have access to a Micro SD card. As a result, we have to transfer the data to the FPGAs via JTAG. To accomplish this, we need to create several new files in the Vivado SDK that will hold the data alongside a set of new functions for reading the data from these files.

Three sets of files were created:

- one for containing the bias, weights for the three Convolutional Layers
- one that holds the bias, weights, and codebook for the Fully Connected Layer
- one for the images

Of the three sets, the ones containing the CNN parameters are the same for all the FPGAs on the QFDB. On the contrary, the last files holding the images differ, as each FPGA is responsible for a different set of images.

Finally, the FPGA transmits the outputs to our machine using the JTAG cable and the CoreSight protocol, and we have dedicated terminals on the PC, one for each FPGA.

Total Power On-chip	42.76W
Total Energy On-chip	162.5J

TABLE 5.14: Energy and Power for the Batch 1 on QFDB

5.6.3 Power Optimizations

After the initial synthesis, implementation, and verification of Architecture 1's expected behavior on both ZCU102 and QFDB platforms, we were able to perform Power Optimizations to further improve power consumption.

This was achieved by tinkering with the different strategy options in the implementation phase in Vivado. It is worth noting that as we are instructing the tools to perform several optional steps for optimizations or run multiple passes of optimization in each step, the time needed to complete the implementation is increasing significantly. There have been cases where a single implementation run lasted more than three or four hours, even when the task ran on the MHL's Zeus server.

In the following tables, we present the Total Power On-chip and Total Energy On-chip for Architecture 1 after the power optimizations for both ZCU102 5.15 and QFDB 5.16

Total Power On-chip	8.493W
Total Energy On-chip	135.8J

TABLE 5.15: Energy and Power for the Batch 1 on ZCU102 after Power Optimizations

Total Power On-chip	34W
Total Energy On-chip	129.2J

TABLE 5.16: Energy and Power for the Batch 1 on QFDB after Power Optimizations

With the power optimizations, we improved the Total Power On-chip and Total Energy On-chip by a factor of 1.25x.

5.7 Batch 2

As shown in the table 5.12, we do not use the majority of the FPGA's resources at our disposal. This led to the introduction of another level of parallelism, batching of images, instead of computing the result of one image, computing the results of the images in parallel. The first approach would

be adding another instance of each Accelerator, effectively doubling the resources utilized. This would make it impossible for the Vivado to route this at the targeted clock of 300MHz. A better approach is to integrate the processing of the two images inside the Accelerators in Vivado HLS, avoiding the duplication of the resources and making it easier to achieve a higher clock frequency.

Because the weights, even after the reduction, are taking the majority of the I/0 of the FPGA, accessing a second image increases our I/0 only by 0.0001%

LUT usage	64%
FlipFlop usage	58%
DSP usage	38%
BRAM usage	16%

TABLE 5.17: Resources Utilization for the Batch 2

From a resource perspective, the FPGA is able to process the data for two images simultaneously. As both the resources as well as the complexity of the Architecture have increased significantly, we were able to achieve a clock frequency of 250MHz on the FPGA, which is lower compared to the 300MHz of the Batch 1 5.6.

5.7.1 ZCU102

As with the previous Architectures, we targeted the ZCU102. Table 5.18 presents the Total Power and Energy needed to produce the results for the 10k images.

Total Power On-chip	13.66W
Total Energy On-chip	126J

TABLE 5.18: Energy and Power for the Batch 2 on ZCU102

5.7.2 QFDB

For this version of the Architecture, we also also targeted the QFDB. As before, we created four instances of the Architecture, one for each FPGA of the QFDB, and each instance calculated the results for 2500 images. As we are calculating eight results simultaneously, we have effectively batch 8.

Total Power On-chip	54.64W
Total Energy On-chip	126J

TABLE 5.19: Energy and Power for the Batch 2 on QFDB

Chapter 6

Implementation and Performance Evaluation of the FPGA Architecture

In this chapter, the results of our Architecture will be presented, obtained from both target platforms, ZCU-102 and QFDB.

6.1 Specification of Compared Platforms

Our architecture was compared with both CPU and GPU platforms. The CPU used for benchmarking was an Intel i7 7700HQ released in Q1 2017, and the GPU was an NVIDIA Quadro K2200 released in Q3 of 2014. A platform with 5 NVIDIA Quadro K2200 GPUs was also used to compare the first architecture.

It is important to note that the results presented were obtained from actual runs of the network, and are not from simulations, as we downloaded the CNN to all the target platforms.

6.1.1 Intel i7 7700HQ

The Intel Core i7 7700HQ is a quad-core processor from Intel's 7th generation Kaby Lake series, launched in early 2017. It was developed on a 14nm node and is commonly found in high-performance laptops and mobile workstations.

Thermal Design Power (TDP) represents the average power, in watts, the processor dissipates when operating at Base Frequency with all cores active under an Intel-defined, high-complexity workload.

Cores	4
Threads	8
Base Clock Frequency	2.8 GHz
Turbo Clock Frequency	3.8 GHz
TDP	45W

TABLE 6.1: Intel i7 7700HQ Specification

6.1.2 NVIDIA Quadro K2200

The Quadro K2200 is a professional graphics card by NVIDIA, launched in Q3 2014. It is built on the 28nm process and is part of the Quadro Kepler Series.

CUDA Cores	640
GPU Memory	4GB
Base Clock Frequency	1046 MHz
Boost Clock Frequency	1124 MHz
Operational Power Consumption	60W
Max Power Consumption	68W

TABLE 6.2: NVIDIA Quadro K2200 Specification

6.2 Power Consumption

Power consumption refers to the amount of energy used per unit of time for the platform to perform a certain task and is measured in Watts (W) or kiloWatts (kW). For this thesis, we have to take into consideration all the power used by the systems hosting the CPU, GPU, and FPGA. Using the official data sheets of the platforms and tools providing the power consumption of the different components needed, we calculated that the system hosting the CPU consumes 100W and the one hosting the GPU 300W.

6.3 Energy Consumption

Energy consumption is the total energy required from the platform in order to complete a task and can be calculated with the following

$$Energy = Power * Time$$

with *Power* being the required power, and *Time* the required time for completing the task. Energy is measured in Joule (J) or kiloJoule (kJ).

6.4 Throughput and Latency Speedup

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

$$Latency = \frac{1}{v} = \frac{T}{W}$$

where v is the execution speed of the task, T is the execution time of the task, and W is the execution workload of the task.

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

$$Throughput = r * v * A = \frac{r * A * W}{T} = \frac{r * A}{L}$$

with r being the execution density and A the execution capacity.

Speedup is defined for both throughput and latency using the following equations

$$S_{throughput} = \frac{throughput_2}{throughput_1}$$

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

with p being the portion of the task that benefits from the optimizations and s the speedup of the part benefiting.

Amdahl's law is a principle that states that the maximum potential improvement to the performance of a system is limited by the portion of the system that cannot be improved. The maximum theoretical speedup can be calculated with:

$$MaxSpeedup = \lim_{x \to \infty} S_{latency} = \frac{1}{1-p}$$

6.5 Performance

6.5.1 First Architecture

We will present the results of our First Architecture both for the case where we used 1 DMA for the Fully Connected Layer 5.3 and the architecture that was completed after the Bandwidth study, where we used 4 DMAs for the Fully Connected Layer 5.5. These architectures were ported only to ZCU102.

In this stage of our architecture, we compare the results with the CPU, GPU, and 5 GPU platforms. Furthermore, this Architecture did not use the entire dataset of 10k images, so the comparisons are made over latency for the results of one image.

1 DMA

The following table presents the results for the First Architecture using 1 DMA in the Dense Layer stage. The platform with one GPU was used as the base for the comparisons.

	ZCU102		CPU	GPU	5xGPU
	Conv	FC			
Total Power	9.759W	4.343W	100W	300W	540W
	14.102W				
Latency	1.982ms	20.3ms	7.6s	4ms	1ms
	22.282ms				
Latency Speedup	0.179x		0.0005x	1x	4x
Total Energy	0.0193J	0.088J	760J	1.2J	0.54J
	0.107J				
Energy Efficiency	11.2x		0.001x	1x	2.2x

TABLE 6.3: Performance for 1 DMA Architecture

In the following table, we present the improvements of the First architecture compared to the CPU, GPU, and 5x GPU platforms. The results are very promising as our architecture is faster in terms of latency compared to the CPU and more energy efficient in producing the first result compared to all platforms, mainly due to the power used by the whole system hosting the platforms.

	CPU	GPU	5xGPU
Latency Speedup	341x	0.179x	0.044x
Energy Efficiency	7102x	11.2x	5.04x

TABLE 6.4: Performance Comparison for 1 DMA Architecture

4 DMA

The following table 6.5 presents the results for the First Architecture using 4 DMAs for the Dense Layer stage. The platform with one GPU was used as the base for the comparisons.

	ZCU102		CPU	GPU	5xGPU
	Conv	FC			
Total Power	9.759W	5.33W	100W	300W	540W
	15.089W				
Latency	1.982ms	9.1ms	7.6s	4ms	1ms
	11.082ms				
Latency Speedup	0.36x		0.0005x	1x	4x
Total Energy	0.0193J	0.048J	760J	1.2J	0.54J
	0.0673J				
Energy Efficiency	62.1x		0.001x	1x	2.2x

TABLE 6.5: Performance for 4 DMA Architecture

In table 6.6, we present the improvements of the First architecture with 4 DMAs compared to the CPU, GPU, 5x GPU platforms, and the initial Architecture with 1 DMA for the Fully Connected layer. The results are very promising as our architecture is faster in terms of latency compared to the CPU and the previous architecture due to better use of the available bandwidth and more energy efficient to produce the first result compared to all platforms, mainly due to the high amount of power used by the whole systems hosting the GPUs.

	1 DMA	CPU	GPU	5xGPU
Latency Speedup	2x	685x	0.36x	0.09x
Energy Efficiency	1.59x	11292x	17.8x	8.0x

TABLE 6.6: Performance Comparison for 4 DMA Architecture

6.5.2 Final Architecture

In this Section, we will present the results of the Final Architecture 5.6 ported on both ZCU102 and QFDB. This version of the Architecture is also

compared to a newer version of the CNN on a single GPU platform (GPU v2).

In this stage of the Architecture, we are using the entire dataset of 10k images. We will introduce, alongside the latency, energy, and power efficiency, the metrics throughput measured in this thesis in Images per second and the very promising Images per Joule, which provided a new dimension to meter efficiency.

For the comparisons, we will use the Power and Energy reported after the power optimizations 5.6.3.

ZCU 102

The table 6.7 presents the results for the Final Architecture ported on ZCU102. The original implementation with one GPU (GPU v1) was used as the base for the comparisons.

	ZCU102	CPU	GPU v1	GPU v2	5xGPU
Total Power	8.493W	100W	300W	300W	540W
Latency	3ms	7.6s	4ms	60ms	1ms
Total Energy	135.8J	288kJ	12kJ	1.5kJ	5.4kJ
Energy Efficiency	88.3x	0.04x	1x	8x	2.2x
Throughput	625	3.47	250	2000	1000
Images/Joule	73.6	0.035	0.83	6.66	1.85

TABLE 6.7: Performance for Final Architecture on ZCU102

In table 6.8, we present the improvements of the Final Architecture ported on ZCU102 compared to the CPU, GPU v1, GPU v2, and 5x GPU platforms. Compared to the CPU platform, our architecture shows improvements in both latency and throughput, as well as greater energy and power efficiency. Compared to the GPU platforms, our architecture has improved throughput only against the GPU v1 implementation, but on the other hand, we have better energy efficiency against all the GPU platforms.

	CPU	GPU v1	GPU v2	5xGPU
Latency Speedup	2533x	1.3x	20x	0.33x
Throughput speedup	180x	2.5x	0.31x	0.625x
Energy Efficiency	2120x	88.3x	11.04x	39.7x

TABLE 6.8: Performance Comparison for Final Architecture on ZCU102

QFDB

The table 6.9 presents the results for the Final Architecture ported on QFDB using the entire dataset of 10k images. The original implementation with one GPU (GPU v1) was used as the comparison base.

	QFDB	CPU	GPU v1	GPU v2	5xGPU
Total Power	34W	100W	300W	300W	540W
Latency	3ms	7.6s	4ms	60ms	1ms
Total Energy	129.2J	288kJ	12kJ	1.5kJ	5.4kJ
Energy Efficiency	92.87x	0.04x	1x	8x	2.2x
Throughput	2756	3.47	250	2000	1000
Images/Joule	77	0.035	0.83	6.66	1.85

TABLE 6.9: Performance for Final Architecture on QFDB

In table 6.10, we present the improvements of the Final Architecture ported on QFDB compared to the CPU, GPU v1, GPU v2, and 5x GPU platforms. Our architecture ported on the QFDB shows significant improvements against all platforms because we effectively have batch 4 as we are using the four FPGAs on QFDB. Furthermore, the architecture is more efficient on energy metrics.

	CPU	GPU v1	GPU v2	5xGPU
Latency Speedup	2533x	1.3x	20x	0.33x
Throughput speedup	794x	11.02x	1.37x	2.75x
Energy Efficiency	2229x	92.8x	11.6x	41.8x

TABLE 6.10: Performance Comparison for Final Architecture on QFDB

6.5.3 Batch 2

In this Section, we will present the results of the Batch 2 ported on both ZCU102 and QFDB. This version of the Architecture is compared with the same platforms that were used in 6.5.2. Furthermore, the entire dataset of 10k images was used to extract these results.

It is important to note that although the final Batch 2 was completed in C. Loukas's thesis and in his work the Batch 2 Architecture was downloaded to a different platform (ExaNeSt Design), since the task's computational load has not changed, the results presented in this thesis, taken from actual runs on both platforms used, are accurate.

ZCU 102

The table 6.11 presents the results for the Batch 2 Architecture ported on ZCU102. The original implementation with one GPU (GPU v1) was used as the comparison base.

	ZCU102	CPU	GPU v1	GPU v2	5xGPU
Total Power	13.66W	100W	300W	300W	540W
Latency	3ms	7.6s	4ms	60ms	1ms
Total Energy	126J	288kJ	12kJ	1.5kJ	5.4kJ
Energy Efficiency	11.9x	0.04x	1x	8x	2.2x
Throughput	1084	3.47	250	2000	1000
Images/Joule	79.3	0.035	0.83	6.66	1.85

TABLE 6.11: Performance for Batch2 on ZCU102

In table 6.12, we present the improvements of the Batch 2 Architecture ported on ZCU102 compared to the CPU, GPU v1, GPU v2, and 5x GPU platforms. Our architecture shows significant improvements against all platforms.

	CPU	GPU v1	GPU v2	5xGPU
Latency Speedup	2533x	1.3x	20x	0.33x
Throughput speedup	312x	4.3x	0.54x	1.08x
Energy Efficiency	2285x	95.2x	11.9x	42.8x

TABLE 6.12: Performance Comparison for Batch 2 Architecture on ZCU102

QFDB

The table 6.13 presents the results for the Batch 2 Architecture ported on QFDB using the entire dataset of 10k images. The original implementation with one GPU (GPU v1) was used as the comparison base.

	QFDB	CPU	GPU v1	GPU v2	5xGPU
Total Power	54.64W	100W	300W	300W	540W
Latency	3ms	7.6s	4ms	60ms	1ms
Total Energy	126J	288kJ	12kJ	1.5kJ	5.4kJ
Energy Efficiency	11.9x	0.04x	1x	8x	2.2x
Throughput	4334	3.47	250	2000	1000
Images/Joule	79.3	0.035	0.83	6.66	1.85

TABLE 6.13: Performance for Batch2 on QFDB

In table 6.14, we present the improvements of the Batch 2 Architecture ported on QFDB compared to the CPU, GPU v1, GPU v2, and 5x GPU platforms.

Our architecture ported on the QFDB shows significant improvements on every metric against all platforms because we effectively have batch 8, as we are using the four FPGAs on QFDB.

	CPU	GPU v1	GPU v2	5xGPU
Latency Speedup	2533x	1.3x	20x	0.33x
Throughput speedup	1249x	17.3x	2.16x	4.33x
Energy Efficiency	2285x	95.2x	11.9x	42.8x

TABLE 6.14: Performance Comparison for Batch 2 Architecture on QFDB

6.5.4 Final Comparisons and Discussions

The following figures present the final results regarding latency improvements achieved over the different stages of this thesis. Furthermore, we present results and comparisons versus the other platforms regarding energy efficiency using the Images per Joule metric and throughput using the Images per second metric that was achieved using the entire 10k dataset.

In figure 6.1, we present the improvement in latency over the course of this Thesis, from the first naive Architecture to the Final one.

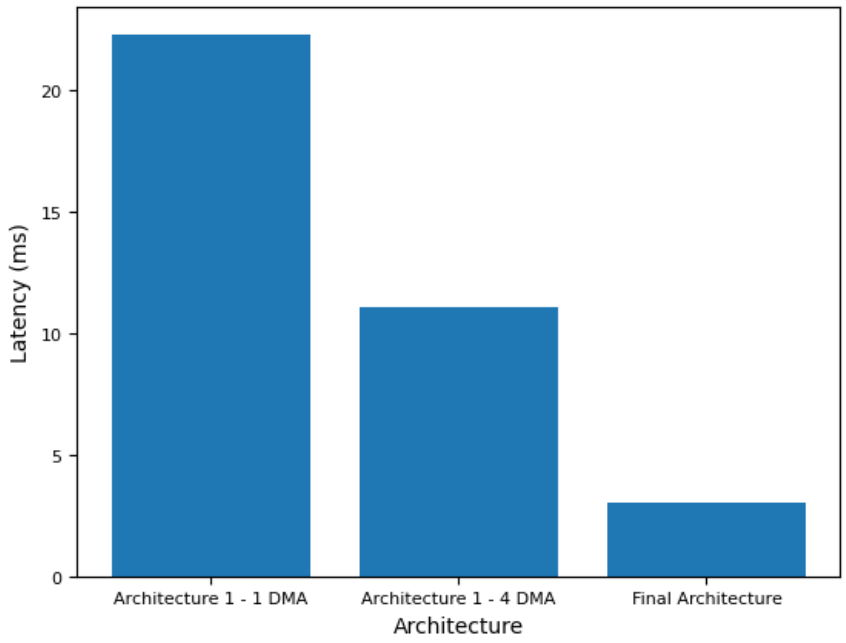

FIGURE 6.1: Latency Improvements

In figure 6.2, we present the improvement in energy efficiency metered in Images per Joule that we achieved over all the platforms.

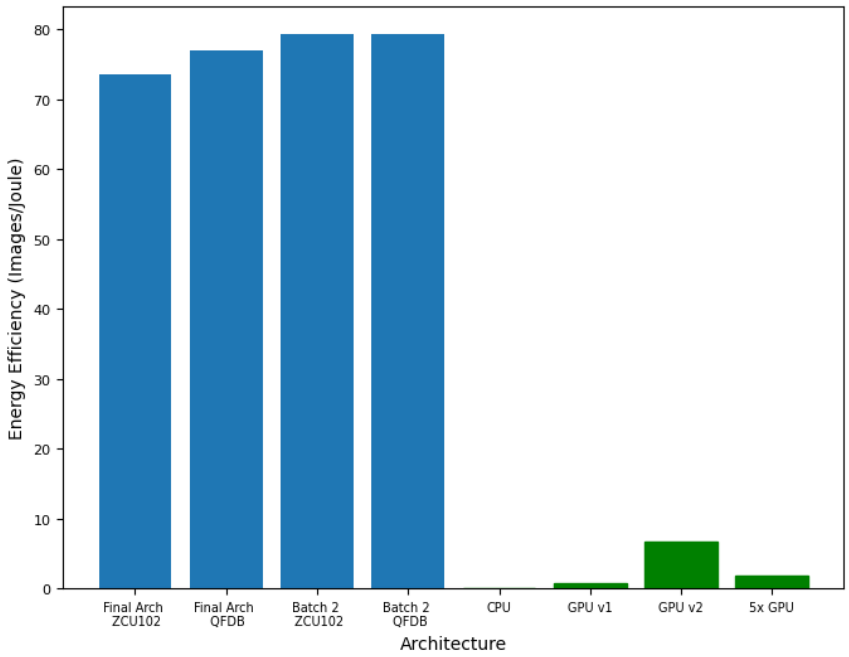

FIGURE 6.2: Energy Efficiency Improvements

In figure 6.3, we present results in terms of throughput. We achieved a 2.16x throughput speedup over the best GPU platform (GPU v2) using the four FPGAs on the QFDB.

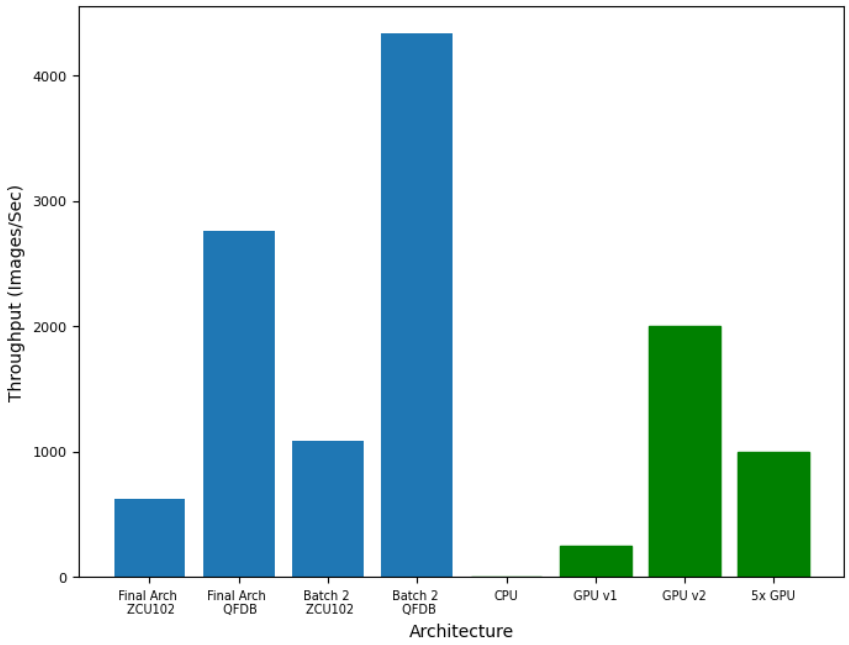

FIGURE 6.3: Throughput Improvements

The results regarding Energy Efficiency using the Images per Joule metrics are promising, as we achieved up to 11.9x vs. the best GPU platform (GPU

v2). Although more recent GPUs may have a 4x energy efficiency improvement over the K2200 GPU we used for the comparisons, there are also newer FPGA platforms, and the FPGAs on the QFDB are of the same technology as the NVidia K2200. We expect the trend to continue favoring the FPGA platform regarding energy efficiency.

Chapter 7

Conclusions and Future Work

In this Chapter, we will sum up and evaluate the research conducted in this thesis. Furthermore, we will explore potential areas for future investigation and identify specific aspects for further improvement.

7.1 Conclusions

In this thesis, we proposed various FPGA architectures to accelerate a specific CNN designed for processing redshift data from the Euclid satellite. Our target platforms were the ZCU102 evaluation board as well as the QFDB platform, which hosts 4 FPGAs of the same family as the ZCU102 and was developed at FORTH. Firstly, we analyzed the network from a computational and memory access standpoint. Multiple architectures were designed and implemented to exploit various opportunities for parallelism and achieve better utilization of the available resources and bandwidth. Furthermore, various optimizations were made to achieve better power consumption. The proposed architectures achieved up to 11.9x in terms of energy efficiency compared to the best platform compared while having a 2.16x throughput speedup.

7.2 Future Work

As future work, we could scale this work to the Mezanine, a platform hosting four QFDBs, meaning 16 FPGAs in total, which we expect to have a linear speedup due to the parallelism of our application as we would effectively have a batch of 32. This could be achieved by incorporating the Architecture into the ExaNeSt design. Modern and larger FPGAs can be targeted with an

increase of the Accelerator's internal parallelism, resulting also in a close to linear speedup.

Furthermore, the streaming of the data on the QFDB to the accelerators through the built-in ethernet can be investigated, as the current Architecture, as mentioned, uses JTAG for streaming.

Finally, master-slave models can be investigated to utilize any remaining available resources in each of the FPGAs by having the Convolutional Layers in one FPGA of the QFDB, the Fully Connected layers in another, and data transfer implemented between them with the Aurora Protocol. This could effectively increase the batch size and the platform's efficiency.

Part of the recommended Future Work has been completed by Loukas Charisios in his thesis [54]

References

[1] *EuroExa*. URL: https://euroexa.eu/.

[2] Kaiyuan Guo et al. "A Survey of FPGA Based Neural Network Accelerator". In: *CoRR* abs/1712.08934 (2017). arXiv: 1712.08934. URL: http://arxiv.org/abs/1712.08934.

[3] Chen Zhang et al. "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170. ISBN: 9781450333153. DOI: 10.1145/2684746.2689060. URL: https://doi.org/10.1145/2684746.2689060.

[4] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229. DOI: 10.1147/rd.33.0210.

[5] M.H. Hassoun. "Fundamentals of Artificial Neural Networks". In: *Proceedings of the IEEE* 84.6 (1996), pp. 906–. DOI: 10.1109/JPROC.1996.503146.

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

[7] TensorFlow. URL: https://www.tensorflow.org/.

[8] Martín Abadi et al. *TensorFlow: A system for large-scale machine learning*. 2016. arXiv: 1605.08695 [cs.DC]. URL: https://arxiv.org/abs/1605.08695.

[9] Jeffrey Dean et al. "Large scale distributed deep networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1223–1231.

[10] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467 [cs.DC]. URL: https://arxiv.org/abs/1603.04467.

[11] Nikhil Ketkar. "Introduction to Keras". In: *Deep Learning with Python: A Hands-on Introduction*. Berkeley, CA: Apress, 2017, pp. 97–111. ISBN: 978-1-4842-2766-4. DOI: 10.1007/978-1-4842-2766-4_7. URL: https://doi.org/10.1007/978-1-4842-2766-4_7.

[12] Keras. URL: https://keras.io/.

[13] Navin Kumar Manaswi. "Understanding and Working with Keras". In: *Deep Learning with Applications Using Python : Chatbots and Face, Object, and Speech Recognition With TensorFlow and Keras*. Berkeley, CA: Apress, 2018, pp. 31–43. ISBN: 978-1-4842-3516-4. DOI: 10.1007/978-1-4842-3516-4_2. URL: https://doi.org/10.1007/978-1-4842-3516-4_2.

[14] Keras. *Keras Documentation: The sequential model*. URL: https://keras.io/guides/sequential_model/.

[15] Keras. *Keras Documentation: The functional API*. URL: https://keras.io/guides/functional_api/.

[16] Yangqing Jia et al. *Caffe: Convolutional Architecture for Fast Feature Embedding*. 2014. arXiv: 1408.5093 [cs.CV]. URL: https://arxiv.org/abs/1408.5093.

[17] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG]. URL: https://arxiv.org/abs/1912.01703.

[18] PyTorch. URL: https://pytorch.org/.

[19] Dan C. Cireşan et al. "Flexible, High Performance Convolutional Neural 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.

[20] Khoa Ho et al. "Improving GPU Throughput through Parallel Execution Using Tensor Cores and CUDA Cores". In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2022, pp. 223–228. DOI: 10.1109/ISVLSI54635.2022.00051.

[21] Sharan Chetlur et al. *cuDNN: Efficient Primitives for Deep Learning*. 2014. arXiv: 1410.0759 [cs.NE]. URL: https://arxiv.org/abs/1410.0759.

[22] NVIDIA. URL: https://developer.nvidia.com/cuda-toolkit.

[23] Xiaqing Li et al. "Performance Analysis of GPU-Based Convolutional Neural Networks". In: *2016 45th International Conference on Parallel Processing (ICPP)*. 2016, pp. 67–76. DOI: 10.1109/ICPP.2016.15.

[24] Andrew Putnam et al. "A reconfigurable fabric for accelerating large-scale datacenter services". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 13–24. DOI: 10.1109/ISCA.2014.6853195.

[25] Kalin Ovtcharov et al. *Accelerating Deep Convolutional Neural Networks Using Specialized Hardware*. Feb. 2015. URL: https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/.

[26] Jeremy Fowers et al. "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications". In: Feb. 2012, pp. 47–56. DOI: 10.1145/2145694.2145704.

[27] Eriko Nurvitadhi et al. "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: ACM, 2017, pp. 5–14. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021740. URL: http://doi.acm.org/10.1145/3020078.3021740.

[28] Tianshi Chen et al. "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning". In: *SIGARCH Comput. Archit. News* 42.1 (Feb. 2014), pp. 269–284. ISSN: 0163-5964. DOI: 10.1145/2654822.2541967. URL: https://doi.org/10.1145/2654822.2541967.

[29] Yunji Chen et al. "DaDianNao: A Machine-Learning Supercomputer". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014, pp. 609–622. DOI: 10.1109/MICRO.2014.58.

[30] Song Han et al. "EIE: Efficient Inference Engine on Compressed Deep Neural Network". In: *CoRR* abs/1602.01528 (2016). arXiv: 1602.01528. URL: http://arxiv.org/abs/1602.01528.

[31] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. "You Cannot Improve What You Do not Measure: FPGA vs. ASIC Efficiency Gaps for Convolutional Neural Network Inference". In: *ACM Trans. Reconfigurable Technol. Syst.* 11.3 (Dec. 2018). ISSN: 1936-7406. DOI: 10.1145/3242898. URL: https://doi.org/10.1145/3242898.

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

[33] Kaz Sato, Cliff Young, and David Patterson. "An in-depth look at Google's first Tensor Processing Unit (TPU)". In: *Google Cloud Big Data and Machine Learning Blog* 12 (2017). URL: https://cloud.google.com/blog/

products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[34] Google. URL: https://cloud.google.com/blog/products/compute/introducing-trillium-6th-gen-tpus.

[35] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. *Hardware-oriented Approximation of Convolutional Neural Networks*. 2016. arXiv: 1604.03168 [cs.CV]. URL: https://arxiv.org/abs/1604.03168.

[36] A. N. Shchelokov et al. *Hardware implementation of a Convolutional Neural Network in FPGA based on Fixed Point Calculations*. 2017. URL: https://arxiv.org/pdf/1808.09945.pdf.

[37] Liqiang Lu et al. "Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs". In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2017, pp. 101–108. DOI: 10.1109/FCCM.2017.64.

[38] Matthieu Courbariaux et al. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. arXiv: 1602.02830 [cs.LG]. URL: https://arxiv.org/abs/1602.02830.

[39] Mohammad Rastegari et al. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks". In: *Computer Vision – ECCV 2016*. Ed. by Bastian Leibe et al. Cham: Springer International Publishing, 2016, pp. 525–542. ISBN: 978-3-319-46493-0. URL: https://arxiv.org/abs/1603.05279.

[40] Xiaofan Lin, Cong Zhao, and Wei Pan. "Towards accurate binary convolutional neural network". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 344–352. ISBN: 9781510860964.

[41] Yaman Umuroglu et al. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. ACM, Feb. 2017. DOI: 10.1145/3020078.3021744. URL: http://dx.doi.org/10.1145/3020078.3021744.

[42] Yixing Li et al. "A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks". In: *J. Emerg. Technol. Comput. Syst.* 14.2 (July 2018). ISSN: 1550-4832. DOI: 10.1145/3154839. URL: https://doi.org/10.1145/3154839.

[43] Chen Zhang et al. "Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster". In: *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ISLPED '16. San Francisco Airport, CA, USA: ACM, 2016, pp. 326–331. ISBN: 978-1-4503-4185-1. DOI: 10.1145/2934583.2934644. URL: http://doi.acm.org/10.1145/2934583.2934644.

[44] Xilinx. *Xilinx Aurora 64B/66B*. URL: https://www.xilinx.com/products/intellectual-property/aurora64b66b.html.

[45] Jiantao Qiu et al. "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: Association for Computing Machinery, 2016, pp. 26–35. ISBN: 9781450338561. DOI: 10.1145/2847263.2847265. URL: https://doi.org/10.1145/2847263.2847265.

[46] Yufei Ma et al. "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: ACM, 2017, pp. 45–54. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021736. URL: http://doi.acm.org/10.1145/3020078.3021736.

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

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

[49] R. Laureijs et al. *Euclid Definition Study Report*. 2011. arXiv: 1110.3193 [astro-ph.CO]. URL: https://arxiv.org/abs/1110.3193.

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

[51] Song Han et al. *Learning both Weights and Connections for Efficient Neural Networks*. 2015. arXiv: 1506.02626 [cs.NE]. URL: https://arxiv.org/abs/1506.02626.

[52] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: *CoRR* abs/1510.00149 (2015). arXiv: 1510.00149. URL: http://arxiv.org/abs/1510.00149.

[53] *ZCU102 Evaluation Board User Guide*. URL: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf.

[54] Charisios Loukas. "Large Scale Design and Implementation of Convolutional Neural Networks based on Large FPGA Arrays". 2020. URL: https://dias.library.tuc.gr/view/84315.

Publications

The following publications resulted from this Thesis

[1] 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.

[2] George Pitsis et al. "Efficient Convolutional Neural Network Weight Compression for Space Data Classification on Multi-fpga Platforms". In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 3917–3921. DOI: 10.1109/ICASSP.2019.8682732.