

TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Design Space Exploration of Hardware Accelerated Continual Learning Methods in Convolutional Neural Networks

Author:

Emmanouil PERAKIS

Thesis Committee:

Prof. Apostolos DOLLAS

Prof. Michail LAGOUDAKIS

Asst. Prof. Grigorios

TSAGKATAKIS (UOC)



*A thesis submitted in fulfillment of the requirements
for the diploma of Diploma Thesis*

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

13/09/2023

Chania, September 2023

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Diploma Thesis

Design Space Exploration of Hardware Accelerated Continual Learning Methods in Convolutional Neural Networks

by Emmanouil PERAKIS

Artificial Intelligence (AI) and Machine Learning (ML) have seen indisputable advancements over the years, spanning a large number of branches from medicine and industry related machinery to data analytics and Internet of Things. One way in which Machine Learning on the edge falters is to learn from new, never seen before data, without having access to the previous data. If the model is left as is without any intervention, trying to learn new classes results in catastrophic forgetting. By training a classifier that is separated from the network's parameters the model can learn new tasks without forgetting previously learned ones, and do this at inference time. This is where Continual Learning, and more importantly to this thesis, Streaming Linear Discriminant Analysis comes into play. In this thesis, an accelerator for the previously mentioned method was fully implemented and downloaded on a Field Programmable Gate Array (FPGA) device and compared to other platforms, such as modern CPUs and Graphical Processing Units (GPUs). This accelerator results in fixed point latency that is two orders of magnitude smaller than even CPUs and GPUs, and hundreds of times more energy efficient. The floating point latency speedup is a lot smaller, but still comparable to modern devices, while retaining the energy efficiency.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Diploma Thesis

Design Space Exploration of Hardware Accelerated Continual Learning Methods in Convolutional Neural Networks

by Emmanouil PERAKIS

Η Τεχνητή Νοημοσύνη και η Μηχανική Μάθηση έχουν δει αναμφισβήτητη πρόοδο τα τελευταία χρόνια, βρίσκοντας χρήση σε διάφορους κλάδους από την ιατρική και τα μηχανήματα που χρησιμοποιούνται στη βιομηχανία μέχρι την Ανάλυση Δεδομένων και το Διαδίκτυο των Πραγμάτων (IoT). Ένας τρόπος με τον οποίο η Μηχανική Μάθηση στα υπολογιστικά άκρα (εδγε ζομπυτινγ) αποτυγχάνει είναι το να μάθει από νέα, πρωτοφανή δεδομένα, χωρίς να έχει πρόσβαση στα προηγούμενα δεδομένα. Αν μένει το μοντέλο ως έχει χωρίς κάποια παρέμβαση, τότε η προσπάθεια εκμάθησης νέων κλάσεων οδηγεί σε ένα φαινόμενο που ονομάζεται καταστροφική λήθη. Εκπαιδεύοντας έναν ταξινομητή, ο οποίος είναι διαχωρισμένος από τις παραμέτρους ενός δικτύου, το μοντέλο μπορεί να μάθει νέες έννοιες, χωρίς να ξεχνάει τις προηγούμενες, και όλα αυτά σε χρόνο συμπερασμού. Εδώ είναι που «εισέρχεται στην εξίσωση» η Συνεχής Μάθηση, και σημαντικότερα για τη συγκεκριμένη διατριβή, η Εισρέουσα Γραμμική Διακριτική Ανάλυση (Στρεαμινγ Λινεαρ Δισκριμιναντ Αναλψις). Στην παρούσα διπλωματική εργασία υλοποιήθηκε ένας επιταχυντής για την προαναφερθείσα μέθοδο και προσαρμόστηκε σε συσκευή Προγραμματιζόμενης Συστοιχίας Πυλών Πεδίου (ΦΠΓΑ) και συγκρίθηκε με άλλες πλατφόρμες, όπως μοντέρνους επεξεργαστές και Επξεργαστικές Μονάδες Γραφικών (ΓΠΥς). Ο επιταχυντής επιτυγχάνει, σε αριθμητική σταθερής υποδιαστολής, καθυστέρηση που είναι δύο τάξεις μεγέθους χαμηλότερη από επεξεργαστές, ακόμα και από Επξεργαστικές Μονάδες Γραφικών, αλλά ταυτόχρονα είναι εκατοντάδες φορές πιο ενεργειακά αποδοτικός. Η βελτίωση στην καθυστέρηση, σε αριθμητική κινητής υποδιαστολής, είναι πολύ μικρότερη, αλλά συγκρίσιμη με σύγχρονες συσκευές, παράλληλα διατηρώντας την προαναφερθείσα ενεργειακή απόδοση.

Acknowledgements

I would like to express my gratitude to my supervisor Prof. Apostolos Dolas who made this thesis possible through his irreplaceable guidance and wisdom regarding computer architecture and work ethic in general. The valuable lessons I learned will remain with me forever, and inspire me to keep learning. Being part of the MHL laboratory was also a privilege to have through the new acquaintances I made and the help I was able to provide to fellow undergraduate students.

Furthermore, I would like to deeply thank Asst. Prof. Grigorios Tsagkatakis (UoC), for the invaluable assistance he provided me with regarding more contemporary Machine Learning methods that broadened my horizons in the field. Additionally, his contributions concerning the direction of this thesis were of immense value and helped shape the path to the final results.

I would also like to thank Prof. Michail Lagoudakis for taking the time to review my thesis.

Moreover, I have to thank Dr. Andreas Brokalakis and Laboratory Teaching Staff Markos Kimionis that helped in keeping the systems in the laboratory operate as expected and even upgrading them when needed.

Last but not least, I would like to express my deepest gratitude to my family and friends that helped me push forward through thick and thin and made me become the person I am today.

Contents

Abstract	iii
Abstract	v
Acknowledgements	vii
Contents	ix
List of Figures	xiii
List of Tables	xv
List of Algorithms	xvii
List of Abbreviations	xix
1 Introduction	1
1.1 Motivation	2
1.2 Scientific Contributions	2
1.3 Thesis Outline	4
2 Theoretical Background	5
2.1 Artificial Intelligence	5
2.2 Machine Learning	5
2.3 Deep Learning	7
2.3.1 Perceptron	7
2.3.2 Activation Function	8
2.4 Neural Network Training Steps	11
2.4.1 Feedforward Neural Network	11
Forward Propagation	11
2.4.2 Loss Function	12
2.4.3 Backpropagation and Gradient Descent	13
2.5 Convolutional Neural Networks	14

2.5.1	Convolutional Layers	15
2.5.2	Subsampling Layers	17
2.6	Continual Learning	19
2.7	Continual Learning algorithms that assess the Catastrophic Forgetting problem	20
2.7.1	Elastic Weight Consolidation (EWC)	20
2.7.2	Learning without Forgetting (LwF)	21
2.7.3	Gradient Episodic Memory (GEM)	21
2.7.4	Deep Streaming Linear Discriminant Analysis (Deep SLDA)	22
3	Related Work	25
3.1	CNN architectures	25
3.1.1	AlexNet	25
3.1.2	VGG	26
3.1.3	ResNet	27
3.1.4	DenseNet	27
3.1.5	MobileNets	28
3.2	Datasets previously used in Continual Learning Tasks	31
3.2.1	CIFAR-10/100	31
3.2.2	CoRE50	32
3.2.3	ImageNet	33
3.2.4	CUB200-2011	33
3.3	Thesis Approach	33
3.4	Hardware designs for Matrix-Vector multiplication	34
3.4.1	GPU based designs	34
3.4.2	FPGA based designs	34
3.4.3	Google TPU	35
4	Robustness Analysis	39
4.1	Arithmetic Representation	39
4.1.1	Fixed Point Representation	39
4.1.2	Fixed Point operations: Advantages and Disadvantages	40
4.1.3	Static SLDA over Plastic SLDA	42
4.1.4	Static SLDA algorithms	43
4.1.5	Software Implementation of Static SLDA	44
4.1.6	NumPy	44
4.1.7	CuPy	45
	Pretrained Python CNNs	46

Dataset and Data orderings	46
Network Base Initialization	46
SLDA among different platforms and programming languages	46
5 System Architecture and FPGA Implementation	49
5.1 Hardware Architecture	49
5.1.1 Update Means Unit	53
5.1.2 Compute Weights Unit and Compute Biases Unit	55
5.1.3 Compute Scores Unit	58
5.2 Tools Used	60
5.2.1 Vitis High Level Synthesis (HLS)	60
Optimization Directives	61
5.2.2 Vivado IDE	62
5.2.3 Vitis Unified Software Platform	63
5.3 Platforms and Xilinx Cores	63
5.3.1 ZCU102	63
6 System Verification and Performance Evaluation	67
6.1 System Verification	67
6.2 Specification of Compared Platforms	67
6.2.1 Intel Core i7-11800H	67
6.2.2 NVIDIA RTX 3050 Ti Mobile	68
6.2.3 ZCU102	69
6.3 Implemented Accelerator Characteristics	69
6.4 Experiments that were sought through	70
6.5 Throughput and Latency Speedup	71
6.5.1 Amdahl's Law	71
6.5.2 Metrics used for latency and throughput	72
6.5.3 Comparison of the accelerator to other platforms	72
6.6 Power Consumption and Energy Consumption	77
6.7 Accuracy Metrics	79
6.8 Results Discussion	81
7 Conclusions and Future Work	83
7.1 Conclusions	83
7.2 Future Work	84

8 Appendix A - FPGA design from High Level Synthesis to Application level testing	87
8.1 Vitis HLS algorithms	87
8.1.1 Compute Weights Unit Algorithm	87
8.1.2 Compute Biases Unit Algorithm	89
8.1.3 Compute Scores Unit Algorithm	90
8.2 Vivado IDE block design	90
8.3 Embedded application using the Vitis Software Platform . . .	92
References	97

List of Figures

2.1	Neuron	8
2.2	Perceptron	9
2.3	Activation Functions	11
2.4	FeedforwardNN	12
2.5	backpropagation	14
2.6	LeNet-5	15
2.7	1DConvoloution	16
2.8	2D Convolution	17
2.9	MaxPooling	18
2.10	AvgPooling	19
2.11	EWC	21
2.12	LwF	22
3.1	AlexNet	26
3.2	VGG	27
3.3	Resnet	29
3.4	Densenet	30
3.5	MobileNets	30
3.6	MobileNets	31
3.7	MobileNets	31
3.8	Core50	32
3.9	GEMM	35
3.10	SLDA	36
3.11	SLDA	36
3.12	TPU	37
3.13	TPU arch	37
4.1	FixedPoint Sign magnitude	40
4.2	FixedPoint 1's complement	41
4.3	Distribution	42
5.1	SLDA top top	51

5.2	SLDA top	52
5.3	Means	54
5.4	Weights	56
5.5	Biases	57
5.6	Scores	59
5.8	ZCU102	65
6.1	util percentage	70
6.2	Static SLDA latency among different platforms.	75
6.3	Static SLDA throughput among different platforms.	76
6.4	Static SLDA on ZCU102 total on-chip power.	78
6.5	Accuracy metrics of the continual learning process on two different splits of the CIFAR10 dataset.	80
8.1	SLDAconnections	92

List of Tables

5.1	ZCU102 specification table. URL	64
6.1	Intel Core i7-11800H Processor specifications. URL	68
6.2	NVIDIA RTX 3050 Ti Mobile specifications. URL	68
6.3	ZCU102 PL fabric specifications.	69
6.4	Static SLDA accelerator for fixed point operations specifications.	69
6.5	Static SLDA accelerator for floating point operations specifications.	70
6.6	Static SLDA classifier latency and throughput metrics on CPU with the use of NumPy and without the use of NumPy.	72
6.7	Static SLDA classifier latency and throughput metrics among the different platforms for feature vectors of dimensionality $N = 512$. These results are without the data transferring optimizations	73
6.8	Static SLDA classifier latency and throughput metrics among the different platforms for feature vectors of dimensionality $N = 512$. These results include the data transferring optimizations	74
6.9	Latencies of the whole inference process + the training of the SLDA classifier	77
6.10	Static SLDA classifier power and energy metrics for floating point arithmetic.	79
6.11	Static SLDA classifier power and energy metrics for 16-bit Fixed Point arithmetic.	79

List of Algorithms

1	Init means algorithm	43
2	Update means, weights and biases algorithm	44
3	Compute Scores algorithm	44

List of Abbreviations

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
CPU	Central Processor Unit
CS	Computer Science
DDR4	Double Data Rate type texbf4 memory
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
FF	Flip Flops
FPGA	Field Programmable Gate Array
GDDR6	Graphics Double Data Rate type 6 memory
GPU	Graphic Processor Unit
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HLS	High Level Synthesis
HPC	Hight Performance Computing
LUT	Look Up Table
MPSoC	Multi Processor System on Chip
PL	Programmable Logic
PS	Processing System
RAM	Random Access Memory
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SLDA	Streaming Linear Discriminant Analysis
SSE	Streaming SIMD Extensions
SSD	Solid State Drive
TDP	Thermal Design Power
URAM	Ultra Random Access Memory
USD	United States Dollar

Dedicated to my family and friends...

Chapter 1

Introduction

Over the last few years, Artificial Intelligence (AI) and Machine Learning (ML) in particular, have taken the world by storm and have found their way into almost every gadget, machine or system around us. This has become possible due to both the advancements in machine learning algorithms and models, that achieve state of the art accuracy (sometimes even better than humans in some tasks) and the hardware architectures that specifically target Machine Learning acceleration and perform the tasks needed faster than ever. These advancements have carried the AI scene from just theoretical models to real life applications that better and simplify our lives.

On the CPU front, an evolution in electronics has taken place over the decades, achieving smaller and smaller transistor sizes (currently under 10nm) as it was expected from Moore's Law. Sadly, due to physical limitations on the quantum level the size of transistors can only get so smaller, or else their operation is hindered. That is why many researchers direct their studies to quantum computing or optical computing, which operates using electromagnetic waves. Moreover, the amount of cores in CPUs is steadily increasing as to have greater parallelism and computational capabilities.

GPUs, or Graphical Processing Units have started developing with one target in mind, accelerating graphics in video games and simulations. Nevertheless, due to high demands in gaming performance they reached a place that renders them of great assistance to ML model training and inference. This becomes possible architecturally because GPUs contain a lot more smaller and simpler processing units that are optimized to perform specific operations.

Naturally, with the rise of ML large companies such as Google have adopted their ASIC design for accelerating inference and training. The Google Tensor

Processing Unit is such a device that boasts great inference capabilities in exchange of flexibility in comparison to its aforementioned counterparts.

Most ML models require a period of training that, depending on the size of the dataset, can take minutes, hours, or even days. On the other hand, inference, that is the pass of the data through a trained model to receive a prediction, is a lot faster. That is the reason most edge devices that require real time prediction capabilities, only perform inference calculations on board. Training is most of the time prohibitive in edge devices as it requires a lot more resources and energy than inference.

1.1 Motivation

As it was mentioned in the previous section training on the edge is most of the time prohibited. If the edge device has access to the internet it can connect to a server that performs training and receive back the new model parameters. This has very high latency due to the satellite links needed and may not meet the time requirements needed in many applications such as autonomous driving and real time image classification tasks. Even when putting aside the round-trip communication overhead some edge devices designed for space or deep ocean exploration may not have access to the internet, making training on servers impossible.

Because of the limitations described, Continual Learning or Incremental Learning can play a massive role in accommodating real time training of the models on new classes without compromising speed. These types of training methods attempt to train a model on only previously unseen data without destroying the prior knowledge acquired on different data.

This thesis attempts to accelerate one of the many continual learning algorithms, the Streaming Linear Discriminant Analysis (SLDA) algorithm, on an FPGA device. The aim is to perform training on new data with as little overhead as possible and as low energy consumption as possible. These are the two main constraints of a real time edge system.

1.2 Scientific Contributions

This thesis is a deep dive into the SLDA algorithm and how its inner calculations work so that it can be constructed and run efficiently on an FPGA

device, in this case the ZCU102. By researching the best ways to incorporate the whole process on a single FPGA device, it can benefit the user who is going to desire just connecting the classifier at the end of a classic inference model using a CPU, GPU or DPU. Furthermore it can benefit the whole device that is running on as there will be no constant communication between the host and the accelerator, leading to lower latency and energy efficient designs, that are critical especially in battery powered devices.

The overall design of the accelerator takes full advantage of the BRAMs and DSPs provided by the FPGA to achieve high levels of parallelism when faced with linear algebra operations, even when compared to contemporary CPUs and GPUs. The design of the architecture resembles more closely the design of the TPU rather than that of the two other platforms. The main way it can perform matrix and vector operations with such high speed is the really wide vectors that are loaded from memory and processed in parallel inside a grid of systolic arrays. The philosophy behind this design is to transfer data in and get data out, without a lot of control overhead and instruction fetching. This simplicity allows for the SLDA accelerator to soar to other domains that require fast linear algebra operations such as other Machine Learning algorithms or solving systems of linear equations fast (eg. weather prediction).

The whole design was made using the high level synthesis (HLS) tool provided by Xilinx and the testing and comparisons was made using a combination of optimized Python libraries, CUDA libraries and C++ libraries.

To sum up the contributions of this thesis in a list:

- An FPGA-based accelerator to be used in power critical systems as the energy efficiency is a lot better than the GPU or CPU counterparts
- The newly developed accelerator has substantially better performance than GPUs or CPUs when it comes to fixed point operations, namely $\sim 1000\times$ latency speedup
- The accelerator achieves $7466\times$ latency speedup for fixed point operations while the floating point operation latency speedup sits at $1.534\times$, with the baseline being NumPy. The energy efficiency for fixed point sits at $7900\times$ and for floating point at $106\times$
- Simple to use SLDA accelerator, exported as an IP core, that can be plugged in any system (edge or not) to perform continual learning.

- With a few tweaks, the user can obtain an accelerator that performs general matrix-matrix and matrix-vector multiplications, resembling an architecture similar to the TPU. The difference here, is that this accelerator is open source.

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** The second chapter dives into the history of Artificial Intelligence and how it evolved over the years from the simple perceptron to CNNs finally leading to the continual learning problem description.
- **Chapter 3 - Related Work:** This chapter explores the works of other researchers in Convolutional Neural Networks architecture, in dataset creation, in Continual Learning algorithms and ultimately in hardware architectures for Machine Learning and Matrix Multiplication.
- **Chapter 4 - Robustness Analysis:** In Robustness Analysis a deep look into the software algorithms and numerical operations is conducted. Furthermore, a thorough description of the custom hardware architecture of this thesis and the ways it can achieve good performance is done.
- **Chapter 5 - FPGA Implementation:** As the custom accelerator was specifically designed for FPGAs a plethora of tools and platforms were used to achieve the final result. This chapter describes these tools and platforms and what they have to offer. Additionally, the algorithms for the High Level Synthesis tool are presented.
- **Chapter 6 - Results:** This chapter contains an assessment of the accelerator compared to other CPU and GPU platforms, both in latency, throughput and in power, energy consumption. At the end, a discussion over the results is performed.
- **Chapter 7 - Conclusions and Future Work:** The final chapter sums up the contributions of this thesis and suggests some ways that can improve the performance and accessibility of the proposed IP core.

Chapter 2

Theoretical Background

In this chapter, a brief summary of the AI and ML advancements over the years is included for completeness. Furthermore, an introduction to the catastrophic forgetting problem, that is being assessed by continual learning, is written. Alongside it, a few continual learning methods are presented.

2.1 Artificial Intelligence

Artificial Intelligence is a relatively contemporary field in Engineering and Science that has emerged almost a century ago and attempts to imitate human level intelligence in all kinds of electronic devices.

This field studies the ways a machine can accomplish Problem Solving, Reasoning, Knowledge Representation, Planning and Decision Making using primarily the mathematical tools of Probability Theory, Logic Theory and Optimization under specific constraints.[1]

Aside from the theoretical and philosophical restlessness on the subject, that has its merits, AI finds applications in entertainment, like synthesising creative art forms (images, poems, text, speech etc.), and tackles real life problems such as pandemic control and prediction, medical condition diagnosis and traffic management, to name a few.

2.2 Machine Learning

Machine learning (or ML for short) is a subfield of Artificial Intelligence and it was first proposed from the AI researcher Arthur Samuel [2] as “the field of study that gives computers the ability to learn without explicitly being

programmed". To be more analytical, Machine Learning is the field that examines the ways a computer can analyze data to extract information and recognize patterns that may help it improve itself and make accurate predictions given a specific task. To achieve this the machine must reprogram some of its parameters employing algorithms that take as input large amounts of training data, receive insights, and make informed decisions.

The three predominant types of machine learning are Supervised Learning, Unsupervised Learning and Reinforcement Learning [1]. There is also Semisupervised Learning which encompasses the uncertainties that may occur in Supervised Learning (eg. some labels might be missing). More details about each type are given below.[3][4][5]

- **Supervised Learning:** This type of Machine Learning technique needs some knowledge about the training data so that it can map the inputs to the outputs. When provided with a new input, in testing, the agent will assign it an output (labeling) according to the previously acquired knowledge. Mathematically this can be expressed as:

Given a training set of N example input–output pairs

$$(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$$

, where each

$$y_j \in \mathbb{R}$$

was generated by an unknown function $y = f(x)$, and

$$x_j \in \mathbb{R}^N$$

, the input feature vector, discover a function h that approximates the true function f .

The main subcategories of Supervised Learning are Regression and Classification. Some algorithms that fall under these categories are Naive Bayes, Nearest Neighbor, Discriminant Analysis, Linear Regression, Support Vector Machines, Decision Trees, Ensemble Methods and Neural Networks [6].

- **Unsupervised Learning:** Learning in this way does not need any prior knowledge about the nature of the data and attempts to infer patterns

on the input. The main subcategory of Unsupervised Learning is Clustering which includes algorithms like K-means, Fuzzy C-means, Hierarchical, Gaussian Mixture and Hidden Markov Models.

- **Reinforcement Learning:** In Reinforcement Learning the AI agent learns through a series of rewards and penalties just as a child learns by using its senses and acting accordingly. Therefore, agents that learn through interaction and try to maximize a reward function use Reinforcement Learning. Algorithms that fall under this category are Q-Learning, SARSA, Deep Q Network (DQN), Deep Deterministic Policy Gradient (DDPG), Trust Region Policy Optimization (TRPO) [7] [1].

2.3 Deep Learning

Deep Learning is a subset of Machine Learning that takes advantage of the capabilities of Artificial Neural Networks (ANNs) that have three or more layers, that means apart from the input and output layers there exists at least one hidden layer. ANNs, or simply Neural Networks (NNs), are inspired by the biological brain of animals and tries to imitate its function by introducing a network of neurons that is fully connected. Each connection of the network is called an *edge* of the network. [8][9][10]

Initially, for a complete comprehension of how Neural Networks function, it is imperative to describe the behavior and anatomy of a biological neuron (nerve cell). As shown in **figure 3.1**, anatomically a neuron mainly consists of dendrites, the soma, the axon and synaptic terminals. The artificial neuron receives multiple inputs, from edges, and sums them up, just like biological dendrites, to produce an output (or activation, representing a neuron's action potential which is transmitted along its axon). [11]

2.3.1 Perceptron

A perceptron, in the context of Machine Learning, is a binary algorithm that takes an input vector x and produces an output $f(x)$ that takes values in the binary set $S = \{0, 1\}$. The term was first coined by Warren McCulloch and Walter Pitts in 1943 [12] and thus the perceptron is also called the McCulloch-Pitts neuron. Each one of the outputs represents a class, so the algorithm learns a binary classifier called a *threshold function* or *activation function*. This

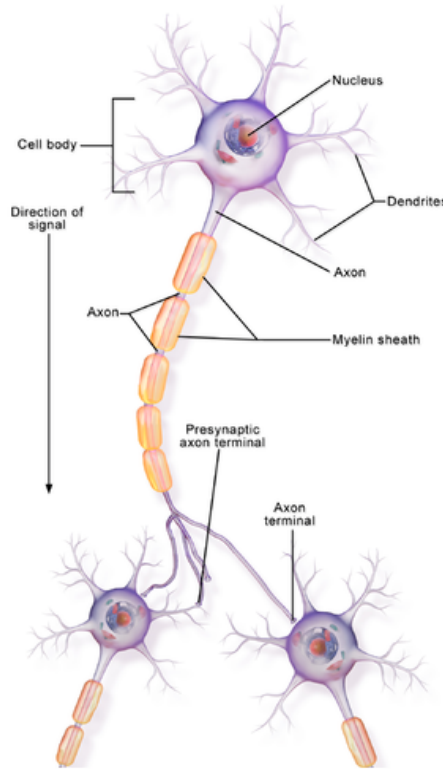


FIGURE 2.1: Anatomy of a Neuron: <https://www.physio-pedia.com/Neurone>.

function takes as input a vector of real values and produces a vector of binary values at the output[13][14][15]. Mathematically, this can be expressed as:

Assume $F(x)$ is the activation function and $N = \text{datadimensionality}$. Then the output of a perceptron $f(x)$ can be described as:

$$f(x) = F(w \cdot x + b)$$

,

where $w \cdot x$ is the inner product of the N -dimensional vector $x \in \mathbb{R}^N$ with an N -dimensional weight vector $w \in \mathbb{R}^N$ and $b \in \mathbb{R}$ is a bias that controls the position of the decision boundary and is input independent. A graphical representation is shown in figure 3.2.

2.3.2 Activation Function

An Activation or Transfer function's nature is to determine whether or not a neuron is activated and should be taken into consideration when calculating

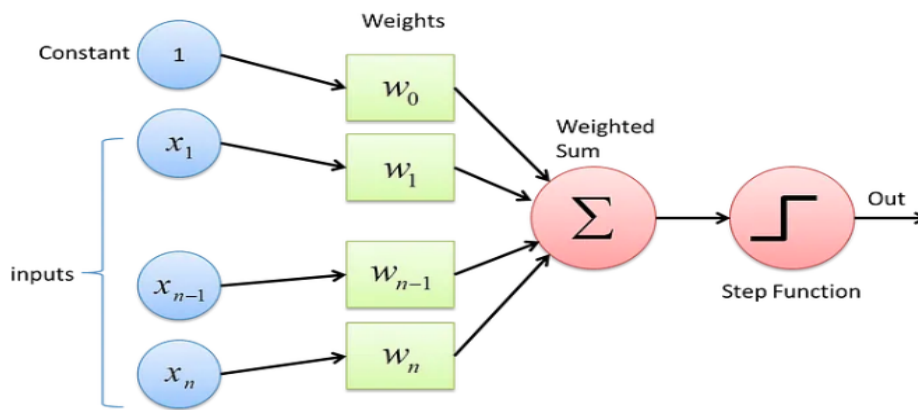


FIGURE 2.2: The Perceptron:
<https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53>.

the next outputs. This behavior simulates the way a brain cell decides if it should fire, based on some threshold.

Although, the linear perceptron, that was mentioned in the previous section can make predictions in simple environments for a larger Neural Network with more complex tasks at hand, it is not ideal. If the output of every neuron in every layer is a linear function then the composite of these functions is also a linear function. Therefore, the model is reduced to a linear regression model. This linearity proves to be insufficient in problems with more than two classes. Consequently, the activation function is used to add non-linearity to the output of the network. [16][17]

- **Binary Step Function:** The Binary Step function assigns outputs, that belong in the binary set $S = \{0, 1\}$, according to some threshold. If the input surpasses the threshold then it is assigned an 1, alternatively it is assigned a 0. The mathematical representation is:

$$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

- **Linear Activation Function:** The Linear Activation Function is considered the identity function in Neural Networks, as it just passes the input to the output. The mathematical representation is:

$$f(x) = x, x \in \mathbb{R}$$

Both of the afore mentioned functions are only useful in linear regression problems.

- **Sigmoid or Logistic Function:** The Sigmoid Function is a continuous function that takes values in the open interval $(0, 1)$. Inputs that are larger than 0 have an output closer to 1, while inputs that are less than 0 have an output closer to 0. The mathematical representation is:

$$f(x) = \frac{1}{1 + e^{-x}}, x \in \mathbb{R}$$

- **Tanh or Hyperbolic Tangent:** Similar to the Sigmoid function with the sole difference being that it receives values in the open interval $(-1, 1)$. The mathematical representation is:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, x \in \mathbb{R}$$

- **RELU:** It combines the Binary Step function with the Linear function, rendering itself non linear. The mathematical representation is:

$$f(x) = \max(0, x), x \in \mathbb{R}$$

- **Softmax:** It can be considered as a mixture of multiple Sigmoids. The SoftMax function returns the probability of each class.

It is most commonly used as an activation function for the last layer of the neural network in the case of multi-class classification. The mathematical representation is:

$$\text{sigmoid}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}, x \in \mathbb{R}$$




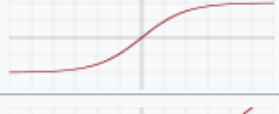

Identity		x
Binary step		$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$
Logistic, sigmoid, or soft step		$\sigma(x) \doteq \frac{1}{1 + e^{-x}}$
Hyperbolic tangent (tanh)		$\tanh(x) \doteq \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Rectified linear unit (ReLU) ^[8]		$(x)^+ \doteq \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ $= \max(0, x) = x \mathbf{1}_{x>0}$

FIGURE 2.3: Plots of the Activation Functions that were described: https://en.wikipedia.org/wiki/Activation_function.

2.4 Neural Network Training Steps

2.4.1 Feedforward Neural Network

A Feedforward Neural Network is the simplest form of a Neural Network, where there are no cycles between nodes and data flows from the input layer to the output layer. The most stripped-down Feedforward NN is the single layer perceptron, which was described in detail in section 3.3.1. A more complex but more functional form of Feedforward NNs is the multilayer perceptron, which is illustrated and detailed below.

Forward Propagation

Following the entry of the raw data through the input layer, each node produces an output using the perceptron algorithm and applying an activation function to it. Going forward, all the outputs are propagated to the next hidden layer. Each neuron in one layer has directed connections to the neurons of the subsequent layer. This process is repeated until the data reach the output layer and a prediction is made.^{[18][19]}

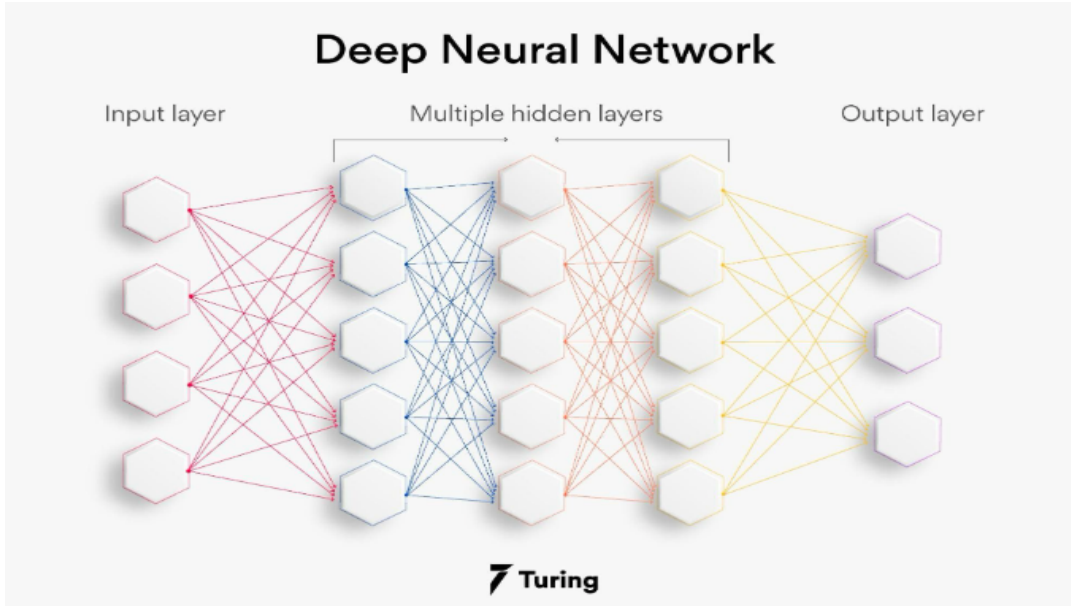


FIGURE 2.4: Architecture of a simple feedforward Neural Network: <https://www.turing.com/kb/mathematical-formulation-of-feed-forward-neural-network>.

2.4.2 Loss Function

Loss Functions constitute an integral part in how a Neural Network learns how the input data correlate with the output data. In layman's terms, it learns from its mistakes, just like any other living organism.

In Machine Learning, the loss function measures how accurate the predicted value is in comparison to the target value. In all cases, for a deep learning model to perform with adequate accuracy the loss function should be minimized. Therefore, an optimization problem is introduced, with the variables that can be tweaked being the weights and the biases of the network. Mathematically this problem can be expressed as:

Minimize $J(w^T, b)$, where

$$J(w^T, b) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)})$$

Furthermore, according to the use case of the model different loss functions are employed[20]. Supervised Learning models more commonly take advantage of loss functions, without it excluding Unsupervised Learning models that are gaining ground in recent years, most notably in image and text generation. A list of problems and the respective loss functions that are used are shown below.

- Regression Problems:

- Mean Square Error (MSE):

$$MSE = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2$$

- Mean Absolute Error (MAE):

$$MAE = \frac{1}{m} \sum_{i=1}^m |\hat{y}^{(i)} - y^{(i)}|$$

- Binary Classification Problems:

- Binary Cross Entropy/ Log Loss:

$$LL = -(y_i \log(\hat{y}_i)) + (1 - y_i)(\log(1 - \hat{y}_i))$$

- Multi-class Classification Problems:

- Categorical Cross Entropy Loss:

$$CELoss = -\frac{1}{n} \sum_{i=1}^N \sum_{j=1}^M y_{ij} \log(p_{ij})$$

2.4.3 Backpropagation and Gradient Descent

A method that is quintessential in making Neural Networks learn, is *Backpropagation*. It was popularized by DE Rumelhart, GE Hinton, RJ Williams in their paper "*Learning Representations by backpropagating errors*" [21]. Fundamentally, this method tries to find a local minimum in the loss function by adjusting the weights and biases of the network. The direction of this adjustment (increase or decrease) in each individual weight and bias is determined by the gradients of the loss function. These gradients are calculated using the Leibniz chain rule.[22][23][24][25][26]

Generally, in multivariate calculus, the negative gradient of a function shows the direction of steepest descent. This theorem is used by the most popular method in finding local minima during backpropagation, called *Gradient Descent*.

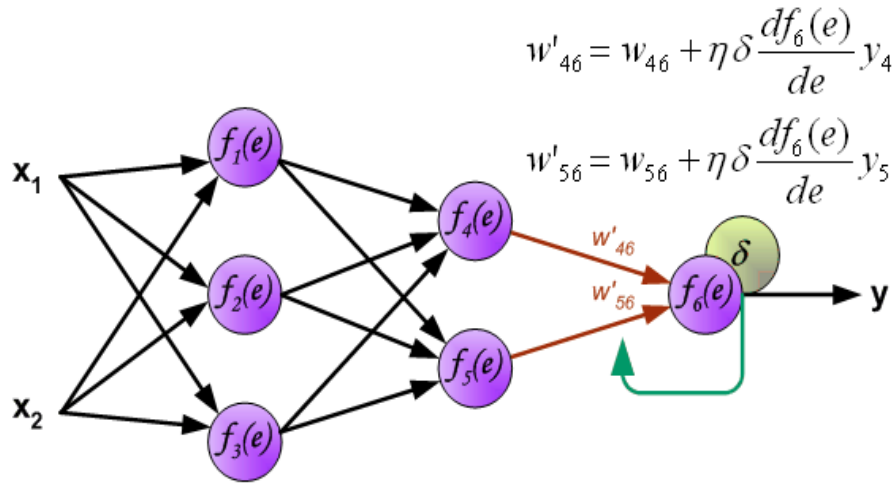


FIGURE 2.5: The Backpropagation algorithm illustrated: http://galaxy.agh.edu.pl/vlsi/AI/backp_t_en/backprop.html.

2.5 Convolutional Neural Networks

Previously, the simple Feedforward Neural Network was discussed, and even though it still finds applications in some simple tasks, it has become obsolete in more recent years due to advancements in other network architectures and more specifically the Convolutional Neural Network (or CNN). The CNN took its first steps as an idea that stemmed from the paper "*Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position*" [27] by Kunihiko Fukushima. Later, the first actual CNN was introduced in 1998, by Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner's "*Gradient-based learning applied to document recognition*" [28] in the form of the LeNet-5. Since then, many other CNN architectures have sprouted and produced excellent accuracy in image recognition tasks.

CNNs find applications in a number of real life problems, such as face recognition, autonomous driving, medical image analysis, time series classification and many more [29].

Diving deeper, CNNs are deep Artificial Neural Networks that do not rely on the user to apply preprocessing to the data, instead it finds useful patterns itself and extracts them through image manipulation techniques like

convolutions (hence the name) and subsampling.

Moreover, after a certain amount of convolution + subsampling layers the data have become one-dimensional and can be fed to a fully connected multilayer perceptron. The classification process is performed there through the means that were detailed in previous chapters.

An image of the first CNN ever engineered, the LeNet-5 is shown below

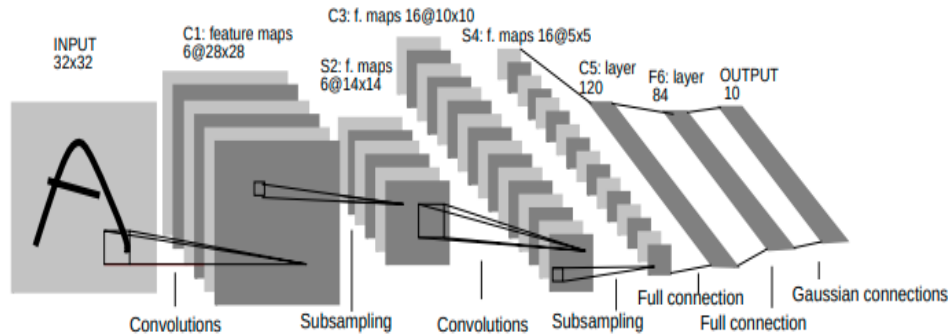


FIGURE 2.6: LeNet-5 Architecture: [28].

It is important to mention, that the input data are two-dimensional in this example but generally speaking the input of a CNN is a tensor with dimensions $H \times W \times D$, where H = Height of the image in pixels, W = Width of the image in pixels, D = Depth of the image, or the number of channels (eg. 3 channels for RGB images). In many APIs such as Tensorflow and Pytorch the number of images in a batch is considered the fourth dimension of the tensor.

2.5.1 Convolutional Layers

Convolution is a mathematical operation that can be performed in N dimensions but this thesis focuses only on 1D and 2D data. It is basically the sliding dot product between a kernel g and the data f . The two functions must be in the same dimension.

Convolution on 1D data is typically needed in time series manipulation. The kernels in this example are windows of certain width that slide through the dimension of time and produce features that are also time series. This mathematical operation is expressed as:

Assume

$$g[n] = [g_1, g_2, \dots, g_N]$$

$$f[n] = [f_1, f_2, \dots, f_M]$$

, then

$$(f * g)[n] = \sum_{i=1}^N f[i] \cdot g[n - i]$$

When trying to implement this operation in a programming language, the user should always remember to zero pad the input function f so as to include the corner cases of convolution, and produce a result with $M + N - 1$ elements. The exact same procedure is followed in the case of image processing, which will be described later.

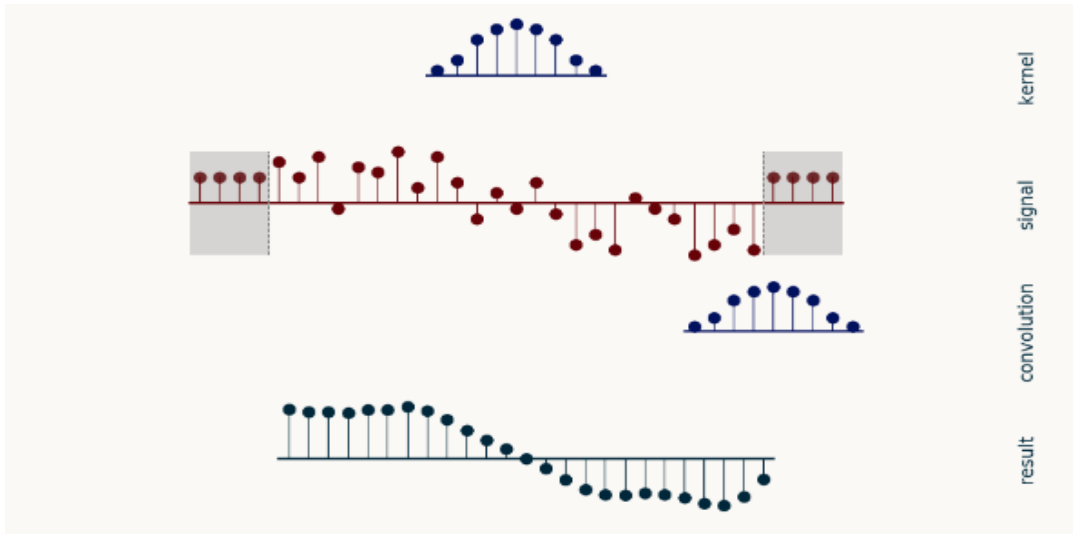


FIGURE 2.7: The convolution operation in the time domain:

https://e2eml.school/convolution_one_d.html.

Furthermore, in applications that have 2D images as input data, the 2D convolution is applicable. At its core it follows the exact same principles as the 1D case, apart from the fact that the functions that take part in the operation receive inputs from two-dimensional space. That means that the kernels are usually square windows with a *WIDTH* and a *HEIGHT*. These two metrics make up the *SIZE* of the window. Another critical value that must be defined in this case is the *STRIDE* which dictates how many pixels to skip in the next iteration, both horizontally and vertically. The mathematics of the 2D convolution are given below:

Assume

$$g[3,3] = \begin{bmatrix} g_{11} & g_{12} & g_{13} \\ g_{21} & g_{22} & g_{23} \\ g_{31} & g_{32} & g_{33} \end{bmatrix}$$

is the kernel and

$$f[h, w] = \begin{bmatrix} f_{11} & f_{12} & \dots & f_{1W} \\ f_{21} & f_{22} & \dots & f_{2W} \\ \vdots & \vdots & \dots & \vdots \\ f_{H1} & f_{H2} & \dots & f_{HW} \end{bmatrix}$$

is the input image, then the convolution of these two can be computed as

$$(f * g)[h, w] = \sum_{i=-1}^1 \sum_{j=-1}^1 g[i, j] \cdot f[h - i, w - j]$$

If the desired output has to have the same dimension as the input image, then zero padding is imminent. The zeros that are added in all four sides of the image are calculated as follows, assuming a square kernel:

$$\#zeros = \left\lfloor \frac{KernelWidth}{2} \right\rfloor$$

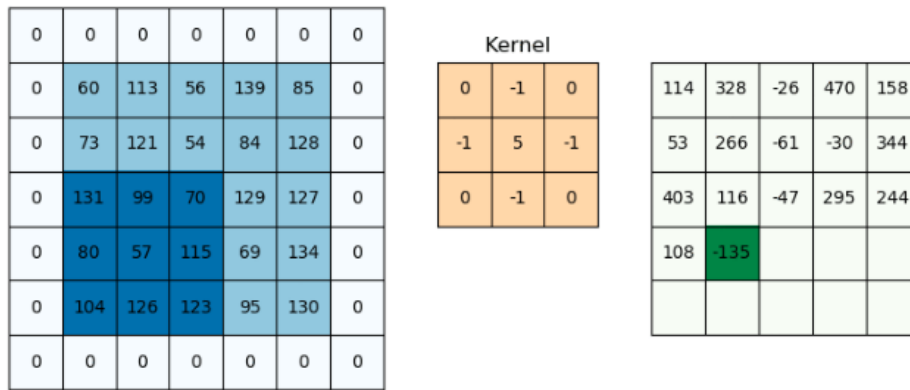


FIGURE 2.8: Convolution of an image with a 3x3 kernel: <https://medium.com/@draj0718/zero-padding-in-convolutional-neural-networks-bf1410438e99>.

2.5.2 Subsampling Layers

Convolutional Neural Networks (CNNs) employ the subsampling approach to shrink the spatial dimensions of the input feature maps. The model becomes less computationally difficult as a result, and it also becomes more

resilient to changes in the input. These layers more often than not proceed the Convolutional Layers. In CNNs, two popular subsampling techniques are used:

- **Max Pooling:** In max pooling, the maximum value in a rectangular neighborhood of the input feature map is selected and used as the output value. The neighborhood is defined by the size of the pooling window and the stride of the pooling operation. Max pooling is the most common subsampling method in CNNs, as it has been found to work well in practice.

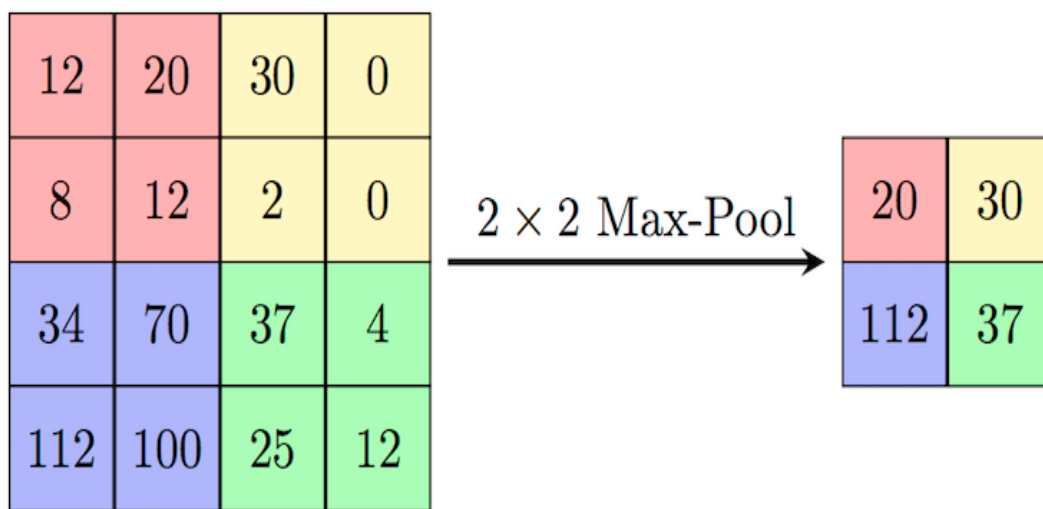


FIGURE 2.9: Max pooling on a 4x4 image, applying a 2x2 pooling window: <https://paperswithcode.com/method/max-pooling>.

- **Average Pooling:** In average pooling, the average value in a rectangular neighborhood of the input feature map is computed and used as the output value. Like max pooling, the neighborhood is defined by the size of the pooling window and the stride of the pooling operation. Average pooling is less commonly used than max pooling, but it can be useful in some cases where the input features have a more uniform distribution.

Other pooling processes, such as L2 pooling, stochastic pooling, and fractional pooling, exist in addition to these two popular subsampling techniques. Although these techniques are less frequently utilized in actual practice, they can be helpful in some particular applications.

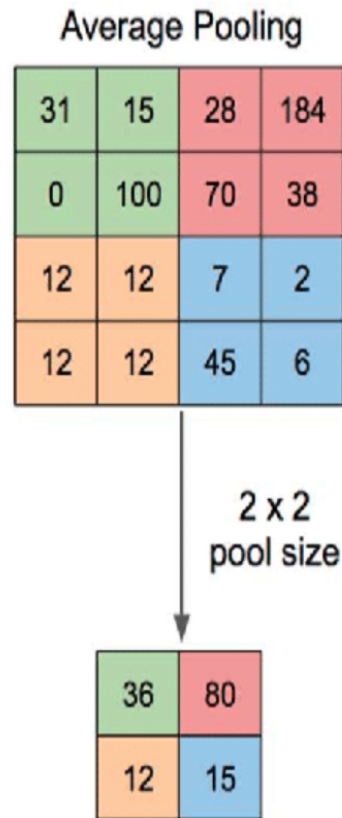


FIGURE 2.10: Average pooling on a 4x4 image, applying a 2x2 pooling window:
<https://paperswithcode.com/method/average-pooling>.

2.6 Continual Learning

All of the previous sections were introductory to the way Artificial Neural Networks operate and which architectures are suitable for each job. Nonetheless, the main topic of this thesis revolves around *Continual Learning*, or *Life-long Learning*. Continual Learning studies the ability of Machine Learning algorithms to learn patterns from non-stationary data and without forgetting previously learned patterns. The last phenomenon is called *Catastrophic Forgetting* [30] and it is the primary reason why building Continual Learning models is not a trivial task.

Catastrophic forgetting stems from the fact that when training on a new task the dataset may have a different distribution from the original data the model was trained on. That means, the weights and biases, that are updated according to the new dataset, fail to produce an adequate prediction on some previously learned classes. The "*Stability- Plasticity dilemma*" examines this exact phenomenon[31]. When a Neural Network focuses only on stability

then the new data, that is used for training, is not coded correctly in the Network, whereas focus on plasticity can impede on the already encoded data, from previous training sessions.

Many strategies for addressing catastrophic forgetting have been proposed. Elastic Weight Consolidation (EWC), one regularization strategy, prevents the network's weights from drifting too far from their initial values. Replay techniques, such generative replay and experience replay, keep track of past data and use it to train the network for a new task. To prevent cross-task interference, parameter isolation techniques like Learning without Forgetting and Gradient Episodic Memory separate the network's parameters for each task.[32][33]. Some of these techniques will be presented in detail in the next chapter.

2.7 Continual Learning algorithms that assess the Catastrophic Forgetting problem

2.7.1 Elastic Weight Consolidation (EWC)

In the research paper "Overcoming catastrophic forgetting in neural networks" James Kirkpatrick et al. [32] set to prevent catastrophic forgetting in deep Neural Networks. By examining the human brain, they found that the synapses associated with previously learned tasks, on a critical level, tend to show decreased levels of plasticity. This is called synaptic consolidation. Elastic Weight Consolidation (or EWC) attempts to mimic this behaviour in ANNs, by constraining important parameters to be altered less easily.

A new task is learned in ANNs by adjusting the weights and biases θ of the network. It is significant to mention, that many configurations of θ result in the same performance. EWC tries to find a configuration θ_B^* , for task B, that still falls in the low error region for the previous task A, with parameters θ_A^* . That means that EWC protects task A by restraining the parameters critical to its performance. This constraint is implemented as a quadratic penalty, and can therefore be imagined as a spring anchoring the parameters to the previous solution, hence the name elastic.

As seen in 2.11, restricting all the parameters equally (green arrow) does not allow the model to learn task B. Not restricting them at all (blue arrow) allows the learning of task B to the detriment of task A. EWC is the red arrow.

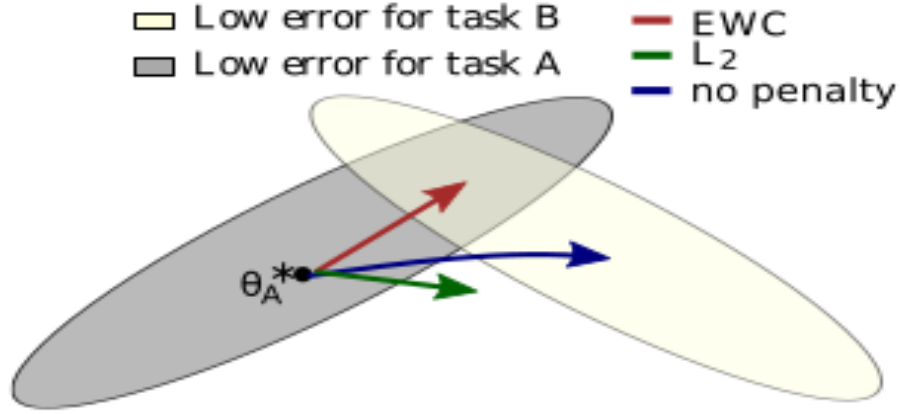


FIGURE 2.11: The low error regions of the configurations for tasks A and B, with their intersection being the region where both of those tasks do not lose performance:
<https://arxiv.org/abs/1612.00796>.

2.7.2 Learning without Forgetting (LwF)

Learning without Forgetting is a continual learning algorithm, that aims to achieve comparable results between performance on old tasks and new tasks, when receiving as input only data from the new task. Zhizhong Li and Derek Hoiem's work [34] is related to networks that transfer knowledge from one network to the other, such as knowledge distillation methods. With each new task, new parameters θ_n are added to the last classification layer and are randomly initialized. The new nodes represent the classes of the new task. Lastly, the network is trained to minimize loss for all tasks.

2.7.3 Gradient Episodic Memory (GEM)

First introduced by David Lopez-Paz and Marc'Aurelio Ranzato [35], Gradient Episodic Memory (or GEM) is a continual learning model that tries to learn tasks that arrive in an episodic manner. The authors, unlike their contemporary counterparts, focus on solving a more "human-like" problem where i) the number of tasks is large, ii) the number of training examples per task is small, iii) the learner observes the examples concerning each task only once, and iv) reported metrics measure both transfer and forgetting. GEM uses an episodic memory M_t which stores a subset of the observed examples task t .

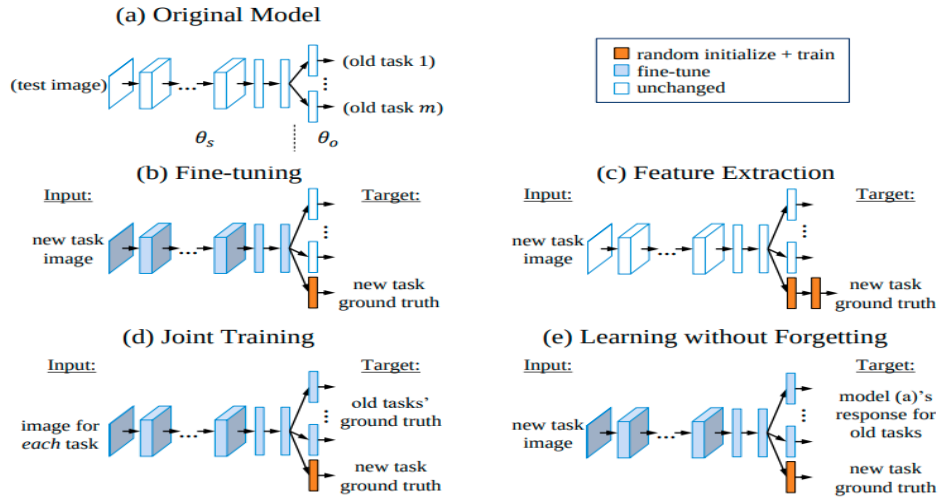


FIGURE 2.12: Graphical representation of different methods for learning θ_n with the assistance of previously learned parameters θ_s : <https://arxiv.org/abs/1606.09282>.

2.7.4 Deep Streaming Linear Discriminant Analysis (Deep SLDA)

Originally termed Incremental Linear Discriminant Analysis (ILDA), this method has risen to the surface by Shaoning Pang, Seiichi Ozawa and Nikola Kasabov [36] to alleviate the problem of streaming data classification, something vanilla LDA and PCA do not solve. This method derives a new discriminant eigenspace, when fed with sequential chunks of data, that are random in size and contain new classes, then adds it to the existing discriminant eigenspace. The end results show promising conservation of the classification accuracy.

Inheriting the aforementioned idea, Tyler L. Hayes and Christopher Kanan [37] were the first to use deep SLDA for feature classification of deep CNNs with an emphasis on large scale image classification datasets. This streaming learning method is ideal for edge devices as it is neither memory nor computationally intensive.

To get into more detail on how this method works, it is assumed that a CNN can be described as a function composition $y_t = F(G(X_t))$, where X_t is the input image. In this case, $G(\cdot)$ describes the first J layers of the network whereas $F(\cdot)$ describes the final fully connected layer, with parameters θ_G and θ_F , respectively. $G(\cdot)$ is trained during base initialization on a fraction of the dataset while $F(\cdot)$ is trained in a streaming manner using the remainder

of the dataset. The authors adapted SLDA to train a linear decoder

$$F(G(X_t)) = Wz_t + b$$

where $z_t = G(X_t) \in \mathbb{R}^d$, $W \in \mathbb{R}^{K \times d}$ is the weight matrix and $b \in \mathbb{R}^K$ is the bias vector. K is the number of classes and d is the dimensionality of the data.

The mean for each class is a vector $\mu_k \in \mathbb{R}^d$, that is stored alongside the count $c_k \in \mathbb{R}$ and the shared covariance matrix $\Sigma \in \mathbb{R}^{d \times d}$. Each class mean vector is updated whenever a new sample passes through the network. Assume the sample is (z_t, y) , that belongs in class y , then the new mean vector and associated count are updated as follows:

$$\mu_{(k=y,t+1)} \leftarrow \frac{c_{(k=y,t)}\mu_{(k=y,t)} + z_t}{c_{(k=y,t)} + 1}$$

$$c_{(k=y,t+1)} = c_{(k=y,t)} + 1$$

For Plastic SLDA the covariance matrix is also updated as follows:

$$\Sigma_{t+1} = \frac{t\Sigma_t + \Delta_t}{t + 1}$$

where Δ_t is:

$$\Delta_t = \frac{t(z_t - \mu_{(k=y,t)})(z_t - \mu_{(k=y,t)})^T}{t + 1}$$

On the contrary Static SLDA uses a pre-initialized covariance matrix, without updating it.

The authors also use shrinkage regularization to compute the precision matrix $\Lambda = [(1 - \epsilon)\Sigma + \epsilon I]^{-1}$. The rows of the weight matrix W are then computed as:

$$w_k = \Lambda\mu_k$$

,and the individual elements of the bias vector b are computed as:

$$b_k = -\frac{1}{2}(\mu_k \cdot \Lambda\mu_k)$$

Chapter 3

Related Work

In the following chapter a number of CNN architectures and datasets that can be used in continual learning applications are presented for completeness. In this thesis, the CNN that was ultimately used is the ResNet18 alongside the CIFAR-10 dataset. Furthermore, attempts at implementing some of the continual learning algorithms in chapter 2 on FPGA platforms are shown. Finally, some works related to how certain hardware architectures handle the linear algebra calculations needed in continual learning, and Machine Learning in general, are discussed.

3.1 CNN architectures

We will talk about, CNN architectures, Continual Learning and hardware architectures.

3.1.1 AlexNet

The Alexnet was first introduced by Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton [38] and won the 2012 Imagenet Competition by achieving top-1 and top-5 error rates of 37.5% and 17.0% respectively. These results considerably outpaced the competition and thus became one of the most prevalent CNN architectures of its time. The architecture is comprised of three Convolutional+Max Pooling layers, two Convolutional layers and three Fully Connected Layers.

Some of the techniques that made AlexNet so accurate were:

- ReLU Nonlinearity and Local Response Normalization
- Multiple GPUs that processed data in parallel

- Overlapping Pooling
- Data Augmentation, for an increase in the training dataset
- Dropout of Neurons

The AlexNet does not come without its drawback, in the form of computational workload. Each image during the feedforward pass through the network reaches a billion computations. Figure 4.1 illustrates the architecture of AlexNet.[39]

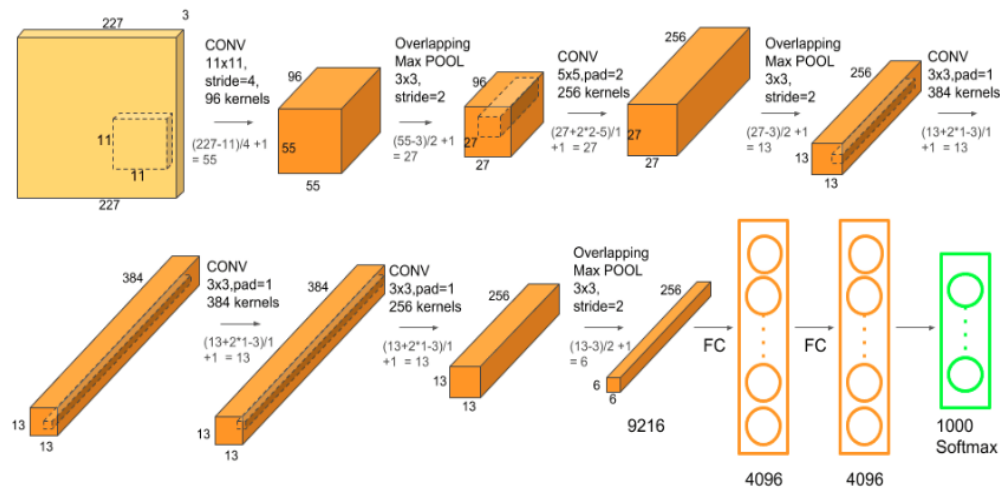


FIGURE 3.1: AlexNet architecture:
<https://www.kaggle.com/code/blurredmachine/alexnet-architecture-a-complete-guide/notebook>.

3.1.2 VGG

VGG, or VGGNet as it is commonly referred as, is a deep CNN architecture that excelled at the ILSVRC competition in 2014, and since then it is one of the pioneering models for image recognition. VGG, which stands for Visual Geometric Group, supports two versions; VGG16 which is comprised of 16 Convolutional layers, and VGG19 which consists of 19 Convolutional layers. It was first proposed by A. Zisserman and K. Simonyan in their paper “Very Deep Convolutional Networks for Large-Scale Image Recognition.”[40].

The VGGNet receives as input colored images with dimensions 224x224. The convolutions are performed using small kernels of size 3x3 and sequentially applying ReLU activation function to add non-linearity. The large number of small kernels increases the non-linearity of the model while the avoidance

of Local Response Normalization decreases memory consumption and training speed. On the other hand, one major problem of this architecture is the *Vanishing Gradient* phenomenon, due to the large depth.[41]

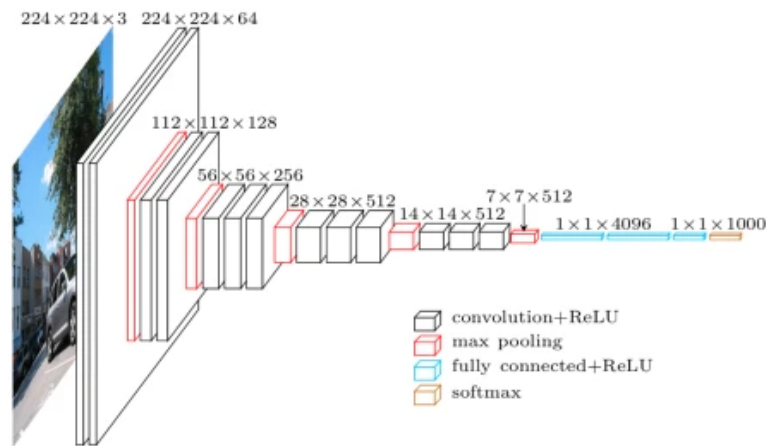


FIGURE 3.2: VGGNet architecture: <https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/>.

3.1.3 ResNet

Deep Residual Neural Networks, or ResNet for short, are the next novel architectures for CNNs, attempting to surpass VGGNet and ultimately succeeding. First introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun in their study “Deep Residual Learning for Image Recognition” [42] that won the ILSVRC COCO 2015 competition.

It addresses the problem of Vanishing Gradient by introducing a type of block, called residual block, that allows connections between layers to be skipped. Furthermore, this assures that the performance of a layer does not show signs of declining in comparison to previous layers. This technique, allows for a network depth of up to 152 layers - 8x the layers of a VGGNet. In addition, this type of CNN will be primarily used during the experiments of this thesis.[43][44]

3.1.4 DenseNet

In their research paper “Densely Connected Convolutional Networks”, Gao Huang, Zhuang Liu, Laurens van der Maaten and Kilian Q. Weinberger [45] observed that shorter connections between layers that are in close proximity

to the input and layers that are near the output, can deduce more accuracy, depth and training efficiency in the network. Following this observation, the DenseNet was born, a network where each layer is connected to all the subsequent layers, in a feedforward fashion. As a result, the connections between layers sum up to $\frac{L(L+1)}{2}$, whereas in previous architectures this sum was equal to the number of layers L . Moreover, DenseNets offer a variety of appealing benefits, including the elimination of the vanishing-gradient issue, improved feature propagation, promoted feature reuse, and significantly fewer parameters.

3.1.5 MobileNets

The MobileNets, produced by researchers at Google Inc. [46], have as their primary purpose not to redefine CNN architecture but to adapt them in such a way so that image classification becomes doable in mobile and edge devices. This is achieved, through techniques that decrease the number of parameters and the number of MAC calculations[47]. MobileNets employ performance enhancing procedures such as:

- Depth-wise and Point-wise convolution: Both of these techniques reduce the computational cost of plain convolution.
- Width multiplier: Further reduction in computational cost through model thinning.
- Resolution multiplier: On top of the previous techniques, this one reduces computational cost by means of reduced representation.

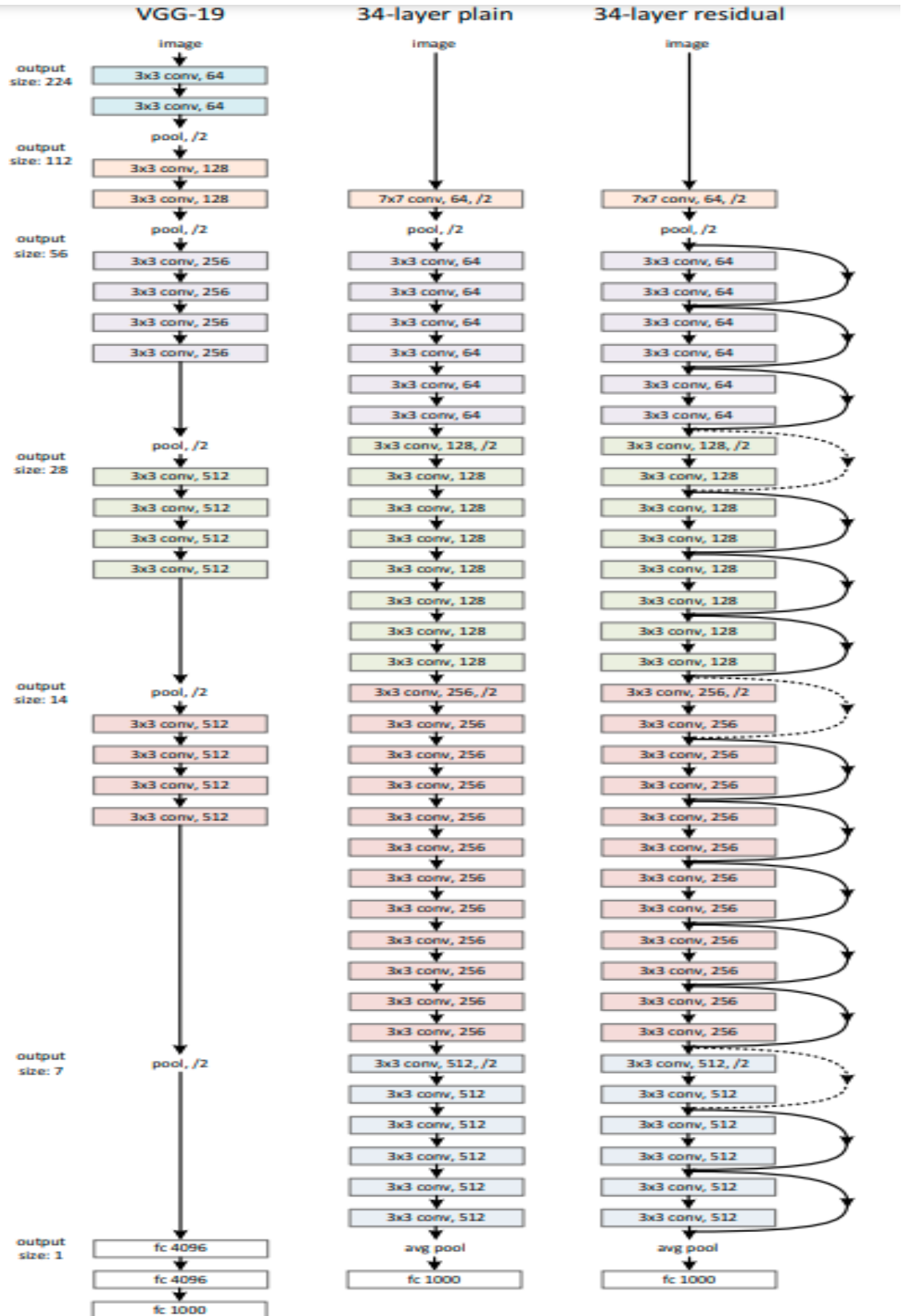


FIGURE 3.3: The ResNet architecture. By introducing residual blocks (curved arrows) the vanishing gradient problem is controlled and the per layer performance is not degraded:

<https://arxiv.org/pdf/1512.03385.pdf>.

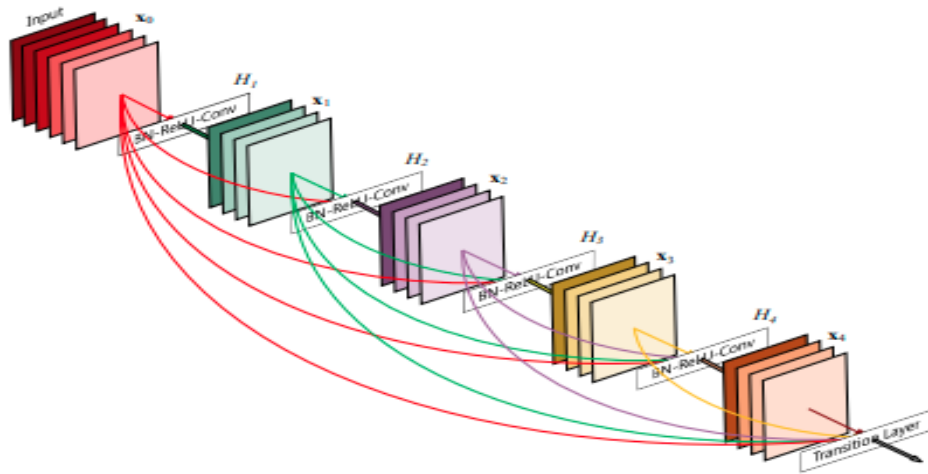
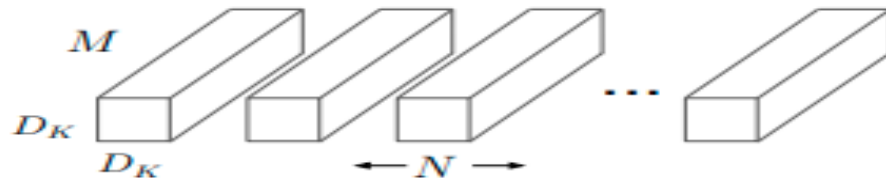
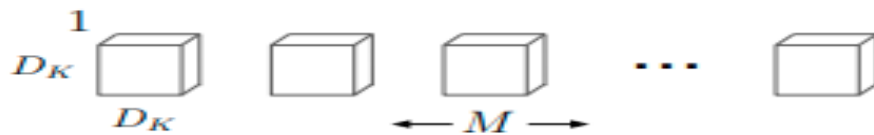


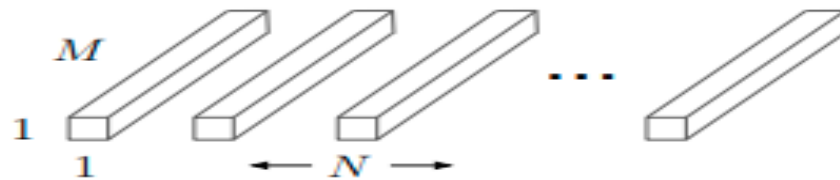
FIGURE 3.4: The DenseNet architecture. In this architecture each layer is connected to all the subsequent layers:
<https://arxiv.org/pdf/1608.06993.pdf>.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

FIGURE 3.5: MobileNets depth-wise convolution and pointwise convolution. These techniques enable the MobileNets to reduce the computational cost and run on edge and mobile devices: <https://arxiv.org/pdf/1704.04861.pdf>.

3.2 Datasets previously used in Continual Learning Tasks

3.2.1 CIFAR-10/100

The CIFAR-10 dataset was composed by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. It contains 60000 images with size 32x32. It is a subset of the 80 million tiny images dataset. The dataset is divided into 10 classes with 6000 images per class. There are 50000 images split into five training batches and 10000 images for testing.

The CIFAR-100 dataset has the exact same number of images, that means 60000 but divided into 100 classes. As a result each class has a 10x less images in comparison to the CIFAR-10. This makes classification of these images more challenging. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs).

Both datasets can be found at <https://www.cs.toronto.edu/~kriz/cifar.html>.

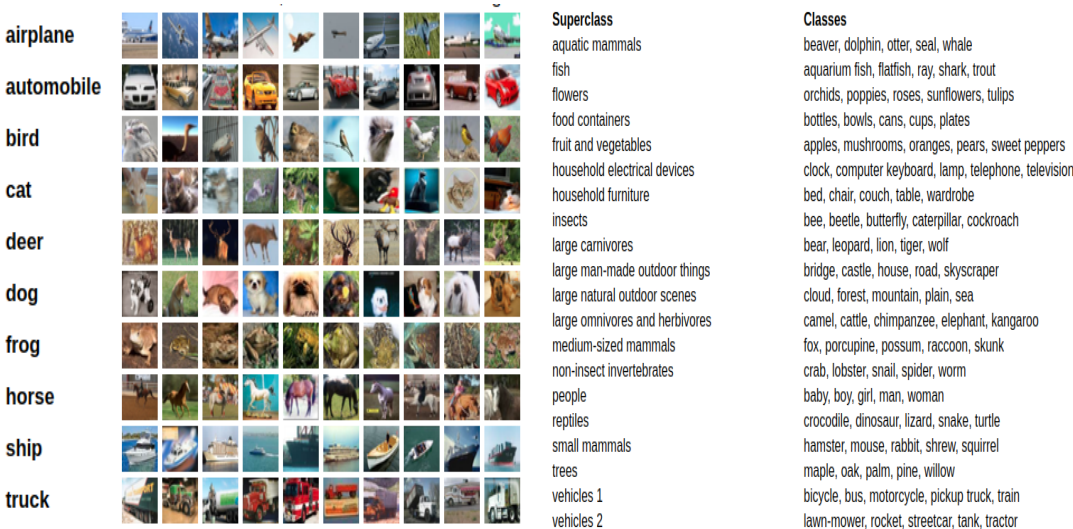


FIGURE 3.6:
CIFAR-10:
<https://www.cs.toronto.edu/~kriz/cifar.html>.

FIGURE 3.7:
CIFAR-100:
<https://www.cs.toronto.edu/~kriz/cifar.html>.

3.2.2 CoRE50

The CoRE50 dataset was first introduced by Vincenzo Lomonaco Davide Maltoni [48] as a dataset that is specifically designed for Continual Learning Object Recognition tasks. It is comprised of 50 domestic objects that fall under 10 distinct categories. The ten categories are: plug adapters, mobile phones, scissors, light bulbs, cans, glasses, balls, markers, cups and remote controls. Each object in each category is captured in video in 11 different sessions. That means that there are 300 RGB-D frames (20fpsx15seconds) for every item in a session.

Furthermore, three different scenarios are taken into consideration when attempting benchmarking with this dataset:

- New Instances (NI): New training patterns of the same classes become available in subsequent batches
- New Classes (NC): New training patterns from different classes become available in subsequent batches
- New Instances and Classes (NIC): New training patterns belonging both to known and new classes becomes available in subsequent training batches



FIGURE 3.8: Poses of the 50 different objects of the CoRE50:
<https://vlomonaco.github.io/core50/>.

3.2.3 ImageNet

ImageNet is a very large dataset containing more than 20000 categories with a total of approximately 14 million images. The most highly-used subset of ImageNet is the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012-2017 image classification and localization dataset. This dataset spans 1000 object classes and contains 1,281,167 training images, 50,000 validation images and 100,000 test images [49].

3.2.4 CUB200-2011

The Caltech-UCSD Birds-200-2011 dataset(CUB200-2011) is a challenging dataset consisting of 200 species of birds, with double the number of images per class compared to CUB200. It also provides part location annotation. The total number of images in the dataset is 11788[50].

3.3 Thesis Approach

This thesis main approach is to adapt the previously mentioned SLDA algorithm on an FPGA device. This will lead to an accelerator that will not only perform inference in real time but also train the linear classifier at the same time. Moreover, a thorough examination of the behaviour of the accelerator, and consequently the nature of the algorithm, when faced with low precision data is carried out.

The SLDA algorithm, along with a plethora of other ML techniques and algorithms, relies heavily on Linear Algebra. This of course means that the employed hardware must have the capabilities to perform vector operations (addition, subtraction, scaling), dot products and matrix multiplications, in an efficient manner.

These operations demand a high degree of parallelism, which cannot be achieved with CPUs, even modern ones, due to their sequential nature. On the other hand, GPUs and FPGAs are made to produce results with substantial throughput and low latency. Moving forward, some common designs for matrix multiplication on the aforementioned platforms will be discussed.

3.4 Hardware designs for Matrix-Vector multiplication

3.4.1 GPU based designs

Graphical Processing Units (GPUs), as implied by their name, were focused on accelerating graphics rendering algorithms, most often utilized by the video game industry but not limited to it. With the evolution of the capabilities of the GPU when faced with a naturally parallelizable algorithm, came the realization that other non-graphical applications may put use to them.

General Matrix Multiplications (GEMMs) on GPUs are performed using a tiling method. To elaborate further, both the input matrices (A and B) and the output matrix (C) are divided into tiles, that are then assigned to thread blocks. For every output tile, the corresponding inputs from A and B are loaded from memory and for every individual element of C the row of A and the column of B are multiplied and accumulated. This achieves high levels of parallelism as all the tiles in C are generated in parallel[51]. An illustration of the above method is depicted in 3.9.

3.4.2 FPGA based designs

In recent years, a lot of research has been carried out on accelerating Matrix Multiplication on FPGAs [sources]. Most importantly though, in the context of this thesis, the work of Duvindu Piyasena, Siew-Kei Lam and Meiqing Wu “Accelerating Continual Learning on Edge FPGA”[52], proposes a design for accelerating some of the computations of SLDA, which are first introduced in [37]. The authors’ design is shown in 3.10.

The Compute Unit is split into two core; a GEMM core, for matrix multiplication and outer product calculation, and a Vector Processing Unit (VPU) for vector operations (eg. vector addition). Both cores communicate with four banks of BRAM, so that the amount of data fetched in every cycle is not limited to the two ports of a single BRAM module. Each core consists of Processing Elements (PEs) that operate in parallel, with every PE contributing one element of the resulting vector or matrix. In 3.11 the architecture of a single PE is demonstrated.

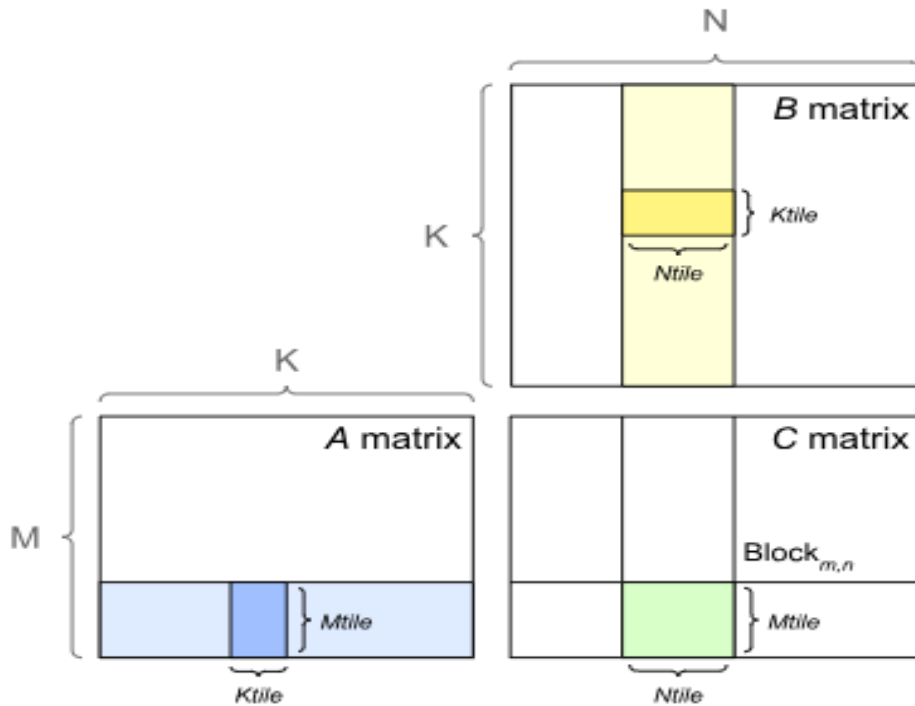


FIGURE 3.9: GEMM using tiles on GPU. Each tile of A (blue) and each tile of B (yellow) are processed by a thread block, producing a tile of C (green): [51].

3.4.3 Google TPU

The Tensor Processing Unit (TPU) was designed and deployed by Google when they realised that the demands of the users regarding inference tasks, starting in 2013, could not be met by their pre-existing datacenters. Thus a custom ASIC that can accelerate inference tasks was given high priority to produce. At its heart the Google TPU using a 128x128 systolic array of MACs that can perform 8-bit Multiply-and-Add operations on signed and unsigned integers. The TPU is on average about 15X - 30X faster than an Nvidia K80 GPU or a server-class Intel Haswell CPU, with TOPS/Watt about 30X - 80X higher. Moreover, using the GPU's GDDR5 memory in the TPU would triple achieved TOPS and raise TOPS/Watt to nearly 70X the GPU and 200X the CPU [53]. In 3.12 the TPU board is displayed as well as block diagrams of the architecture in 3.13a, 3.13b.

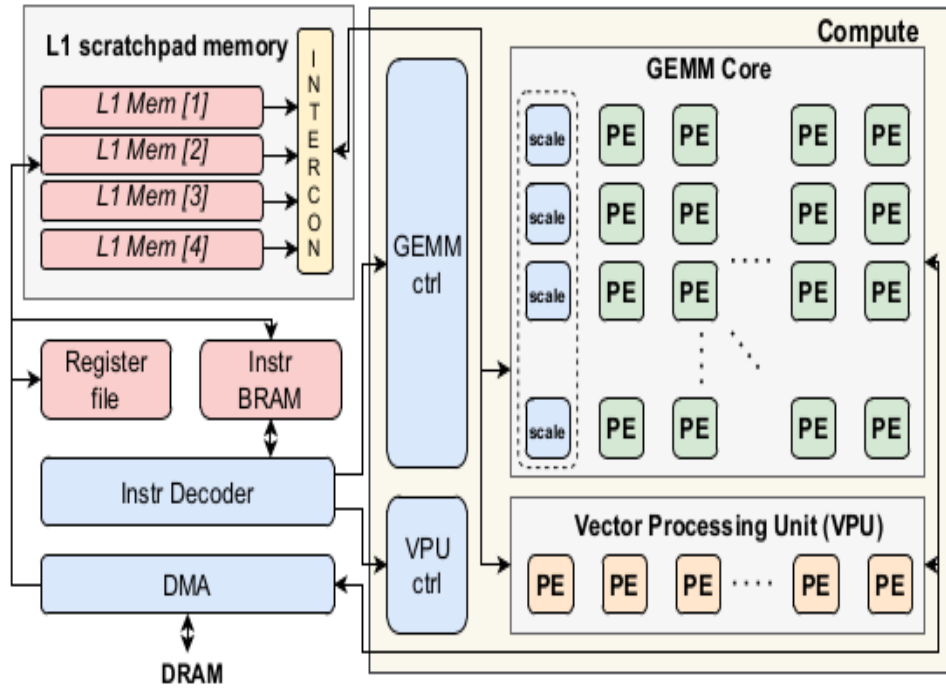


FIGURE 3.10: Hardware accelerator for SLDA, deployed on an FPGA device. Each Processing Element (PE) produces one element of the resulting vector or matrix: <https://ieeexplore.ieee.org/document/9556356>.

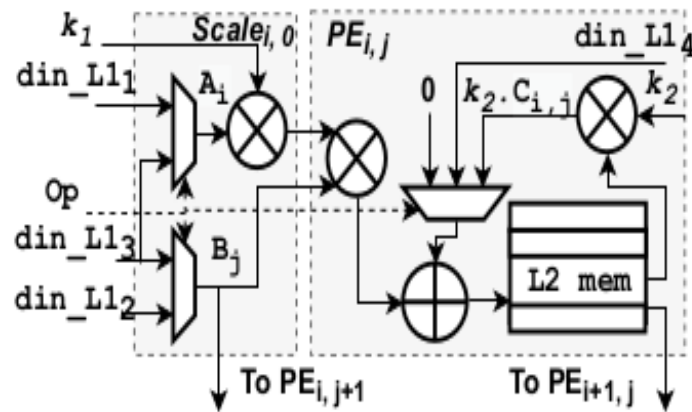


FIGURE 3.11: Processing Element used in GEMM core and VPU: <https://ieeexplore.ieee.org/document/9556356>.

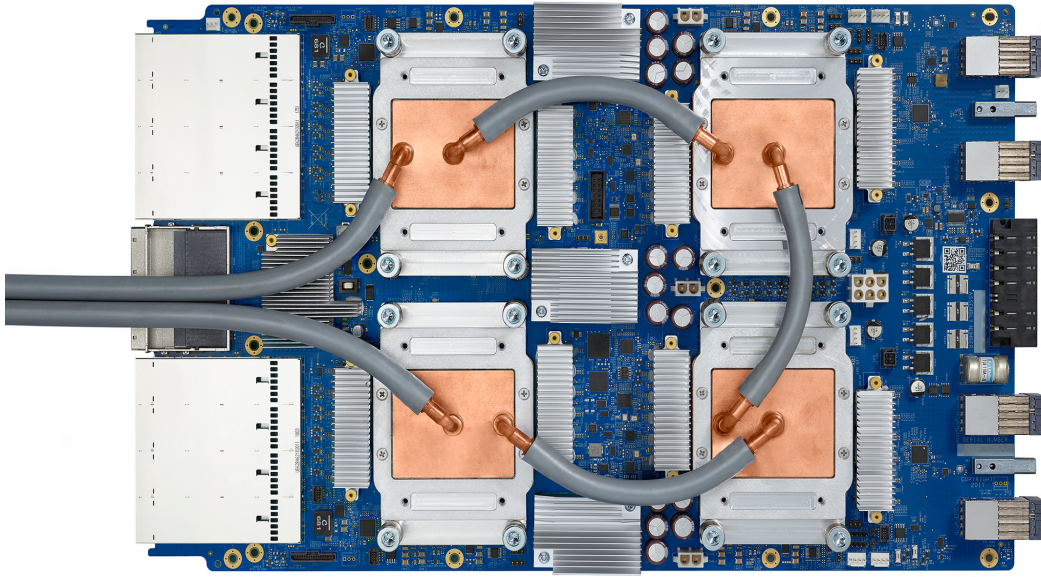
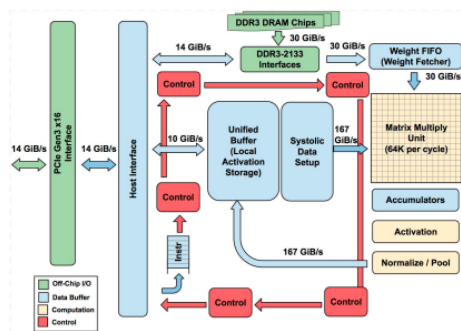
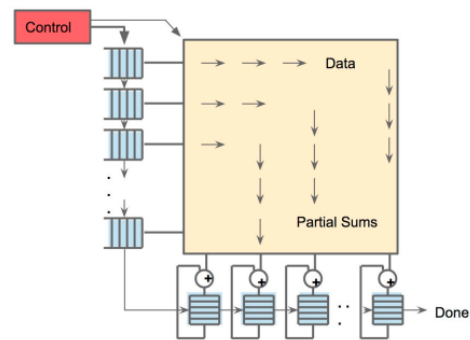


FIGURE 3.12: Google TPU top view. As can be seen there are four ASICs computing machine learning operations in parallel:

<https://cloud.google.com/tpusection-3>.



(A) Google TPU block diagram:
<https://arxiv.org/pdf/1704.04760.pdf>



(B) Google TPU systolic array architecture:
<https://arxiv.org/pdf/1704.04760.pdf>

FIGURE 3.13: TPU architectures. On the left side the top level architecture alongside with the IO and interconnections between the modules of the TPU is shown. On the right side the Matrix Multiply Unit systolic array architecture is shown:

<https://cloud.google.com/tpusection-3>.

Chapter 4

Robustness Analysis

This chapter focuses on the fact that the Static SLDA algorithm updates its internal parameters while the CNN's parameters remain still. This means that the base initialization step performed to initially train the CNN can be done with full precision. On the other hand the output vectors (which are in floating point) of the initialized CNN can be translated into a fixed point representation and be fed to the SLDA accelerator. This way the accelerator can leverage the high throughput of the fixed point operations without compromising on accuracy and network stability. Of course, the right fixed point representation must be picked and so an analysis on the distribution of the feature vectors is needed. The CNN used for the software implementation of the continual learning system is the ResNet18 coupled with the CIFAR-10 dataset. This dataset is not particularly made for continual learning applications making the task at hand more challenging.

4.1 Arithmetic Representation

4.1.1 Fixed Point Representation

Numbers in any kind of hardware are represented as a sequence of 1s and 0s and the way these sequences are interpreted either as fixed point numbers or floating point numbers. Fixed point number terminology is going to be discussed in this section.

Signed binary numbers in hardware can be represented in three different ways [54] [55], which are:

- **Sign Magnitude Form:** In this form the sign of the number is represented by the MSB (Most Significant Bit). If the MSB is 0 then the number is positive, if it is 1 then the number is negative. The remaining bits

determine the magnitude of the number (e.g. if the number is 8-bit then only the first 7 bits can be used for the number magnitude)

The sign magnitude form of a binary number is illustrated in ??.

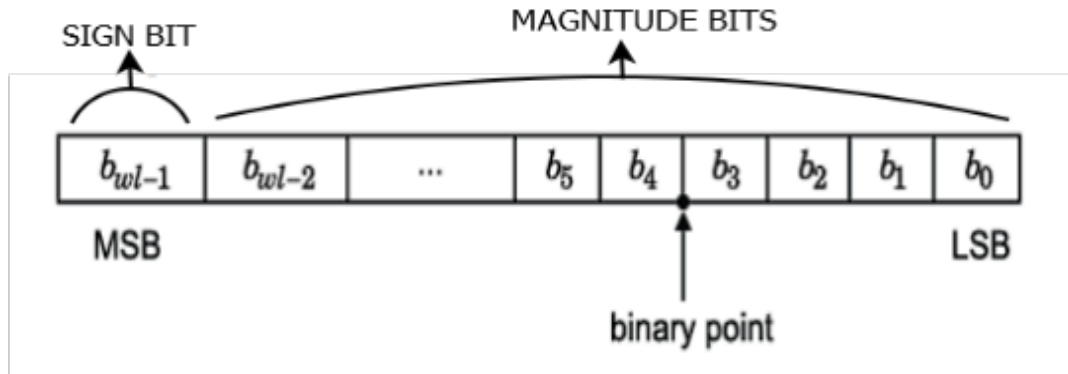


FIGURE 4.1: Fixed Point Number sign magnitude representation: <https://www.mathworks.com/help/dsp/ug/concepts-and-terminology.html>.

- **1's complement:** By complementing each bit in a signed binary integer, the 1's complement of a number can be derived. A result is a negative number when a positive number is complemented by 1. Similar to this, complementing a negative number by 1 results in a positive number 4.2. An illustration can be seen in figure 4.2.
- **2's complement:** By adding one to the signed binary number's 1's complement, a binary number can be converted to its 2's complement. Therefore, a positive number's 2's complement results in a negative number. The complement of a negative number by two yields a positive number. An illustration can be seen in figure 4.2.

In this work, the representation that is used by the libraries, both in software and in hardware, is the sign magnitude form, which means that when a signed fixed point number is declared the first bit can only be used to show the sign.

4.1.2 Fixed Point operations: Advantages and Disadvantages

Machine Learning applications and structures predominantly make use of the high precision that is provided by floating point numbers, especially when it comes to training and optimization algorithms. During the back-propagation algorithm gradients are propagated through the network, with

$$\begin{array}{r}
 \begin{array}{cccc}
 & 2^1 & 2^0 & 2^{-1} \\
 0 & 1 & 0 & . & 1 \rightarrow 2.5 \\
 1 & 0 & 1 & . & 0 \rightarrow (1's \text{ complement of } 2.5) \\
 & & & & +1 \\
 \hline
 1 & 0 & 1 & . & 1 \rightarrow -2.5 \\
 & & & & (2's \text{ complement of } 2.5)
 \end{array}
 \end{array}$$

FIGURE 4.2: Fixed Point Number 1's complement and 2's complement representation:
<https://www.geeksforgeeks.org/fixed-point-representation/>.

each layer iteration further diminishing the gradients. Fixed point operations, with their limited precision, further exacerbate this phenomenon leading to the problem of the vanishing gradient. On the other hand, Zhang et al.[56] observed that the degradation of training accuracy is mainly attributed to the dramatic change of data distribution, which may lead a more efficient way to do backpropagation using fixed point representation.

This thesis' novelty stems from the fact that the SLDA algorithm does not rely on updating the network, only on the output feature vector that is produced after a feedforward pass of the data through the network. Thus, even though the accuracy of the continual learning algorithm takes a toll over the lower precision arithmetic, it is not catastrophic. This revelation means that the FPGA can utilize the fixed point operations to achieve higher throughput and speedup.

The data type that was selected for the accelerator is 8-bit fixed point, that pertains to all the submodules. Furthermore, the integer part uses the 4 most significant bits and therefore the fractional part uses the remaining 4 least significant bits. The aforementioned bitwidth is deduced by receiving all the feature vectors that are produced by forward passing images through the CNN and finding the distribution of the data, put differently the mean and the variance. The expected distribution is a normal distribution, as the population of data is immense and the Law of Large numbers applies. This is visible in 4.3

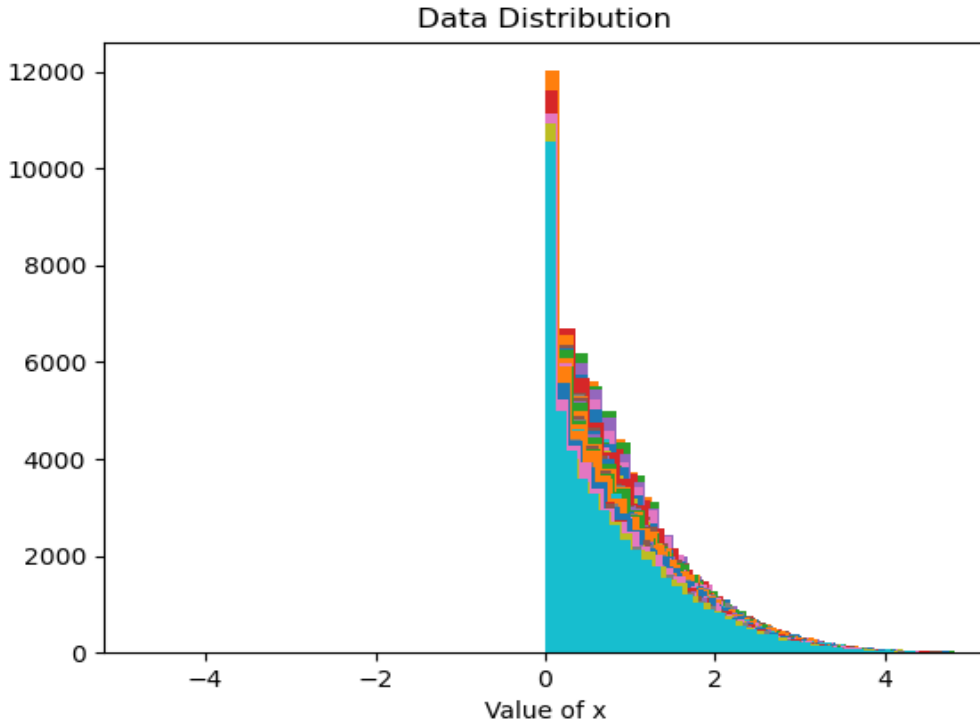


FIGURE 4.3: Distribution of the elements of the feature vectors. As can be seen the largest possible value does not exceed the 3 bit precision of the integer part.

Using 3 bits for the integer part (excluding the sign bit) the range of integers that can be portrayed in unsigned fixed point numbers is $[0, 7]$, as the smallest number is 000 and the largest is $111 = 1 * 2^0 + 1 * 2^1 + 1 * 2^2 = 7$. The probability of receiving a value that is out of range in a feature vector is zero judging by the distribution. This is not guaranteed though as the distribution varies between datasets and CNN architectures. From the above, the bitwidth satisfies the conditions that need to be met by the integer part.

The smallest number that can be represented with 5 fractional bits (except 0) is $00001 = 2^{-4} = 0.0625$ which does not allow for high precision but it is adequate for the application.

4.1.3 Static SLDA over Plastic SLDA

Initially when studying the works of [52][37] one comes to the realization that there are two primary versions of SLDA; Static SLDA, where the shared covariance matrix remains unaltered throughout the training process, and Plastic SLDA, where the shared covariance matrix updates with each iteration of the algorithm. The evaluation of both methods in the articles show

that the gain in accuracy is apparent, in Plastic SLDA, without the difference being dramatic. In this thesis only the Static SLDA algorithm is supported as the goal is not to achieve state of the art accuracy results but to make a fully operational low-cost machine learning system with continual learning capabilities.

Furthermore, the proposed accelerator is designed to perform all matrix multiplications and vector operations, which means that the updating of the shared covariance matrix would take up most of the resources and bandwidth on the FPGA, due to the inverse matrix calculation needed to compute Lambda

$$\Lambda = [(1 - \epsilon) \cdot \Sigma + \epsilon \cdot I]^{-1}$$

. The trade off is not advantageous, especially when it comes to smaller edge devices.

4.1.4 Static SLDA algorithms

The algorithms that are essential for the operation of Static SLDA are presented here, in the form of pseudo code.

First of all the means matrix and the covariance matrix must be initialized based on the feature vectors that are produced in the base initialization stage. The covariance matrix is determined using the Oracle Shrinkage Estimator[57], and is calculated in full precision. The means matrix is calculated using the code in Algorithm 1.

Algorithm 1 Init means algorithm

Require: *feature_vector, label, class_counters*

- 1: $means[label] \leftarrow \frac{class_counters[label] \times means[label] + feature_vector}{class_counters[label]}$
 - 2: $class_counters[label] \leftarrow class_counters[label] + 1$
-

The Lambda matrix is initialized and remains unaltered using the equation

$$\Lambda = [(1 - \epsilon) \cdot \Sigma + \epsilon \cdot I]^{-1}$$

.

The Static SLDA method needs to update the Weights and biases of the classifier after every feature vector passes through it. Algorithm 2 showcases how this update is performed.

Algorithm 2 Update means, weights and biases algorithm**Require:** *feature_vector*, *label*, *class_counters*, *Lambda*

- 1: $means[label] \leftarrow \frac{class_counters[label] \times means[label] + feature_vector}{class_counters[label]}$ \triangleright Update class mean
- 2: $class_counters[label] \leftarrow class_counters[label] + 1$
- 3: $Weights \leftarrow Lambda \cdot means$ \triangleright Matrix multiply Lambda with means
- 4: **while** $i \leq num_classes - 1$ **do**
- 5: $biases[i] \leftarrow -0.5 \times (means[i] \cdot Weights[:, i])$ \triangleright Dot product between every row of means and every column of Weights
- 6: $i \leftarrow i + 1$
- 7: **end while**
- 8: return *Weights*, *biases*

In Algorithm 2 all the elements of the feature vector, the means matrix, the Lambda matrix, the weights matrix and the biases vector can be in floating point or fixed point representation, depending on the user's needs.

Finally, when desiring to test the capabilities of the classifier Algorithm 3 computes the prediction scores for a single image, and consequently the feature vector that is generated from it.

Algorithm 3 Compute Scores algorithm**Require:** *feature_vector*, *Weights*, *biases*

- 1: $scores \leftarrow Weights \cdot feature_vector + biases$ \triangleright Matrix-vector multiply between Weights and *feature_vector*
- 2: return *scores*

4.1.5 Software Implementation of Static SLDA

4.1.6 NumPy

NumPy is a Python package that is regularly utilized in scientific computing, implementing most linear algebra operations and other mathematical operations, such as trigonometric operations. Some of the operations are shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations and random simulation. The basic block of the NumPy package is the *ndarray* that is an array of identical data type elements that are stored in contiguous memory.

Numpy uses a plethora of optimizations that render it one of the best options for linear algebra calculations that rival even naive GPU kernels. Some of the optimizations are listed below[58][59][60]:

- **Vectorized Operations (SIMD):** NumPy utilizes the vector instructions provided by the specific architecture of the CPU used (x86) and can process multiple data with one instructions, without the use of explicit for loops. These operations are implemented using highly optimized C and Fortran routines, making them much faster than traditional Python loops.
- **Multithreading:** By default, NumPy utilizes multithreading if possible. Therefore, a modern multicore CPU is used to its full extent.
- **Optimized C/Fortran routines:** NumPy is a light wrapper that includes many routines that are written in optimized C/Fortran. The algorithms used for linear algebra are state of the art and use the BLAS library under the hood.
- **Contiguous memory layout:** All ndarrays in NumPy are stored contiguously in memory, which allows for efficient access and manipulation of array elements.

4.1.7 CuPy

CuPy is a drop-in replacement of NumPy, that leverages the capabilities of parallel computing on NVIDIA or AMD GPUs. Furthermore, it is a library that enables the user to easily access GPU-accelerated kernels from a Python API. Under the hood, it uses other low level GPU-accelerated libraries such as CuBLAS, CuFFT, CuSPARSE, CuSOLVER, CuRAND and CuTENSOR to achieve the best possible performance.

Some of the optimizations over NumPy are [61]:

- **GPU acceleration:** CuPy uses the GPU capabilities to accelerate the NumPy methods.
- **Implemented in CUDA:** The whole API is implemented in CUDA and so it is natively optimized to run on GPUs.
- **CUDA kernel generation:** CuPy automatically generates optimized CUDA kernels for array operations, such as element-wise arithmetic, broadcasting, and reductions.
- **Memory transfer optimizations:** The memory transfers from CPU to GPU and vice versa, are optimized in CuPy

Pretrained Python CNNs

The ResNet18 CNN is utilized for the continual learning experiments that are going to be used as a baseline for comparison with the FPGA-implemented counterpart. This CNN model is pretrained on ImageNet1K-V1 and is provided by the Pytorch API. A detail worth mentioning is that since the ResNet18 is trained on a dataset that contains 1000 classes, the final fully connected layer dimensions must be altered according to the number of classes in the dataset. As will be described below, the CIFAR-10 dataset that has been opted contains only 10 classes so the fully connected layer has dimensions [10, 1].

Dataset and Data orderings

Many datasets are among the options for continual learning applications, such as the ImageNet1K, CoRE50, CUB200-2011 and many more. Emphasis was given though to the CIFAR10 dataset, as it is large enough for assessment of the SLDA algorithm while not taking too long to train on.

The dataset is split into five subsets of classes, with each subset containing two classes. For each new epoch, after base initialization, two more never seen classes are presented to the model, to classify without modifying the networks parameters.

The goal of this thesis is not to achieve the highest accuracy rates for state of the art datasets, but to compare the efficiency of SLDA, and continual learning methods in general on different platforms.

Network Base Initialization

Before attempting to continually train a network on CIFAR10, a base initialization of the network must take place. Therefore, the whole ResNet18 is trained and tested on the first two classes of the CIFAR10 dataset, using back-propagation and an SGD optimizer. The learning rate is set to $lr = 0.1$.

SLDA among different platforms and programming languages

The platforms that were used for assessing the SLDA performance were:

- CPU: Intel Core i7 11800H
- GPU: NVIDIA RTX 3050Ti
- Google TPU

- FPGA: ZCU102

On the CPU side, SLDA was implemented in C++, along with the Eigen Linear Algebra library, and Python, along with NumPy. The data types tested in C++ are 16-bit fixed point, 32-bit fixed point and float, while the data types tested in Python are 16-bit fixed point and float.

On the GPU side, a C++ CUDA kernel was implemented for 32-bit floating point data types, using the CuBLAS library that provides optimized kernels for matrix multiplication, vector operations and dot products.

On the FPGA side, the accelerator was implemented in Vitis HLS, using 16-bit fixed point arithmetic and floating point arithmetic.

Chapter 5

System Architecture and FPGA Implementation

In this chapter the tools provided by Xilinx for designing, implementing and downloading hardware architectures on an FPGA device, are presented. Moreover, a thorough analysis is performed on how the architectural decisions were made regarding the SLDA accelerator's internal behaviour. Also included with the analysis are detailed block diagrams that are presented from top to bottom showcasing the whole system's IO and compute capabilities.

5.1 Hardware Architecture

The accelerator of the whole system has as the main focus to accelerate the computations needed for the training process of **Static SLDA** as well as the inference process. The accelerator is divided in four primary units, which are:

- **Update Means Unit (Training):** This Unit updates the means of the different classes according to the Static SLDA algorithm
- **Compute Weights Unit (Training):** This Unit computes the Weight matrix according to the Static SLDA algorithm
- **Compute Biases Unit (Training):** This Unit computes the bias vector according to the Static SLDA algorithm
- **Compute Scores Unit (Inference):** This Unit uses the produced Weight matrix and bias vector to transform the feature vector and compute the scores for each class

As can be understood from the above, the granularity that was chosen is not extremely fine-grained, and as a result the level of abstraction, when compared to similar works [52], is higher. This decision was made intentionally as the purpose of the core is not to provide general purpose functionalities but to accelerate very specific mathematical operations over a predetermined data type (16 bit fixed point data). The target of this accelerator is to perform all the computations of the Static SLDA on the FPGA, without having to transfer data from the PL side to the PS side.

All the inputs are received from the PS side initially through the use of streams that are synthesised as FIFOs that buffer them and store them in cache on the accelerator. The width of the data streams is 128-bits uniformly, and as a consequence 16 8-bit integers can be transferred simultaneously. The data width cannot reach higher limits as the Master and Slave AXI ports on the ZCU102 are at best 128-bits wide.

All the intermediate results produced by the submodules are stored in fragmented BRAM and can be accessed very quickly in succeeding iterations of the algorithm.

The output of the accelerator is stored in BRAM that is memory mapped and is easily accessible by the PS side, through AXI-Interconnect.

More details on how the PS side communicates with the PL side are discussed in chapter 5.

A block diagram of the top module of the SLDA accelerator, with details on how the data get transferred to and from the PS is illustrated in figure 5.1. A lower level image of the SLDA accelerator is illustrated in figure 5.2, and includes information on how the different modules in the accelerator produce and consume data.

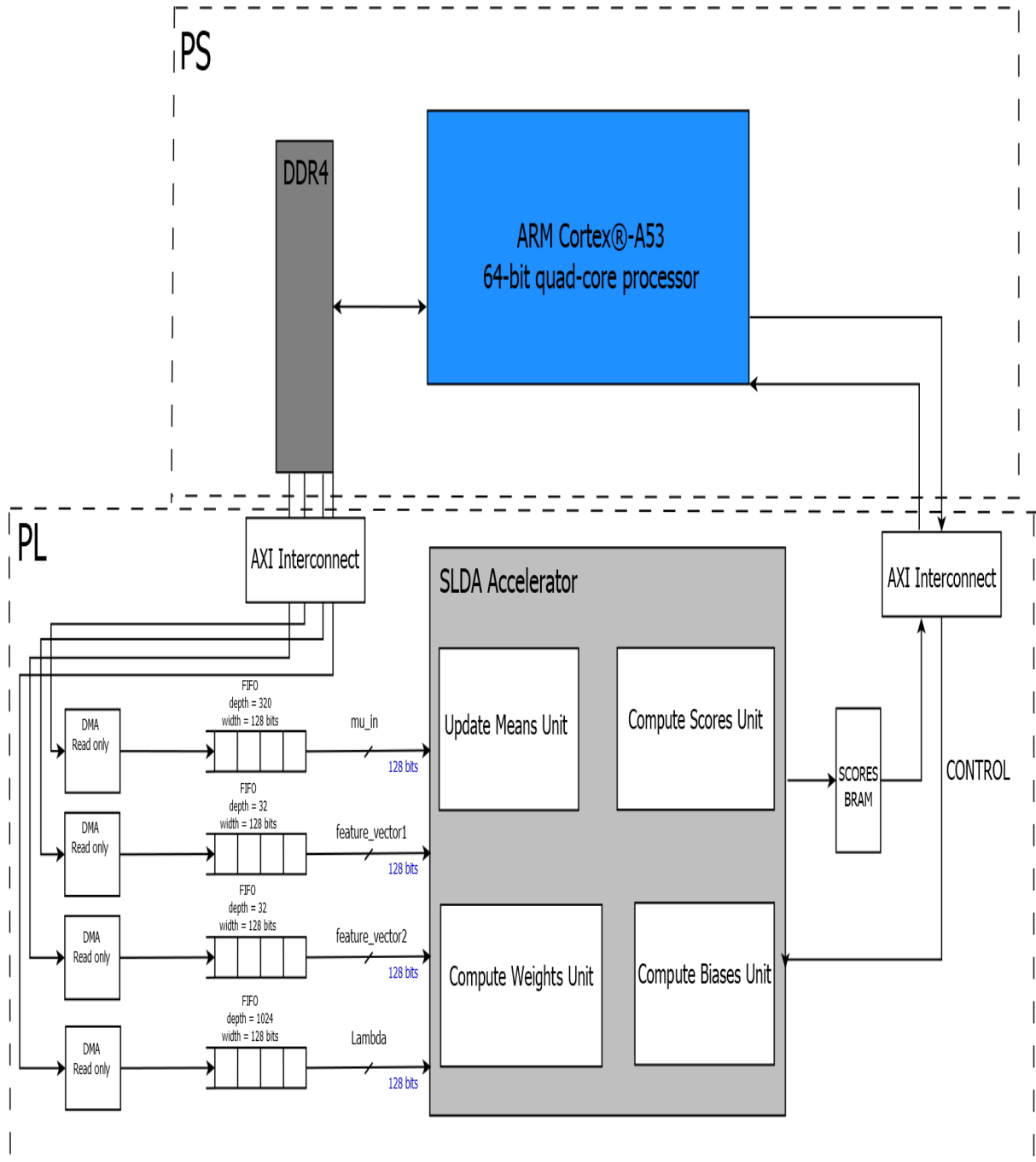


FIGURE 5.1: A higher level top module of the Static SLDA accelerator, including the connections between the PS and PL.

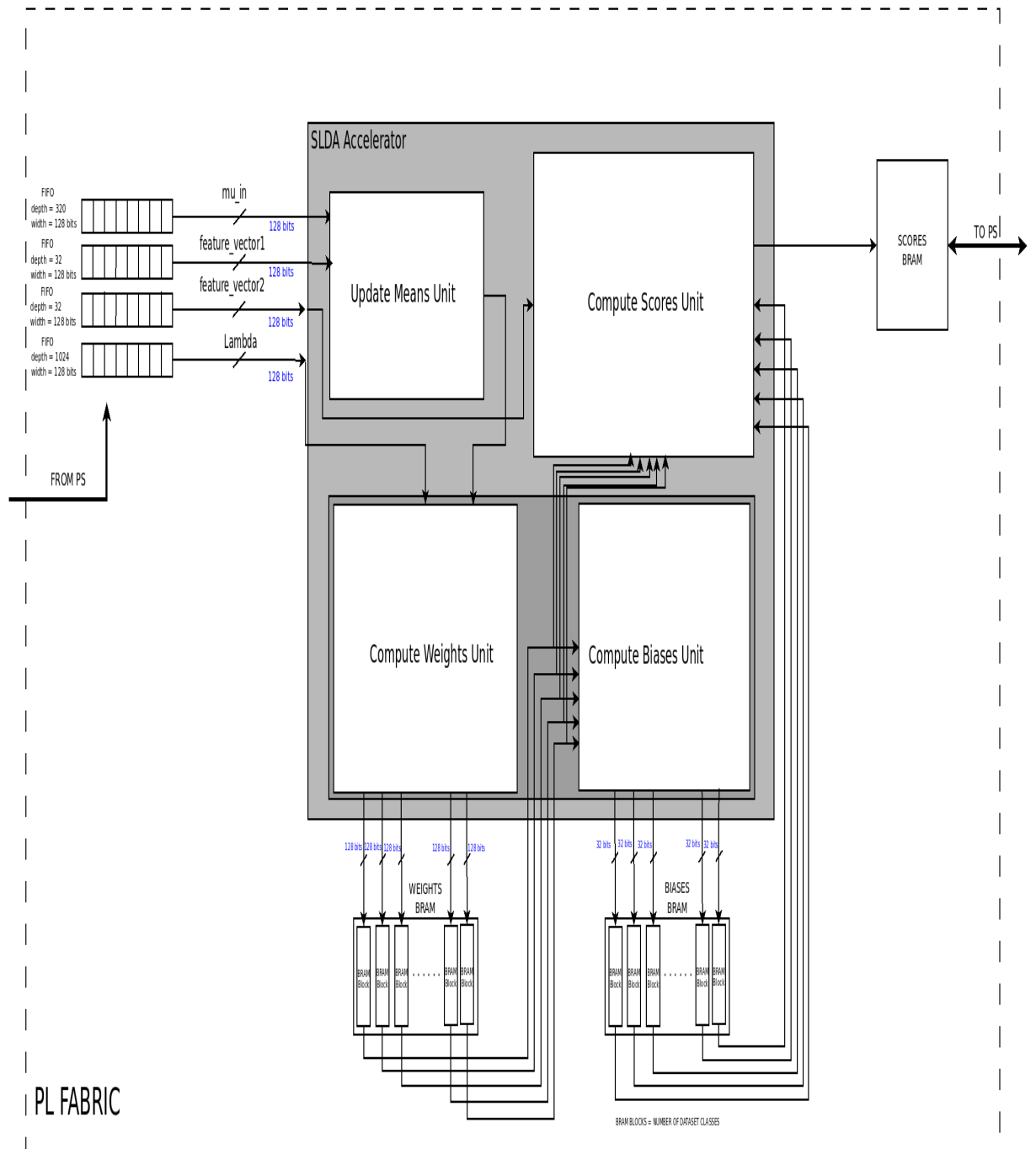


FIGURE 5.2: The top module of the Static SLDA accelerator, that shows in more detail the inner working of the accelerator.

5.1.1 Update Means Unit

The Update Means Unit reads the values of the feature vector and the means of the classes and then updates the means and stores them in the local cache for the subsequent unit to be able to access the data faster. There is no need for the application running on the ARM processor to have knowledge about the new means of the classes. That leads to the fact that after base initialization the means can be accessed from cache without needing to read them from the DDR. Only the feature vector needs to be read from DDR in every single iteration of the algorithm.

The operation to update the means is the following[52][37]:

$$\mu_{k,t+1} = \frac{counter_k}{counter_k + 1} \cdot \mu_{k,t} + \frac{x_t}{counter_k + 1}$$

where $counter_k$ is the number of samples encountered in class k. The counter of class k is updated as follows when a new sample belonging in that class is found.

$$counter_k = counter_k + 1$$

.

The division and addition operations are performed on blocks of data that are 128 bits wide, as to perform multiple of these operations per cycle.

The block diagram of the Compute Means Unit is shown in figure 5.3

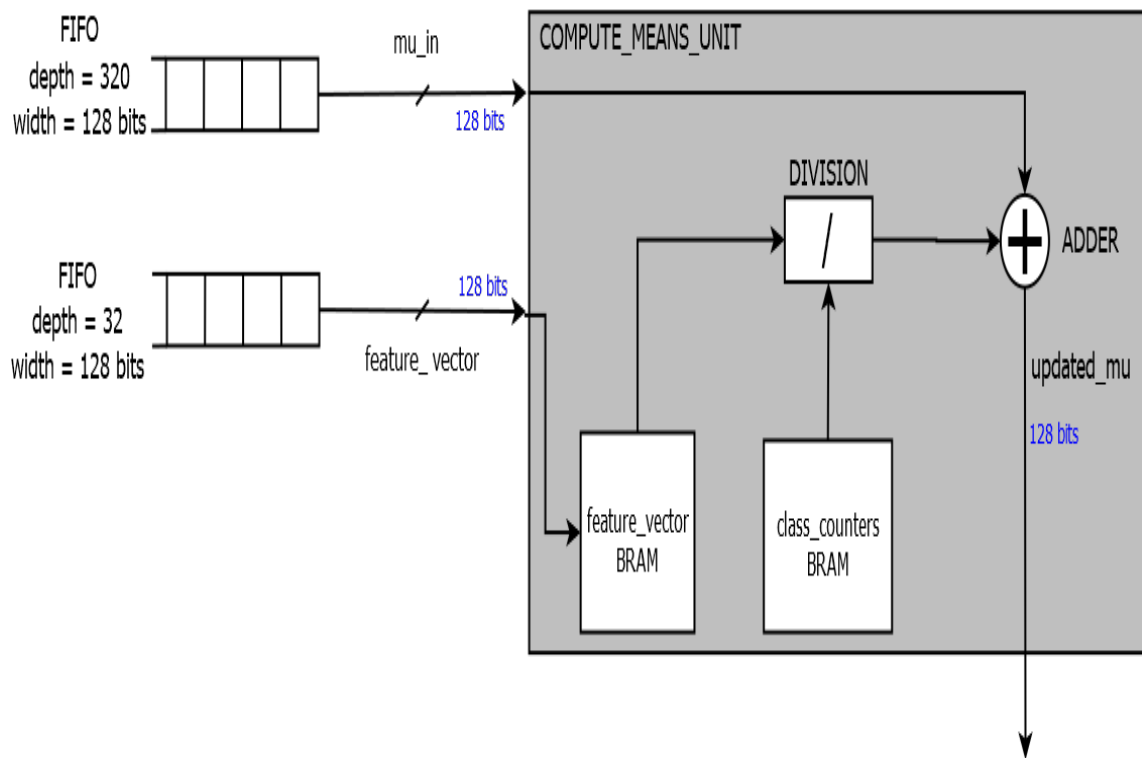


FIGURE 5.3: The submodule that updates the means required for computing the weights and biases of the linear classifier.

5.1.2 Compute Weights Unit and Compute Biases Unit

These Units sits at the heart of the accelerator and take up most of the resources and the area on the FPGA. The main target of the cores is to produce the Weight matrix and the bias vector of the linear classifier used in SLDA as fast and as efficiently as possible.

The input vectors of data have a width of 128 bits, that means that 16 8-bit fixed point values can be transferred to the module with one read operation on the BRAM. The blocks of cache memory that store the values of Λ and μ partitioned in a way which enables the access of all the rows of $\Lambda \in \mathbb{R}^{N \times N}$, where $N = 32$, matrix and all the columns of $\mu \in \mathbb{R}^{N \times M}$, where M = number of classes, at the same time.

Moreover, The unit consists of $M \times N$ small systolic arrays of size $S = 16 = 4 \times 4 = \text{data_transferred_per_read_op}$ that perform Multiply and Accumulate operations on the 8-bit data. Each individual PE element of the systolic array computes a partial sum that is then transferred to the next PE element. Consequently, The last PE element computes the correct weight value that is ready to be transferred. Lastly, The weight matrix is also outputted in blocks of data that, with each block being produced by the respective systolic array.

The architecture of the Compute Weights Unit is shown in 5.4.

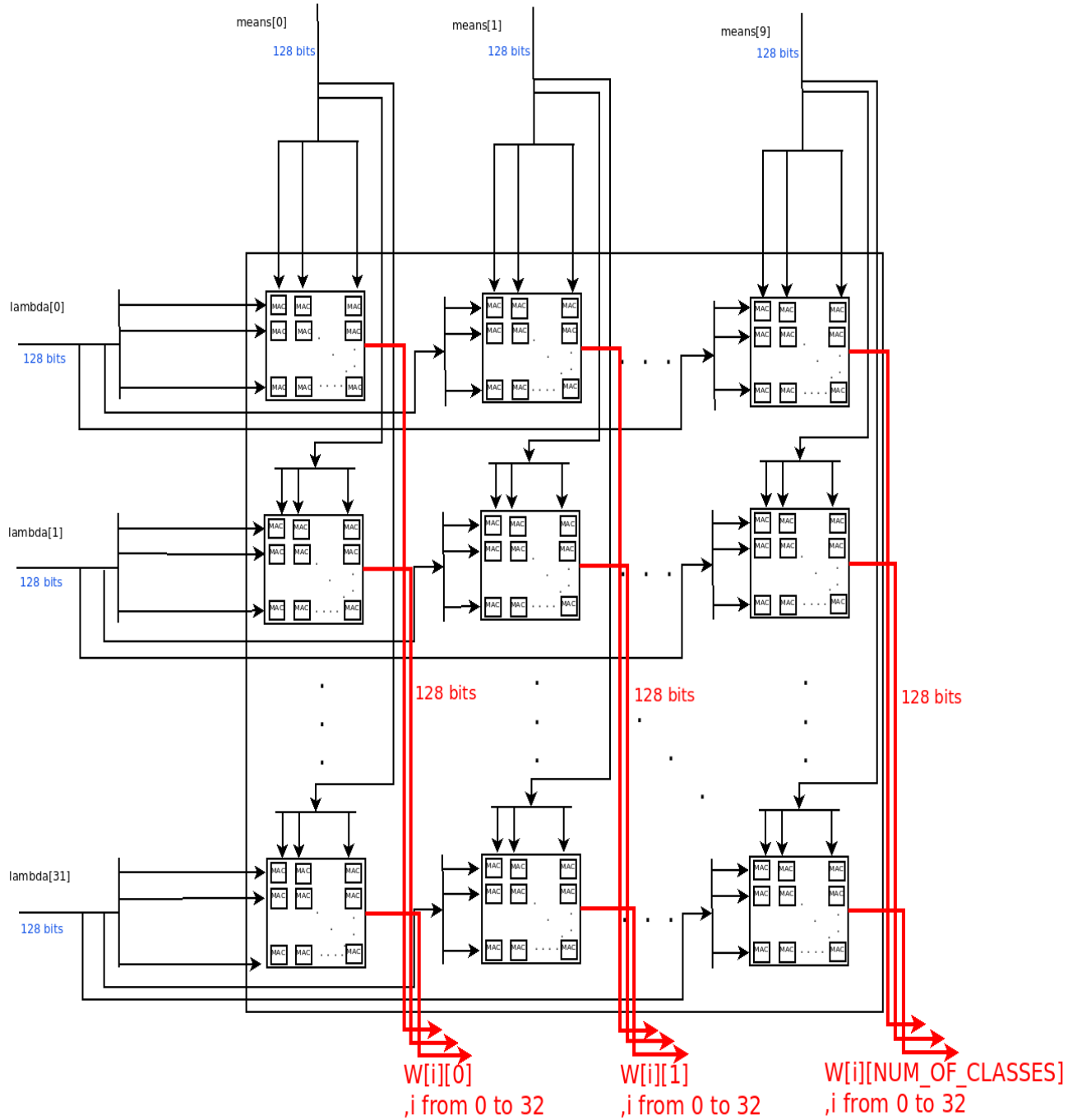


FIGURE 5.4: The submodule that computes the weights required for the linear classifier. The PE elements are notated with "MAC".

The Compute Biases Unit utilizes the same size systolic arrays used in the Compute Weights Unit. The grid of systolic arrays though is one dimensional, just as the bias vector, and the dot products that need to be computed are only ten in contrast to the 320 needed for the computation of the weight matrix. Consequently, this unit is a lot smaller but uses the same principles as the previously mentioned unit.

The architecture of the Compute Biases Unit is shown in 5.5.

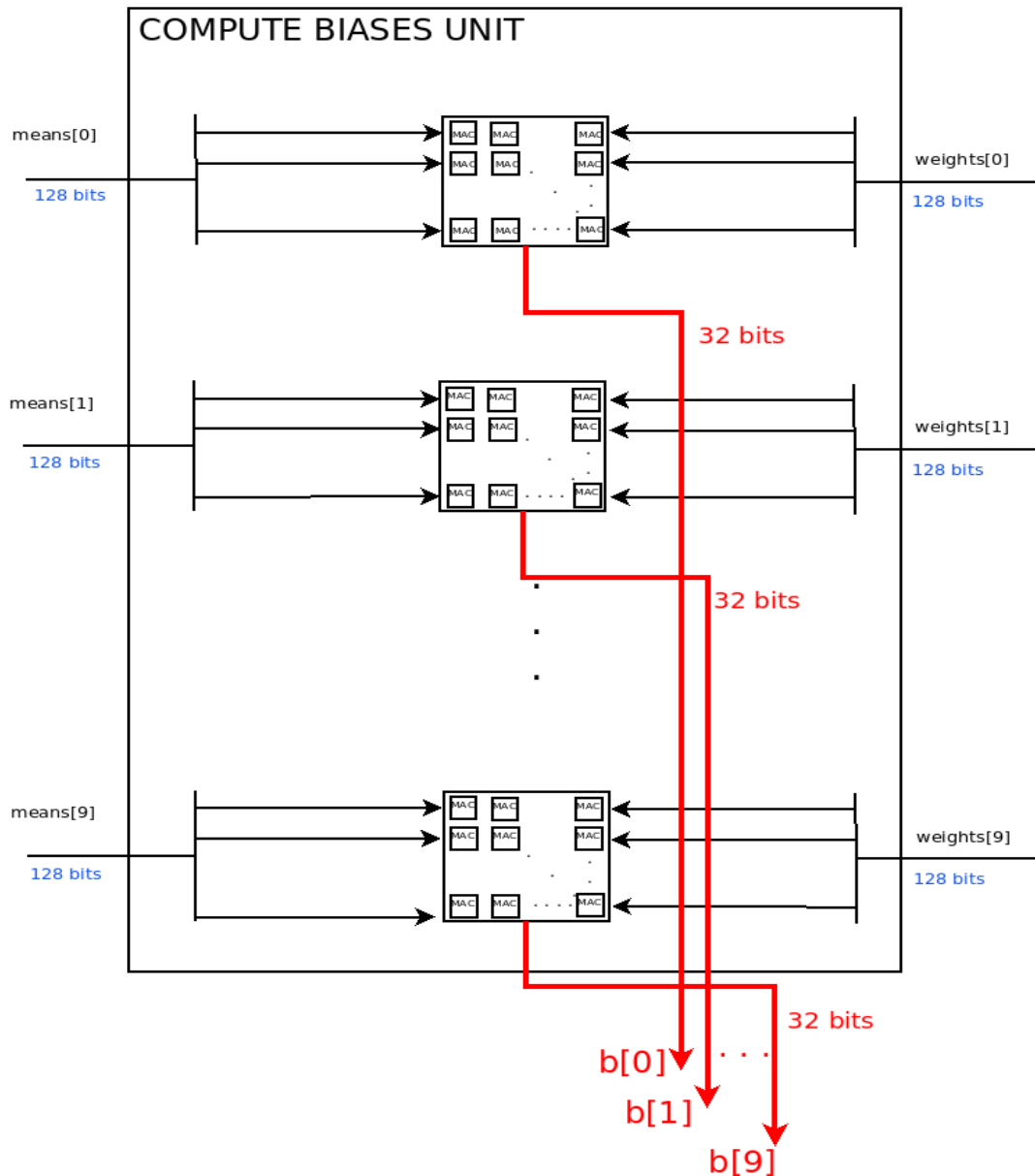


FIGURE 5.5: The submodule that computes the biases required for the linear classifier. The PE elements are notated with "MAC".

5.1.3 Compute Scores Unit

Inference is performed in the Compute Scores Unit, on every sample that is processed by the neural network and used to update the means and the counters of each class. The mathematical operation is a simple linear transformation of the feature vector denoted as

$$scores = W \cdot x_t + b$$

, where x_t is the feature vector [52][37].

The feature vector is loaded from external memory after the CNN has computed it.

For the calculation of the Matrix-Vector multiplication a similar unit to the Compute Biases Unit was used, with the only difference being the introduction of an adder at the end of each systolic array output, to add the bias value to the product. A block diagram of the Compute Biases Unit is illustrated in figure 5.6.

All the algorithms concerning the implementation and testing of the aforementioned units are included in Appendix A 8.

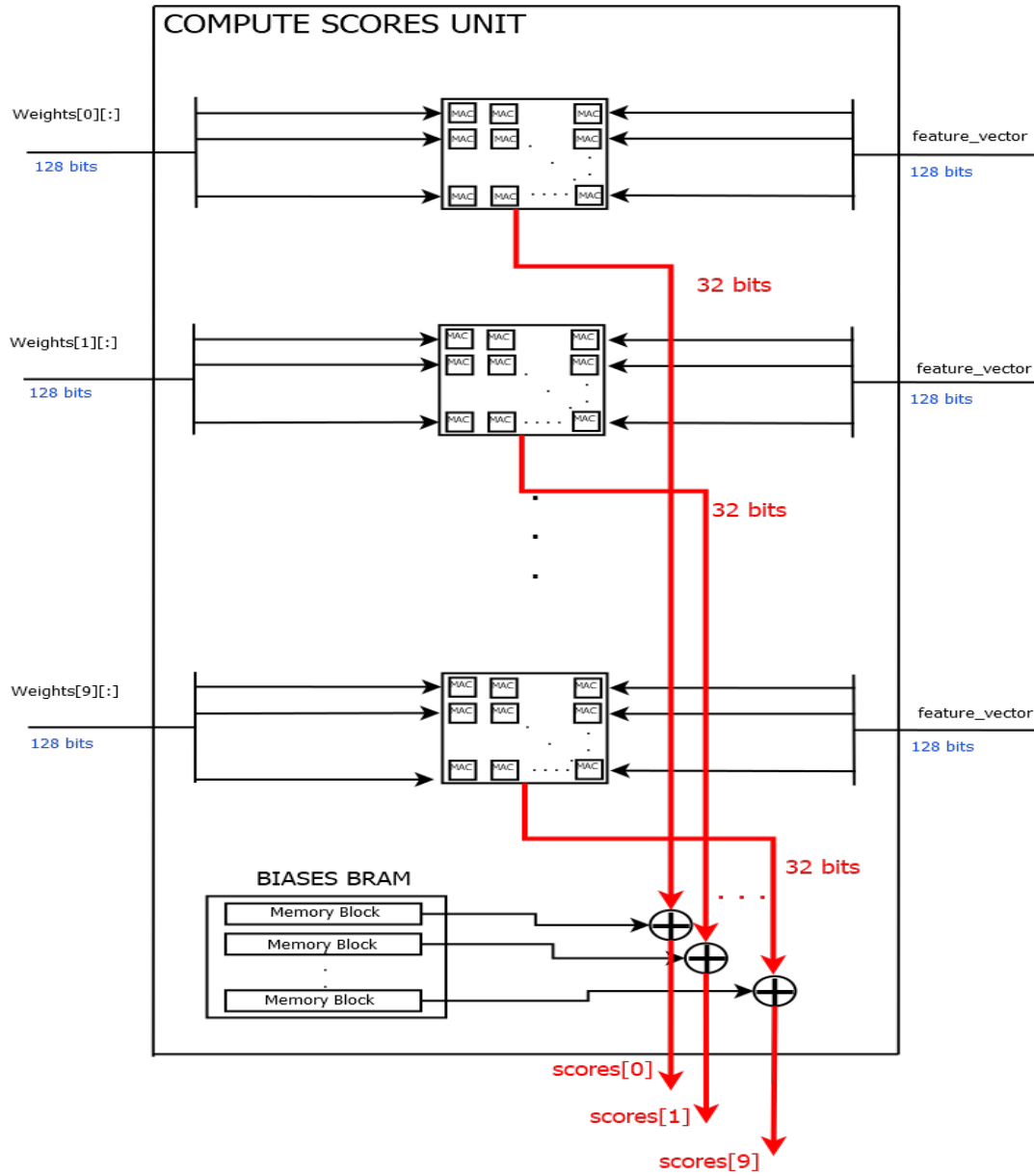


FIGURE 5.6: The submodule that computes the final scores for the image classification. The PE elements are notated with "MAC".

5.2 Tools Used

The tools that are used to develop the accelerator, connect it to the processor on the ZCU102 and develop the application that tests the accelerator are all provided by Xilinx, in its Vitis Unified Software Platform, and are going to be presented in detail in this section. The Vitis Unified Software Platform succeeded the Vivado Software Development Kit (SDK) bringing alongside it some indispensable features such as Vitis AI Development Environment, Vitis Accelerated Libraries, Xilinx Runtime Library and Vitis HLS [Xilinx_Vitis_Unified].

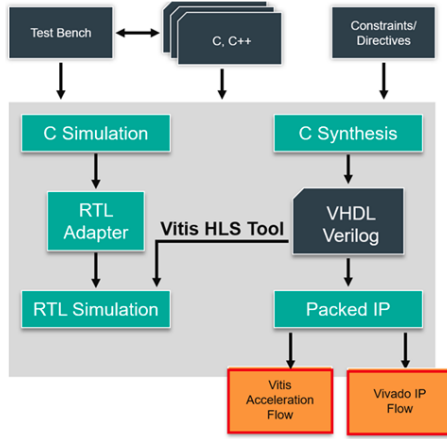
The workflow, when utilizing the tools that are provided by the Vitis Unified Platform, goes as follows:

- Identify the performance-critical portions of the application that demands acceleration. In this case the Continual Learning algorithm.
- Design the Accelerator using Vitis High Level Synthesis in C/C++, OpenCL or RTL. C++ was the programming language of choice.
- Build, Analyze and Debug to verify functional correctness and validate performance goals are met.
- Export the module to be used in Vivado IDE, where the whole platform, including the ways the PS side communicates with the PL, is designed.
- Deploy Accelerated Application on the ZCU102 using Vitis Unified Platform. The drivers of the custom made accelerator and other modules are inferred by the platform automatically.

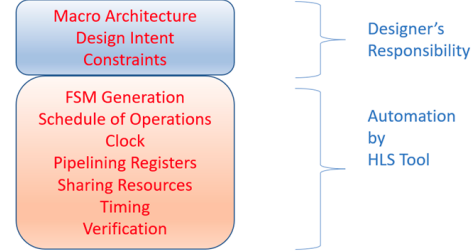
5.2.1 Vitis High Level Synthesis (HLS)

Vitis HLS is a tool that allows the user to generate complex RTL designs by synthesizing C/C++ functions. HLS can enable the path of creating high-quality RTL, rather quickly than manually writing error-free RTL. The design and the macro-architecture of it, fall to the hands of the developer who should specify the constraints, like clocking and performance, and the interfaces used to communicate with the outside world. The micro-architecture decisions are all handled by the tool (FSMs, datapath, register pipelines) so the process of designing RTLs is a lot simpler than using HDL languages[62].

In 5.7a the development flow of the HLS tool is shown, while in 5.7b the design processes are illustrated.



(A) Vitis HLS development flow:
<https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS>



(B) Vitis HLS design processes:
<https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Navigating-Content-by-Design-Process>

Optimization Directives

The optimization directives provided by Vitis HLS allow the user to alter the synthesized design to their liking, without having to interfere with the source code across different solutions. Regularly these directives optimize the design further rendering them almost essential to use in many applications. The directives are added into the code by using the pragma command of the preprocessor. Alternatively, they can be added using the tcl command `set_directive_*`[63].

The directives that were used in this thesis are explained below while the directives that were not used are just mentioned for completion.

- **Array Partition:** Partitions an array into smaller arrays or individual elements. Results in RTL with multiple small memories or multiple registers instead of one large memory. Effectively increases the amount of read and write ports for the storage. Potentially improves the throughput of the design. Requires more memory instances or registers.
- **Dataflow:** Specifies that dataflow optimization be performed on the functions or loops, improving the concurrency of the RTL implementation.
- **Interface:** Is only supported for use on the top-level function, and cannot be used for sub-functions of the HLS component. The INTERFACE

pragma or directive specifies how RTL ports are created from the function arguments during interface synthesis. The Vitis HLS tool automatically determines the I/O protocol used by any sub-functions.

- **Unroll:** Transforms loops by creating multiples copies of the loop body.
- **Pipeline:** Reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations. A pipelined function or loop can process new inputs every N clock cycles, where N is the initiation interval (II). An II of 1 processes a new input every clock cycle.
- **Stream:** If the data stored in the array is consumed or produced in a sequential manner, a more efficient communication mechanism is to use streaming data, where FIFOs are used instead of RAMs. When an argument of the top-level function is specified as INTERFACE type `ap_fifo`, the array is automatically implemented as streaming. When the array is implemented in a DATAFLOW region, it is common to use the `-depth` option to reduce the size of the FIFO.

Vitis HLS supports other directives that are not used in this thesis.

5.2.2 Vivado IDE

The Vivado IDE provides the users with an interface that makes hardware designing more intuitive and faster. All the tools can be used through the IDE's GUI or through tcl commands on the tcl console. The Vivado Design Suite replaces the ISE Design Suite. It replaces all of the ISE Design Suite point tools, such as Project Navigator, Xilinx Synthesis Technology (XST), implementation, CORE Generator tool, Timing Constraints Editor, ISE Simulator (Vivado simulator), Vivado, Chip Scope Analyzer, Power Analyzer, FPGA Editor, PlanAhead design tool, and SmartXplorer[64].

Users can integrate to their designs IPs that are given by Xilinx for free or their custom made IPs that are exported from Vitis HLS.

After running Synthesis and Implementation on the design, on vivado creates a variety of reports such as timing report, methodology report, DRC report, utilization report, power report and schematic. These reports are essential for the architect to validate the correctness of the design. Through them the architect can also check how the entire system is implemented under the hood.

Moreover, when a design passes the implementation part, a bitstream can be created and exported so that it can be downloaded onto a platform, or used in Vitis Software platform to develop an application.

5.2.3 Vitis Unified Software Platform

The Vitis Unified Software Platform is the Environment where the embedded application development is performed. It provides the user with all the libraries and tools for debugging and building the application. The applications are written in C/C++ and can be targeted for baremetal or for linux OS, like Petalinux and FreeRTOS[65] [66] [67].

Firstly, one must import a hardware platform in the form of a bitstream file, that is exported from Vivado IDE. Then Vitis generates the required drivers for all the cores that are used for accessing the hardware through software. Additionally, Vitis let the user interfere with the BSP settings and configure STDIO peripheral settings, compiler flags, add/remove libraries, assign drivers to peripherals etc.

The user can transmit the program through JTAG locally and debug it that way or build a hardware server and set a new debug configuration that connects to the server's IP and sends the program that way. One must be cautious that the hardware server must be the same exact version as the Vitis Unified Software Platform.

5.3 Platforms and Xilinx Cores

5.3.1 ZCU102

The platform of choice in this dissertation is the ZCU102 Zynq Ultrascale+MPSoC. The ZCU102 is a general purpose evaluation board for rapid-prototyping based on the Zynq® UltraScale+™ XCZU9EG-2FFVB1156E MPSoC (multiprocessor system-on-chip). High speed DDR4 SODIMM and component memory interfaces, FMC expansion ports, multi-gigabit per second serial transceivers, a variety of peripheral interfaces, and FPGA logic for user customized designs provides a flexible prototyping platform[68].

The specifications of the board are listed in table 5.1 and can be found in detail in the following link <https://doodle.com/meeting/participate/id/dLJrmMja>

HD banks	5 banks, total of 120 pins
HP banks	4 banks, total of 208 pins
MIO banks	3 banks, total of 78 pins
PS-side GTR 6 Gb/s transceivers	4 PS-GTRs
PL-side GTH 16.3 Gb/s transceivers	24 GTHs
Logic cells	599,550
CLB flip-flops	548,160
Max. distributed RAM	8.8 Mb
Total block RAM	32.1 Mb
DSP slices	2,520

TABLE 5.1: ZCU102 specification table. [URL](#)

The Top Level block diagram of the ZCU102 is illustrated in 5.8

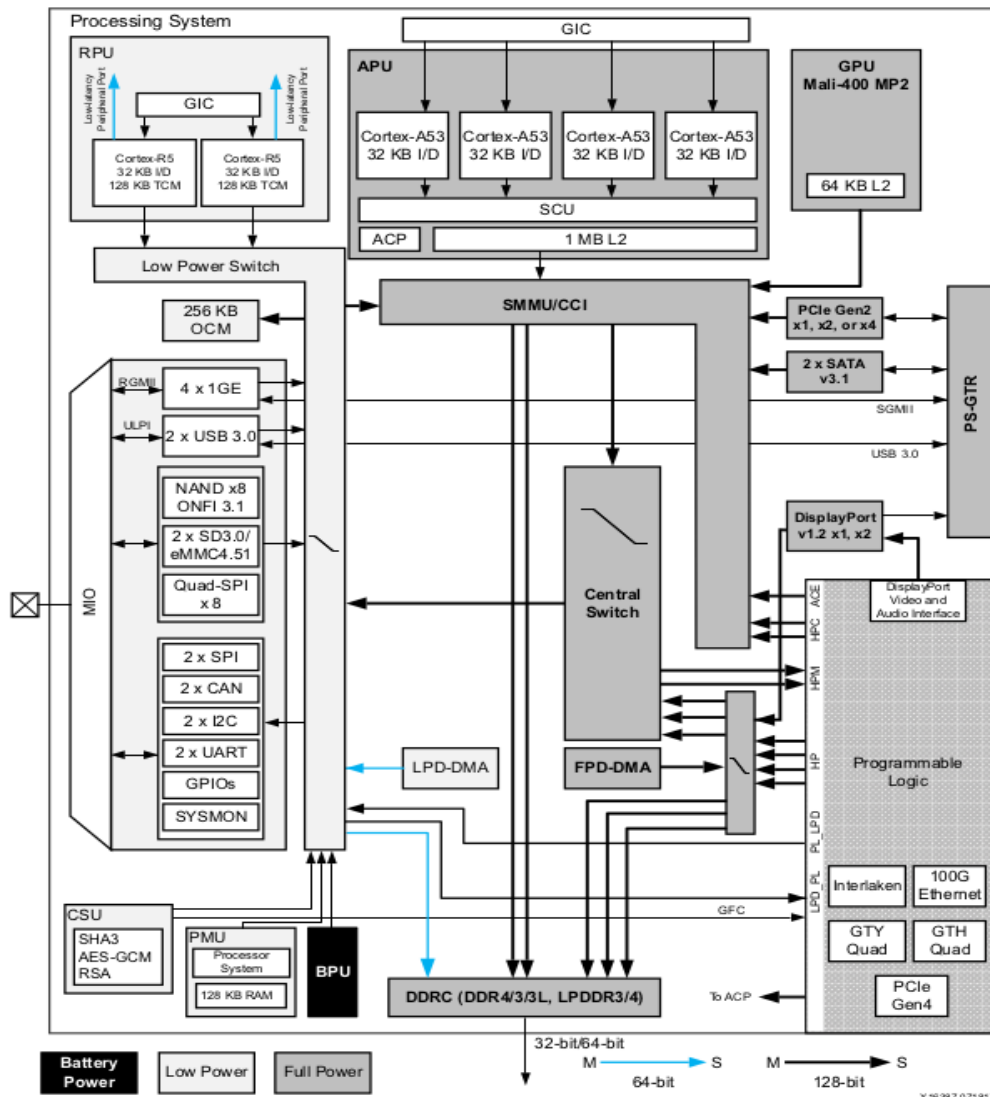


FIGURE 5.8: Top level block diagram of the ZCU102 :
<https://docs.xilinx.com/v/u/en-US/ug1182-zcu102-eval-bd>.

Chapter 6

System Verification and Performance Evaluation

In this chapter, the results of the implemented hardware design are put to the table and compared with implementations on relatively new hardware and software that is considered state of the art in linear algebra calculations.

6.1 System Verification

To verify the correct operation of the Static SLDA accelerator the whole design was downloaded on a ZCU102 FPGA platform. The following results are gained from different runs on the FPGA platform. The feature vectors that are fed to the SLDA classifier are first exported from Pytorch, after the images pass through the CNN.

6.2 Specification of Compared Platforms

6.2.1 Intel Core i7-11800H

The Intel Core i7-11800H is a high-end octa-core SoC for gaming laptops and mobile workstations. It belongs to the Tiger Lake H45 family and was announced in mid 2021. It integrates eight Willow Cove processor cores (16 threads thanks to Hyper-Threading). The base clock speed depends on the TDP setting and is 2.3 GHz at 45 W. The single core Boost can be as high as 4.6 GHz while all cores can run at up to 4.2 GHz. The CPU offers 24 MB of Level 3 cache and supports DDR4-3200 memory. The specifications of the Intel Core i7-11800H are shown in [6.1](#) and can also be found in

<https://ark.intel.com/content/www/us/en/ark/products/213803/intel-core-i711800h-processor-24m-cache-up-to-4-60-ghz.html>

Lithography	10 nm
Recommended Customer Price	435.00 USD
Total Cores	8
Total Threads	16
Max Clock Frequency	4.60 GHz
Cache	24 MB
Configurable TDP-up	2.30 GHz
Configurable TDP-down	1.90 GHz
Max Memory Bandwidth	51.2 GB/s

TABLE 6.1: Intel Core i7-11800H Processor specifications. [URL](https://ark.intel.com/content/www/us/en/ark/products/213803/intel-core-i711800h-processor-24m-cache-up-to-4-60-ghz.html)

6.2.2 NVIDIA RTX 3050 Ti Mobile

The GeForce RTX™ 3050 is built with the powerful graphics performance of the NVIDIA Ampere architecture. It offers dedicated 2nd gen RT Cores and 3rd gen Tensor Cores, new streaming multiprocessors, and high-speed G6 memory. The detailed specifications of the NVIDIA RTX 3050 Ti Mobile are shown in 6.2 and can also be found in <https://www.techpowerup.com/gpu-specs/geforce-rtx-3050-ti-mobile.c3778>.

Architecture	Ampere
Lithography	8 nm
Base Clock - Boost Clock	735 MHz - 1035 MHz
CUDA Cores	2560
Tensor Cores	80
Memory	4 GB GDDR6
Memory Bandwidth	192.0 GB/s
TDP	75 W

TABLE 6.2: NVIDIA RTX 3050 Ti Mobile specifications. [URL](https://www.techpowerup.com/gpu-specs/geforce-rtx-3050-ti-mobile.c3778)

6.2.3 ZCU102

An overview of the ZCU102 board specifications are presented in 5.1. A more in depth look at the logic components present in the PL fabric of the ZCU102, that can be found in the Vivado IDE implementation summary, are shown in 6.3.

CLB LUTs	274080
CLB Registers	548160
CARRY8	34260
F7 Muxes	137040
CLB	34260
LUT as Logic	274080
LUT as Memory	144000
Block RAM	912
DSPs	2520

TABLE 6.3: ZCU102 PL fabric specifications.

6.3 Implemented Accelerator Characteristics

The proposed accelerator's specifications as obtained by Viavdo IDE, after the implementation process, are shown in 6.4.

Clock Frequency	116 MHz
CLB LUTs	28386 (11%)
CLB Registers	17558 (3%)
CARRY8	1597 (5%)
F7 Muxes	35 (<1%)
CLB	6148 (18%)
LUT as Logic	26406 (10%)
LUT as Memory	1980 (1%)
Block RAM	122 (13%)
DSPs	1472 (58%)

TABLE 6.4: Static SLDA accelerator for fixed point operations specifications.

As can be deduced from the above results the static SLDA accelerator, alongside the DMAs and AXI interconnects, only covers a small percentage of the resources found on the ZCU102, except for the DSPs that play an essential role in performing the linear algebra operations. An interesting look into this, makes clear that multiple duplicates of the accelerator can be used to

achieve even greater throughput. Alas, in this thesis a single SLDA accelerator is being looked into for assessing the base performance.

The utilization of the floating point implementation is shown in 6.5, as it was reported from the Vitis HLS tool.

Clock Frequency	116 MHz
LUT	172364 (62%)
FF	303469 (55%)
DSP	646 (25%)
BRAM	1411 (77%)

TABLE 6.5: Static SLDA accelerator for floating point operations specifications.

A histogram that shows the difference utilization percentages between the fixed point implementation and the floating point implementation is illustrated in 6.1.

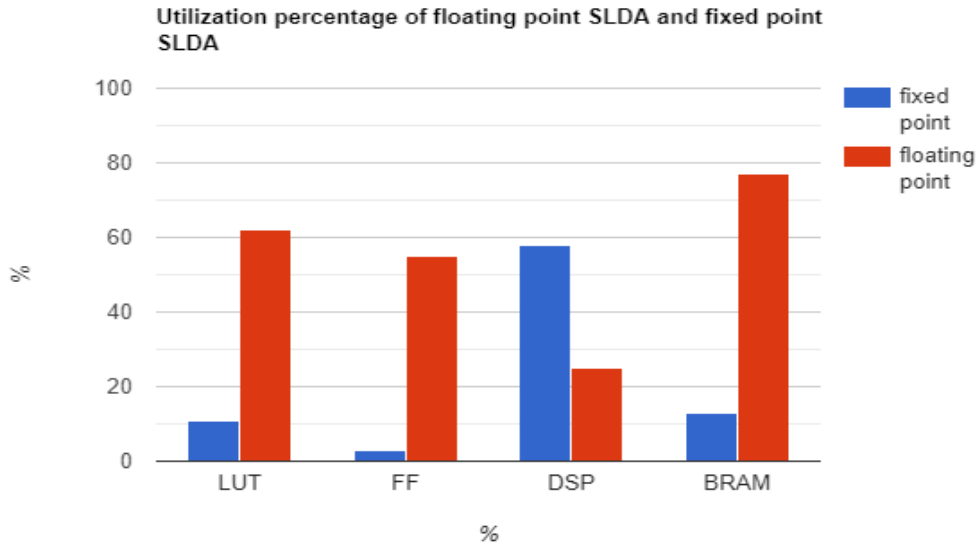


FIGURE 6.1: Utilization percentage histogram. The vertical axis measures the percentage of the resource that the design needs.

6.4 Experiments that were sought through

To confirm the efficiency and usability of the Static SLDA accelerator, and therefore other continual learning methods, on FPGAs a number of experiments were sought through both in comparing different hardware architectures (CPU, GPU) and different numerical precisions (floating point, 16-bit

fixed point).

For the latency, throughput, energy consumption and power consumption metrics:

On the Intel i7-11800H CPU, a naive implementation running on a single core was written in C++ Eigen Linear Algebra library, examining matrices with floating point data and matrices with 16-bit fixed point data. Along those lines, a Python NumPy implementation was written that utilizes all the cores of the CPU examining again the same data types.

On the NVIDIA RTX 3050 Ti, a naive implementation using CuBLAS without any memory transfer optimizations and kernel optimizations was written alongside a Python CuPy implementation that leverages all of the GPU's potential.

For the accuracy metrics on the CIFAR10, the ResNet-18 CNN was employed. The SLDA classifier's aptness was assessed for both floating point calculations and fixed point calculations.

6.5 Throughput and Latency Speedup

6.5.1 Amdahl's Law

Amdahl's law was first mentioned from Gene M. Amdahl in AFIPS spring joint computer conference, 1967[69][70]. It practically states that the performance gain in any task is limited from the fraction of the task that is serial. Mathematically this can be expressed as:

$$Speedup = \frac{Latency_{baseline}}{Latency_{new}} = \frac{1}{f + \frac{1-f}{P}}$$

, where:

- f : fraction of the task that is serial
- $1 - f$: fraction of the task that is parallelizable
- P : Number of processors

$$Speedup_{max} = \lim_{P \rightarrow \infty} Speedup = \frac{1}{f}$$

6.5.2 Metrics used for latency and throughput

Latency or *response time* in hardware systems is the amount of time it takes a module or a system to produce the corresponding result. In general, it is the time between the start and the completion of an event. In this work, latency is measured from the transferring of the feature vector to the device to the storing of the resulting scores in memory

In contrast, *Throughput* or *Bandwidth* is the amount of tasks that can be processed by the module or system in a given time. Usually throughput is measured per every second of work. In this work, two types of throughput are measured, the first being $\frac{\text{feature vectors}}{\text{second}}$, and the second being $\frac{\text{dot products}}{\text{second}}$. The second measurement is more widely useful as it can assess the capability of the SLDA accelerator (that is tailor made to perform continual learning) to perform more general matrix-matrix, matrix-vector and vector-vector operations.

6.5.3 Comparison of the accelerator to other platforms

GPU (CuPy)	CPU (NumPy) GPU (Naive)	CPU (Naive) FPGA (ZCU102)
Clock Frequency (MHz)	4200	4200
Latency 16-bit FxP	112 ms	24.7 ms
Latency FP32	477.2 μ s	11.08 ms
Latency Speedup 16-bit FxP	1x	4.93x
Latency Speedup FP32	1x	0.043x
Throughput1 16-bit FxP (feature vectors/s)	8.92	40.48
Throughput2 16-bit FxP (dot products/s)	45.848K	208.1K
Throughput 16-bit FxP Speedup	1x	4.53x
Throughput1 FP32 (feature vectors/s)	2096.4	90.25
Throughput2 FP32 (dot products/s)	10.77M	463.88K
Throughput FP32 Speedup	1x	0.043x

TABLE 6.6: Static SLDA classifier latency and throughput metrics on CPU with the use of NumPy and without the use of NumPy.

	CPU (NumPy)	GPU (CuPy)	GPU (Naive)	FPGA (ZCU102)
Clock Frequency (MHz)	4200	1035	1035	116
Latency 16-bit FxP	112 ms	1.35 ms	NA	50 μs
Latency FP32	477.2 μs	1.33 ms	8.1 ms	2.4 ms
Latency Speedup 16-bit FxP	1x	82.96x	NA	2240x
Latency Speedup FP32	1x	0.358x	0.058x	0.198x
Throughput1 16-bit FxP (feature vectors/s)	8.92	740.74	NA	20000
Throughput2 16-bit FxP (dot products/s)	45.848K	3.8M	NA	102.8M
Throughput 16-bit FxP Speedup	1x	83.04x	NA	2242x
Throughput1 FP32 (feature vectors/s)	2096.4	751.8	123.4	416.6
Throughput2 FP32 (dot products/s)	10.77M	3.86M	634.27K	2.14M
Throughput FP32 Speedup	1x	0.358x	0.058x	0.198x

TABLE 6.7: Static SLDA classifier latency and throughput metrics among the different platforms for feature vectors of dimensionality $N = 512$. These results are without the data transferring optimizations

The results shown in 6.7 indicate that the FPGA implementation of the Static SLDA algorithm outclasses the other platforms when it comes to Fixed Point operations, as it was expected. The GPU and CPU libraries and modern architectures all contain specialized units and processor instructions that optimize floating point operations. This becomes apparent when comparing the FP32 latency of CuPy and NumPy to the FPGA, which is not optimized for floating point operations.

It is worth mentioning though, that a large fraction of the time needed to execute the SLDA computations on the FPGA is taken by the transferring of the data of the Lambda and the Means matrix. When $N = 512$ the elements in the Lambda matrix are 262144, which dictates that at least 262144 clock cycles are needed for the FIFO to receive all the data.

To counteract this limitation, the Lambda and Means matrices are stored in local cache (BRAM) banks and blocks of data can be accessed in parallel.

After the initial data transfer the accelerator can operate by recycling the matrices, that means loading them from BRAM, processing them and storing them back. For iterative methods, such as some continual learning methods, the accelerator works best (6.8)

	CPU (NumPy)	GPU (CuPy)	GPU (Naive)	FPGA (ZCU102)
Clock Frequency (MHz)	4200	1035	1035	116
Latency 16-bit FxP	112 ms	1.35 ms	NA	15 μs
Latency FP32	477.2 μ s	1.33 ms	8.1 ms	310.89 μs
Latency Speedup 16-bit FxP	1x	82.96x	NA	7466x
Latency Speedup FP32	1x	0.358x	0.058x	1.534x
Throughput1 16-bit FxP (feature vectors/s)	8.92	740.74	NA	66666.6
Throughput2 16-bit FxP (dot products/s)	45.848K	3.8M	NA	342.66M
Throughput 16-bit FxP Speedup	1x	83.04x	NA	7473x
Throughput1 FP32 (feature vectors/s)	2096.4	751.8	123.4	3216.57
Throughput2 FP32 (dot products/s)	10.77M	3.86M	634.27K	16.53M
Latency Speedup FP32	1x	0.358x	0.058x	1.534x

TABLE 6.8: Static SLDA classifier latency and throughput metrics among the different platforms for feature vectors of dimensionality $N = 512$. These results include the data transferring optimizations

In 6.2 6.3 the latency and throughput of the different platforms are presented in the form of a bar plot accordingly. The y axis is in logarithmic scale to render the differences more easily visible.

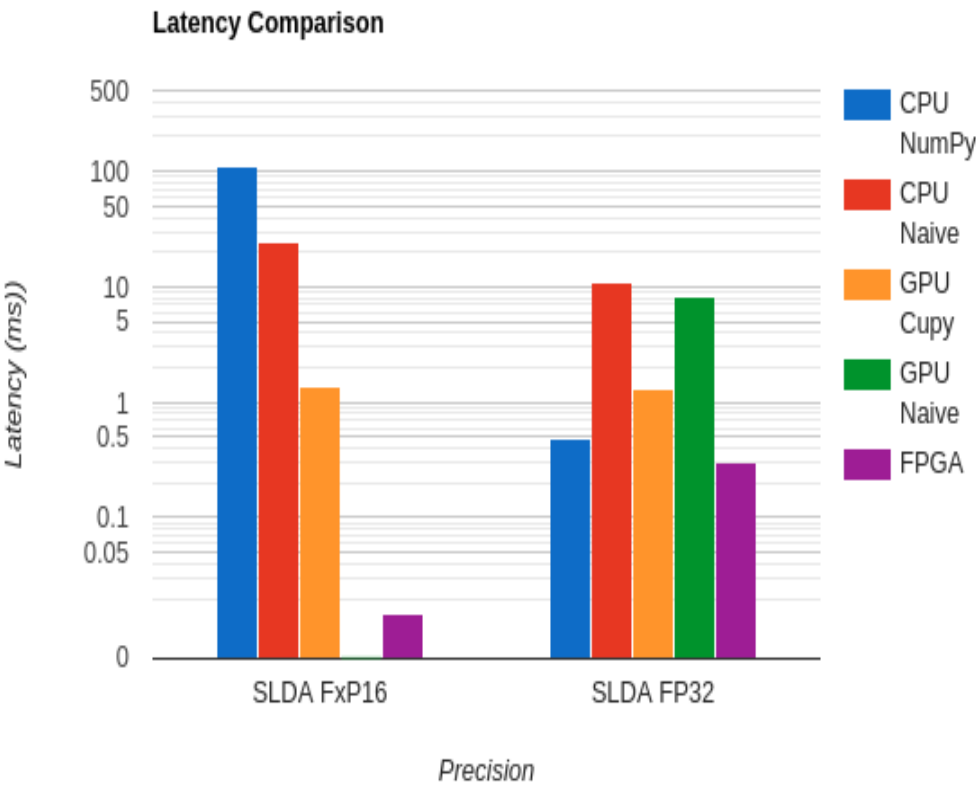


FIGURE 6.2: Static SLDA latency among different platforms.

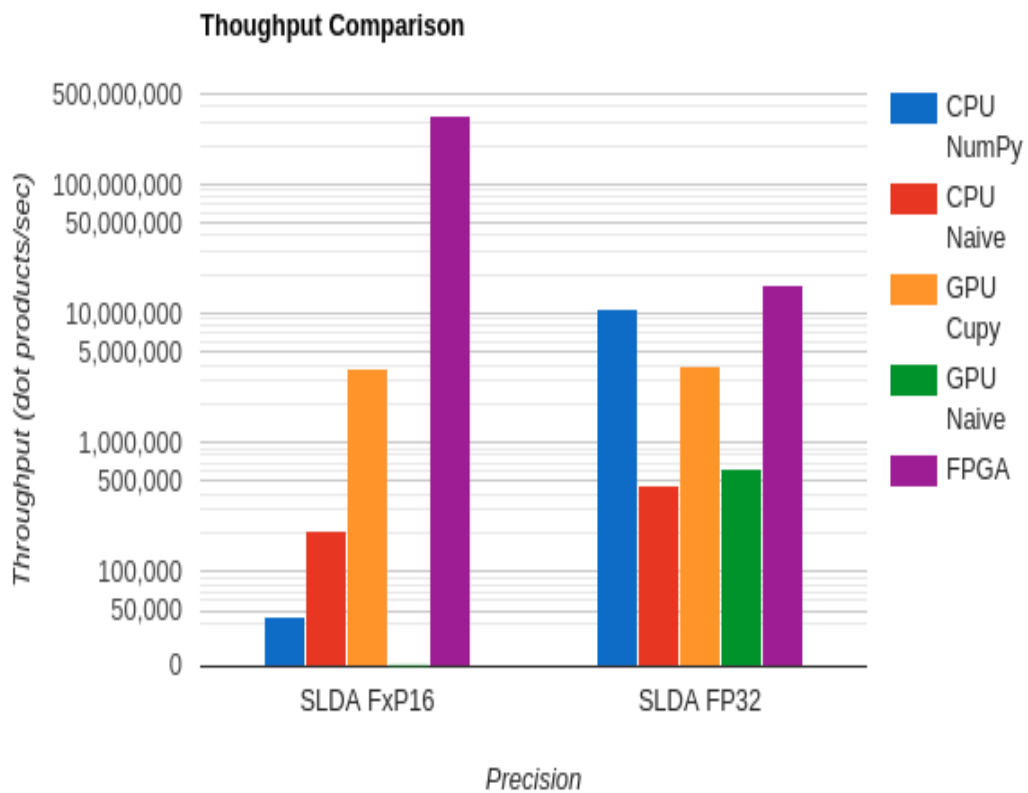


FIGURE 6.3: Static SLDA throughput among different platforms.

Going forward, an analysis on the speedup of the whole inference process is sought through. Amdahl's algorithm is ever present in this case and is used to compute the speedup, when taken into consideration that the first layers of the CNN are the fraction of the process that is not parallelized. On the other hand, the SLDA algorithm is the fraction that can become faster. The inference time of the CNN is measured to be $T = 5.5ms$ running on the NVIDIA RTX 3050 Ti GPU.

	CPU (NumPy)	CPU (Naive)	GPU (CuPy)	GPU (Naive)	FPGA (ZCU102)
Latency 16-bit FxP	117.5 ms	30.2 ms	6.85 ms	NA	5.51 ms
Latency FP32	5.97 ms	16.58 ms	6.83 ms	13.6 ms	5.81 ms
Latency Speedup 16-bit FxP	0.25x	1x	4.4x	NA	5.48x
Latency Speedup FP32	2.77x	1x	2.42x	1.21x	2.85x

TABLE 6.9: Latencies of the whole inference process + the training of the SLDA classifier

It becomes clear from table 6.9 that by accelerating the SLDA classifier on an FPGA the theoretical best speedup governed by Amdahl's law is best approximated. This leads to the realisation that an inference model can also learn new tasks without compromising its speed and power efficiency.

6.6 Power Consumption and Energy Consumption

In this section the power consumption of the different platforms is compared. Power in physics is the amount of energy transferred or converted per unit time. In other words it is the rate of change of the energy or work with respect to time, denoted as:

$$P = \frac{dW}{dt} = \frac{dE}{dt}(W)$$

. Power is measured in Watts. Average power is computed using the following formula

$$P_{avg} = \frac{\Delta W}{\Delta t} = \frac{\Delta E}{\Delta t}(W)$$

.

The average on-chip power for the different scenarios of the SLDA classifier are obtained from the *powertop* tool in the case of CPU implementation (naive and NumPy), the *nvtop* tool in the case of GPU implementation (naive and CuPy) and finally from the Vivado IDE in the case of FPGA implementation (ZCU102).

Alongside the power metrics, the energy efficiency of the different platforms are measured, in Joules, by using the following formula:

$$E = P_{avg} \cdot T = P_{avg} \cdot Latency_{avg}(\text{Joules})$$

, where

- E = average energy consumption
- P_{avg} = average power consumption
- T = total runtime
- $Latency_{avg}$ = The latency of SLDA, starting from the data transfers to the classifier and ending when the results are written into memory

On-Chip Power

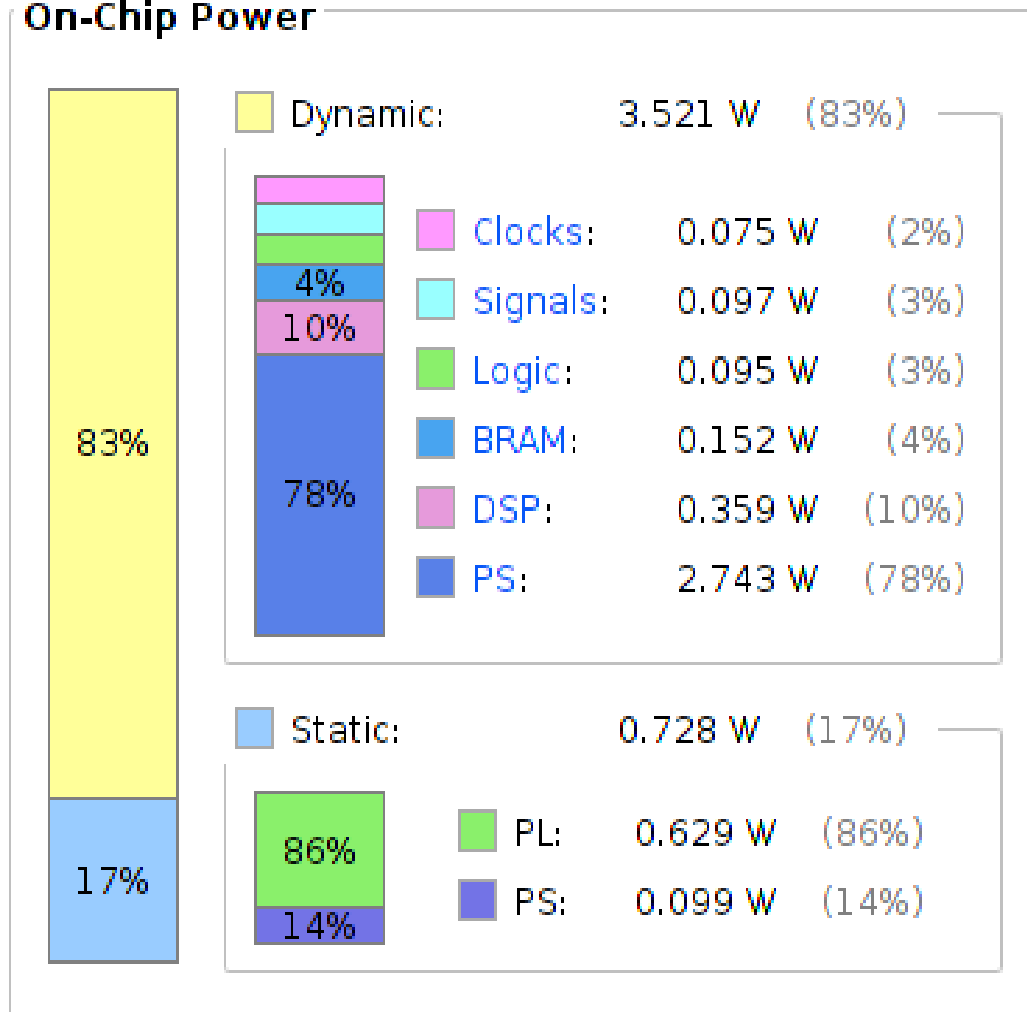


FIGURE 6.4: Static SLDA on ZCU102 total on-chip power.

	CPU (NumPy)	CPU (Naive)	GPU (CuPy)	GPU (Naive)	FPGA (ZCU102)
Clock Frequency (MHz)	4200	4200	1035	1035	116
Power Consumption (Watts)	19.2	19.2	25	25	7
Power Efficiency	1x	1x	0.76x	0.76x	2.74x
Energy Consumption (Joules)	0.009	0.212	0.033	0.202	0.002
Energy Efficiency	23.55x	1x	6.42x	1.04x	106x

TABLE 6.10: Static SLDA classifier power and energy metrics for floating point arithmetic.

	CPU (NumPy)	CPU (Naive)	GPU (CuPy)	GPU (Naive)	FPGA (ZCU102)
Clock Frequency (MHz)	4200	4200	1035	1035	116
Power Consumption (Watts)	19.2	19.2	25	25	4.25
Power Efficiency	1x	1x	0.76x	0.76x	4.51x
Energy Consumption (Joules)	2.150	0.474	0.033	NA	0.00006
Energy Efficiency	0.22x	1x	14.36x	NA	7900x

TABLE 6.11: Static SLDA classifier power and energy metrics for 16-bit Fixed Point arithmetic.

6.7 Accuracy Metrics

The accuracy metrics show the competency of the Static SLDA algorithm to learn new classes without updating the weights of the network. To measure the ability of the continual learning algorithm to learn new tasks the paradigm proposed by [48] is employed, which is the New Classes (NC) scenario with a full test dataset for validation. That means, in every epoch of training only new classes are shown to the network. After the training of the classifier it is then tested on the full test dataset containing samples from all the classes, even unseen ones.

The dataset of choice is split-CIFAR10, which is split in two different forms:

- The first form includes only the first two classes of CIFAR10 at the base initialization step (see [37]) and at every epoch a new pair of classes is presented to the network without changing the weights of it.
- The second form includes the first four classes of CIFAR10 as to have better representation of the feature space of the dataset, at the base initialization step. At every epoch a pair of new classes is presented except for the last epoch where only already seen classes are presented.

In 6.5 it can be observed that by initializing the network with more classes the learning process is faster and results in better final accuracy. The final

accuracy for the first split is 42.94% and for the second split 44.27%

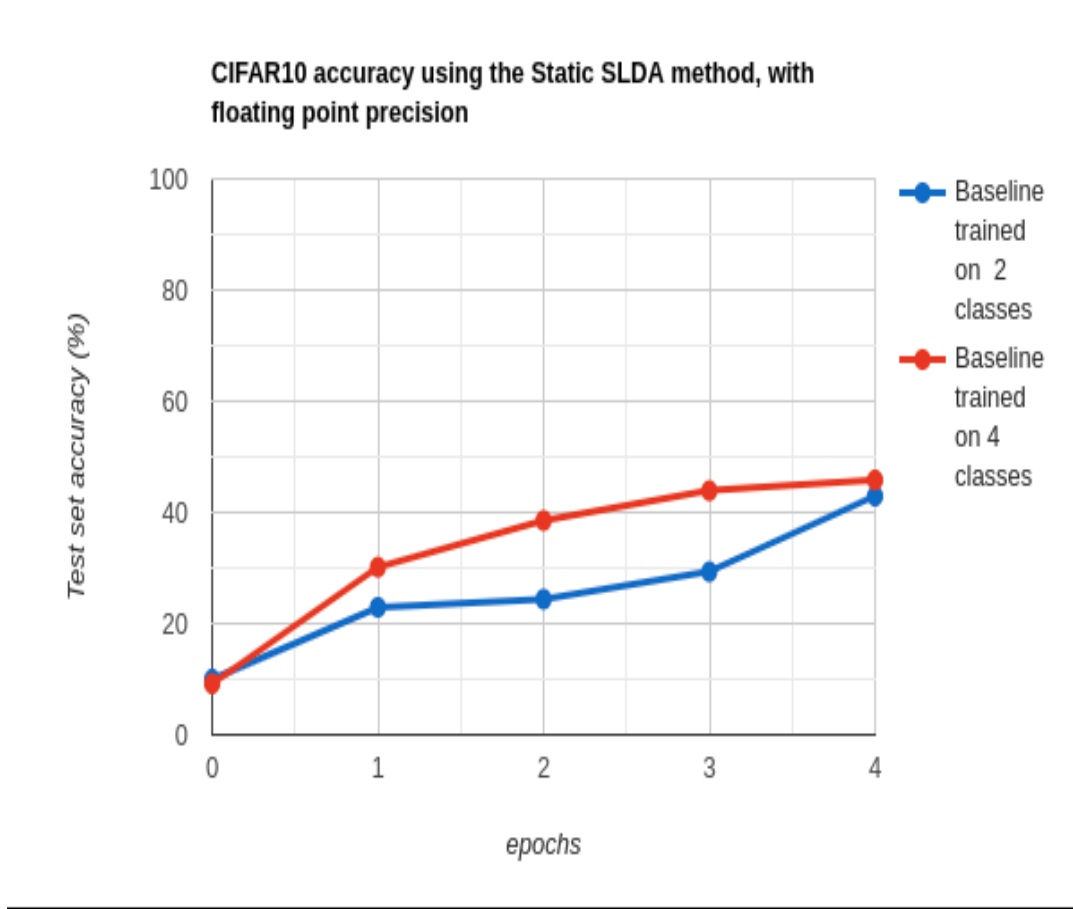


FIGURE 6.5: Accuracy metrics of the continual learning process on two different splits of the CIFAR10 dataset.

The accuracy results show the ability of the SLDA classifier to learn new classes of data without compromising the previously learned information. From a measly 10% accuracy at the first epoch it reaches almost 50% accuracy when all the classes are learned.

It is important to state that these accuracy results can be achieved in both floating point and fixed point precision, with a small difference in actual accuracy gain over the epochs. By choosing the precision of the fixed point numbers correctly, there is no important information loss, and the SLDA accelerator can still classify the images with good accuracy. This observation ultimately means that the best latency provided by the fixed point SLDA implementation can be leveraged to produce accurate results extremely fast.

6.8 Results Discussion

The evaluation of the aforementioned results show the advantages, both in latency and power efficiency, of a fully FPGA-based design of the Static SLDA algorithm. The FPGA accelerator in the worst-case scenario (floating point without data movement optimizations) is comparable to state of the art Hardware (CPU, GPU) and APIs (NumPy, CuPy). In the best case scenario it achieves latency and energy efficiency that is two orders of magnitude better than the CPU and GPU counterparts.

This can be explained by examining two major factors that are in play here.

First of all the disaggregated memory of the FPGA, that is found in small banks, is on a level that even high speed GDDR6 GPU memory cannot surpass. By sacrificing the clock frequency of the accelerator the parallelism that can be achieved is immense. If more parallelism is needed from a user, it can be accomplished by loading even more data from BRAM in one clock cycle, by widening the data. This of course comes with a high demand in resources.

Secondly, the matrix sizes in this kind of application is not very large and cannot utilize the high bandwidth GPU memory or the multiple CUDA cores that are present. In streaming continual learning the computations must be performed on the fly without leaving open the possibility to first gather all the feature vectors and labels and perform a batched Matrix-Matrix multiplication on the GPU. In batch continual learning theoretically a modern GPU can achieve the same performance as in streaming learning, but in this case the FPGA will fall quickly behind due to its limited resources. That said, a more modern FPGA that includes a lot more resources or a QFDB[71], that contains four ZCU102 on a chip, may be able to surpass even these limitations when met with very large matrix sizes.

Moreover the systolic arrays used in the architecture can perform a dot product calculation in one clock cycle, without having to perform multiple load-stores in memory.

On the power and energy efficiency front, the FPGA implementation again takes the lead as the ZCU102 consumes a lot less energy than the 8-core 16-thread CPU and the RTX3050 Ti GPU. As can be seen in 6.4 a large percentage of the total on-chip power is on the PS side of the board. Still the ARM 4-core embedded processor is not that power hungry as it is expected from this kind

of board. The PL side is very power efficient due to many factors, some of which are:

- No Instruction Fetch Overhead
- Reduced Data Movement, as the data are closer to the processing units
- Low leakage power
- Specialized hardware blocks such as DSPs, BRAMs etc.

Compared to the work of [52] both the Weights and biases of SLDA are computed on the FPGA and not externally. This adds some latency and resources to the developed accelerator. Thus, a fair comparison to the authors' platform is not possible, as most of the computational capacity of the FPGA is possessed by the Weights Compute Unit.

Chapter 7

Conclusions and Future Work

This chapter sums up the thesis and suggests some ways it could be expanded in a way that can benefit both the scientific research domain and the industry standard AI accelerators, that become more and more popular.

7.1 Conclusions

In summary, Artificial Intelligence has been all around us in the last few years, having generated a vast amount of research topics that are looked into by both academics and industries alike. The powerful capabilities that Machine Learning models have acquired over the last few years are admirable and have pushed forward many applications such as autonomous driving, text generation, speech generation, graphics performance acceleration and medical analysis. This revolution of ML models owes a lot to the advancements in hardware technology that produces more and more capable systems spanning from NVIDIA high-end GPUs to the Google TPU, that specifically accelerates AI, to the Apple M1 and the rumored meteor lake intel CPU, that are going to include an embedded system on chip that will be used for linear algebra acceleration.

In this work, the focus was given to continual learning applications that can benefit real time edge inference models without any intervention by a human being, such as unmanned flight, space exploration, ocean exploration and more. The accelerator that was developed on the ZCU102 can be used as a standalone IP that can be implemented in any other FPGA platform, as it is a complete classifier module that does not need any external processing unit (GPU or CPU) to produce the results needed. By leveraging the parallelism and low memory footprint of a systolic array, as well as the memory partitioning of BRAMs and the specialized DSPs provided, the accelerator

can outperform state of the art CPU and GPU platforms, when it comes to throughput and latency, for a fraction of the power and energy consumption. It is also worth mentioning that the IP core that was developed may be proven useful in many other applications where matrix multiplication is needed, especially when it comes to low precision arithmetic, where FPGA acceleration truly shines.

In the next section some possible next steps for the hardware design and for some more general use cases of the accelerator are presented.

7.2 Future Work

Due to the limited amount of time required for the completion of this thesis some ideas were left out of the examination that was performed. Nevertheless, some of them are going to be listed here for further exploration on the subject for whoever finds interest in it.

- **Complete inference model running on the FPGA:** Through the usage of the Deep Learning Processing Unit (DPU) and the Vitis-AI framework provided by AMD-Xilinx a complete inference model running only on the FPGA without any assistance from a CPU or GPU could be developed. The DPU over the last two years has been greatly advanced and includes state of the art models such as CNNs, YOLOv5, transformers and more. By quantizing the model and pruning it, it can fit inside the FPGA alongside the SLDA accelerator performing inference and very fast speeds and extremely low power. When on the edge a platform that is power efficient is crucial to the success of the mission.
- **Testing performed on more modern Machine Learning models and datasets:** An interesting direction that this thesis can follow is to try to apply continual learning, and more specifically SLDA, to modern ML models like the YOLOv5 for object detection or Transformers and other generative models. Alongside these models more recent datasets and datasets that are more tailor made for continual learning, such as the CORE50 dataset [48], can be and should be examined. This study can also provide data on the scalability of the SLDA accelerator as the feature vectors get larger.

- **Assessment of more streaming learning algorithms that run on FPGAs:** Besides the Static SLDA method more continual learning methods running on FPGAs should be examined as they have the potential to be accelerated and produce similar, if not better, results than the SLDA implementation [33]. A few of these methods that retain a fixed network are: PathNet[72], PackNet[73], Piggyback[74], HAT[75].
- **Incorporate a bigger accelerator or multiple smaller accelerators on the QFDB[71]:** As it was described in chapter 6 a lot more dot product operations per second can be performed if multiple duplicates of the accelerator could fit on the ZCU102 or if the systolic array grid was larger, meaning more resource hungry. These limitations could be surpassed if an implementation using the Quad Daughter FPGA Board or a more contemporary version of the ZCU102 was developed. The parallelism can be increased by widening the data in the BRAMs even more than 128-bit that is currently now. Furthermore, if the grid of systolic arrays increases in size the latency needed to compute a matrix multiplication will decrease.
- **Inspect the Vitis HLS directives further:** Some of the directives of Vitis HLS that were not used in this implementation may prove to be useful in decreasing latency, resource utilization or both.
- **Test FPGAs that connect through PCIe to a conventional CPU or GPU:** The IP core that is developed can be tweaked to receive data through PCIe connection and thus be able to connect to some more powerful server CPUs or GPUs, leaving behind the limitations that come from the 4-core ARM edge CPU on the ZCU102. This way continual learning can be used in mainstream applications, and not be limited to the ZCU102 users.
- **Inspect scalability of the accelerator as a matrix multiplication unit:** The Static SLDA accelerator was made to perform the specific computations needed for the algorithm to make correct predictions. These computations are heavy on dot product calculations leading to the realisation that it can be used for General matrix-matrix and matrix-vector multiplications. Assume that $C = AB$ is the matrix multiplication of matrices $A \in \mathbb{R}^{N \times N}$ and $B \in \mathbb{R}^{N \times M}$ while $D = Cx$ is the multiplication of the matrix $C \in \mathbb{R}^{N \times M}$ with the vector $x \in \mathbb{R}^{M \times 1}$. In this thesis $N=512$ and $M=10$, which is small in size compared to what a contemporary GPU can handle. For $N \in \{1024, 2048, \dots, 65536, \dots\}$

and $M \in [10, 1000]$ the dot products needed to be computed are orders of magnitude larger in size and amount. Theoretically speaking, if the QFDB was to be used the matrices of $N=2048$ could be computed with the same latency (as with $N=512$, $L=15 \mu s$) and 4x the throughput (at $\tilde{1.37}$ Giga dot products/second = $\tilde{5.61}$ TIOPS(Integer Operations per Second)) meaning that for larger matrix sizes the accelerator may perform on par with high end GPUs.

- **Iterative methods that take advantage of matrix multiplications should be examined:** Iterative mathematical methods such as the Gauss-Seidel and Jacobi methods for solving systems of linear equations, the Power Iteration method for finding the dominant eigenvector and eigenvalue of a square matrix and the PageRank algorithm all can benefit from the recycling of matrices in the FPGA without the overhead of having to transport new matrices every time from the host to the FPGA. These methods can be accelerated to their fullest as seen by the latencies in table 6.8. If the recycling of matrices is not possible then the latencies from table 6.7 are more realistic.

Chapter 8

Appendix A - FPGA design from High Level Synthesis to Application level testing

8.1 Vitis HLS algorithms

The whole SLDA accelerator was developed using the tools provided by Vitis HLS. Some of the algorithms that were used to produce the desired hardware architecture are shown below.

8.1.1 Compute Weights Unit Algorithm

First and foremost, the algorithm that produces the Compute Weights Unit is displayed. The directives **pragma HLS partition** initially inform the compiler to divide the 2-dimensional matrices, stored in a big BRAM, into rows of matrix A and columns of matrix B (small blocks of BRAM). This way all the rows and all the columns can be accessed and processed simultaneously. Continuing, there are five encapsulated for loops, where the first two are pipelined with Initiation Interval 1. That means that every clock cycle a row and a column are loaded and begin processing without waiting for the first result to be computed. Following this, the next loops traverse the elements of a row and a column, in a blocking manner which enables the user to adjust for better latency or better utilization. Each block element contains 16 8-bit integers that are accessed in parallel as the loop gets unrolled using the directive **pragma HLS unroll**. As can be deduced from listing 8.1 the module can be customized by using the PIPELINE and UNROLL directives in different loops.

```

2 void compute_weights_with_matrix_mult(ap_int<DATA_WIDTH> A[N][N
    ], ap_int<DATA_WIDTH> B[N][M]) {
3
4     #pragma HLS ARRAY_PARTITION variable=A block factor=32 dim=2
5     #pragma HLS array_partition variable=B block factor=32 dim=1
6     #pragma HLS ARRAY_PARTITION variable=W block factor=32 dim=2
7     //#pragma HLS array_partition variable=C block factor=32 dim
=1
8
9     // Perform matrix multiplication
10 WEIGHTS_LOOP:
11     for(int i = 0; i < M; i++) { //Iterate columns of B
12         for(int j = 0; j < N; j++) { //Iterate rows of A
13 #pragma HLS PIPELINE II=1
14         W[i][j] = 0;
15         ap_int<DATA_WIDTH> temp = 0; //Used to store partial
results from the systolic arrays
16         for(int k = 0; k < N; k+=P) { //Determines how many
blocks are accessed in parallel
17             for(int p = 0; p < P; p++) {//Process the blocks
in parallel using unrolling
18             #pragma HLS unroll factor=32
19                 int x1[P] = {0};
20                 int x2[P] = {7,7,7,7,7,7,7,7,7,
21                     7,7,7,7,7,7,7,7,7,
22                     7,7,7,7,7,7,7,7,7,
23                     7,7,7,7,7,7,7,7,7};
24                 for(int l = 0; l < 16; l++){//process all 16
elements in the 128bit vector
25                     temp += ap_ufixed<8, 4, AP_RND_CONV, AP_WRAP>(
A[j][k+p].range(x2[p], x1[p])) * ap_ufixed<8, 4, AP_RND_CONV
, AP_WRAP>(B[k+p][i].range(x2[p], x1[p]));
26 #ifndef __SYNTHESIS__
27                     std::cout << "W[" << i << "][" << j << "]" : "
<< std::fixed;
28                     std::cout << ap_ufixed<8, 4, AP_RND_CONV,
AP_WRAP>(temp.range(x2[p],x1[p])) << std::endl;
29 #endif
30                     temp << 8;
31                     x1[p] += 8;
32                     x2[p] += 8;
33                 }
34             }
35         }
36         W[i][j] = temp; //Write the output vector in BRAM
37     }

```

```

38     }
39 }

```

LISTING 8.1: Compute Weights Unit code in Vitis HLS

8.1.2 Compute Biases Unit Algorithm

The Compute Biases Unit uses the same directives as the aforementioned unit to fragment the matrices into smaller arrays. The difference lies in the operations that take place inside of the loops. Each element of the bias vector is calculated by taking the dot product of the mean of each class with the corresponding column of the Weight matrix. This essentially renders this unit as a dot product calculation unit and can perform multiple dot product calculations in parallel. The code in listing 8.2 shows how the dot product calculations are performed.

```

1 BIASES_LOOP:
2     for(int j = 0; j < M; j++) {//Iterate columns of D and rows
        of W
3 #pragma HLS PIPELINE II=1
4         //b[j] = 0;
5         ap_int<32> temp = 0; //Used to store partial results from
            the systolic arrays
6         for(int k = 0; k < N; k+=P) {//Determines how many
            blocks are accessed in parallel
7             for(int p = 0; p < P; p++) {//Process the blocks in
                parallel using unrolling
8 #pragma HLS unroll factor=32
9                 int x1 = 0;
10                int x2 = 7;
11                for(int l = 0; l < 16; l++){//process all 16 elements in the
                    128bit vector
12                    temp += ap_ufixed<8, 4, AP_RND_CONV, AP_SAT_SYM>(W[j][k+p].
                        range(x2, x1)) * ap_ufixed<8, 4, AP_RND_CONV, AP_SAT_SYM>(D[
                            k+p][j].range(x2, x1));
13                    x1 += 8;
14                    x2 += 8;
15                }
16            }
17        }
18        b[j] = temp;
19    }

```

LISTING 8.2: Compute Biases Unit code in Vitis HLS

8.1.3 Compute Scores Unit Algorithm

The Compute Scores Unit follows the same logic as the Compute Biases Unit to compute the matrix vector multiplication of the linear classifier $W \cdot x_t$. At first, the feature vector elements, that are used to perform the $W \cdot x_t$ calculation, are loaded sequentially from a FIFO. At the end of each dot product calculation the corresponding bias element is added to the result.

```

1 [language=C++, caption=Compute Scores Unit code in Vitis HLS,
   label=lst:Compute_Scores_Code]
2 ap_int<DATA_WIDTH> fv_data[N];
3
4 for(int k = 0; k < N; k++){
5   fv_data[k] = feature_vector.read().data;
6 }
7
8 SCORES_LOOP:
9   for(int j = 0; j < M; j++) {
10    #pragma HLS PIPELINE II=2
11    //b[j] = 0;
12    ap_int<32> temp = 0;
13    for(int k = 0; k < N; k+=P) {
14      #pragma HLS unroll factor=32
15      for(int p = 0; p < P; p++) {
16        int x1 = 0;
17        int x2 = 7;
18        for(int l = 0; l < 16; l++){
19          temp += ap_ufixed<8, 4, AP_RND_CONV, AP_SAT_SYM>(W[j][k+p].
            range(x2, x1)) * ap_ufixed<8, 4, AP_RND_CONV, AP_SAT_SYM>(
              fv_data[k+p].range(x2, x1));
20          x1 += 8;
21          x2 += 8;
22        }
23      }
24    }
25    scores[j] = temp + b[j];
26  }

```

8.2 Vivado IDE block design

The next step into running the accelerator on the targeted platform is to use the Vivado IDE to connect the Vitis HLS module to the PS system. The four inputs of the final SLDA implementation are loaded directly from external

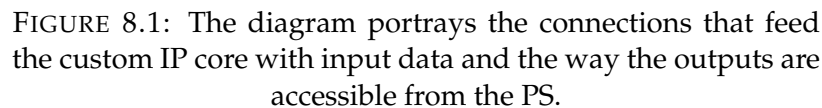
memory with the use of four DMAs that are configured to perform only read operations on data with 128 bit width.

The control signals and the output that is stored in BRAM communicate with the PS side through an AXI interconnect that connects the corresponding ports to the Master ports of the Zynq Ultrascale+.

The DMAs are controlled from the Master ports of the Zynq Ultrascale+ while they load the data from the slave ports of the Zynq Ultrascale+. They are also clocked at the same frequency as the PL clock.

The clock of the SLDA module is the same as the PL clock that is configured to be set at 115 MHz. The reset comes from an external reset generator and is common with the ZCU102 reset.

All the discussed connections are presented in the block diagram of figure 8.1.



First of all, before running the system the SLDA module and the DMAs must be initialized.

```
1 int status;
2 //Initialize custom made accelerator
3 ip_xconfig = XSLda_final_LookupConfig(
4     XPAR_XSLDA_FINAL_0_DEVICE_ID);
```

```

4 if(ip_xconfig == NULL){
5     printf("Failed to initialize device slda_final with id: %d",
        XPAR_XSLDA_FINAL_0_DEVICE_ID);
6     return -1;
7 }
8
9 status = XSlda_final_CfgInitialize(&ip_instance, ip_xconfig);
10 if(status != XST_SUCCESS){
11     printf("Failed to initialize device slda_final with id: %d",
        XPAR_XSLDA_FINAL_0_DEVICE_ID);
12     return -2;
13 }

```

LISTING 8.3: SLDA initialization in Vitis

The initialization of a DMA module is shown in listing 8.4

```

1 //Initialize dma 0
2 dma_xconfig_0 = XAxiDma_LookupConfig(XPAR_AXIDMA_0_DEVICE_ID);
3 if(dma_xconfig_0 == NULL){
4     printf("Failed to initialize device axi dma 0 with id: %d",
        XPAR_AXIDMA_0_DEVICE_ID);
5     return -3;
6 }
7
8 status = XAxiDma_CfgInitialize(&dma_instance_0, dma_xconfig_0);
9 if(status != XST_SUCCESS){
10     printf("Failed to initialize device axi dma 0 with id: %d",
        XPAR_AXIDMA_0_DEVICE_ID);
11     return -4;
12 }
13
14 XAxiDma_IntrDisable(&dma_instance_0, XAXIDMA_IRQ_ALL_MASK,
        XAXIDMA_DMA_TO_DEVICE);
15 XAxiDma_IntrDisable(&dma_instance_0, XAXIDMA_IRQ_ALL_MASK,
        XAXIDMA_DEVICE_TO_DMA);

```

LISTING 8.4: DMA initialization in Vitis

The main function of the application first initializes the DDR memory space of the four DMAs with the corresponding data, that are: the means of the classes, the two duplicate feature vectors (for achieving greater levels of parallelism) and the Lambda matrix. The data type and the amount of data to be transeferred depend on the scenario that is being run at the moment.

After that, the program falls into an infinite loop that transfers data through the DMAs, after the cache is flushed for coherency, then waits for the accelerator to complete processing the data before outputting the final scores to the console. For measuring the latency of the accelerator a global timer of the PSU starts before the data transfers and stops after the final result is stored in RAM.

All of the above are shown in 8.5

```

1  int main(){
2
3  //Initialize the platform
4  init_platform();
5
6  //Initialize IPs
7  init_IPs();
8
9  int8_t* axi_dma_tx_0 = (int8_t*) TX_DATA_BASE_ADDR_0;
10 int8_t* axi_dma_tx_1 = (int8_t*) TX_DATA_BASE_ADDR_1;
11 int8_t* axi_dma_tx_2 = (int8_t*) TX_DATA_BASE_ADDR_2;
12 int8_t* axi_dma_tx_3 = (int8_t*) TX_DATA_BASE_ADDR_3;
13 //timer_t timer;
14 XTime start, startHW, end, endHW;
15
16 int counter = 0;
17 int8_t constant1 = 0x01;
18 int8_t constant2 = 0x02;
19
20 //Initialize the mean matrix data
21 for(int i = 0; i < 640; i+=2){
22     axi_dma_tx_0[i] = constant1;
23     axi_dma_tx_0[i+1] = constant2 ;
24 }
25
26 //There are 2 feature vectors so that predictions and updates
    can be performed independently
27 //feature vector used for updating means
28 for(int i = 0; i < 64; i+=2){
29     axi_dma_tx_1[i] = constant1 + i;
30     axi_dma_tx_1[i+1] = constant2 + i;
31 }
32
33 //feature vector used for the prediction
34 for(int i = 0; i < 64; i+=2){
35     axi_dma_tx_2[i] = constant1;
36     axi_dma_tx_2[i+1] = constant2;

```

```

37 }
38
39
40 //Initialize the Lambda matrix data
41 for(int i = 0; i < 2048; i+=2){
42     axi_dma_tx_3[i] = constant1 + i;
43     axi_dma_tx_3[i+1] = constant2 + i;
44 }
45
46 while(1){
47
48     XTime_GetTime(&start);
49     XSlda_final_Start(&ip_instance);
50
51     Xil_DCacheFlushRange((INTPTR)axi_dma_tx_0, sizeof(int8_t)
        *5120);
52     Xil_DCacheFlushRange((INTPTR)axi_dma_tx_1, sizeof(int8_t)*512)
        ;
53     Xil_DCacheFlushRange((INTPTR)axi_dma_tx_2, sizeof(int8_t)*512)
        ;
54     Xil_DCacheFlushRange((INTPTR)axi_dma_tx_3, sizeof(int8_t)
        *262144);
55     //Xil_DCacheFlushRange((INTPTR)axi_dma_rx_0, sizeof(int64_t)*
        ELEMENTS_TO_RECEIVE);
56     printf("Send data to the ip core via dma");
57     XTime_GetTime(&startHW);
58     XAxiDma_SimpleTransfer(&dma_instance_0, (UINTPTR)axi_dma_tx_0,
        sizeof(int8_t)*5120, XAXIDMA_DMA_TO_DEVICE);
59     XAxiDma_SimpleTransfer(&dma_instance_1, (UINTPTR)axi_dma_tx_1,
        sizeof(int8_t)*512, XAXIDMA_DMA_TO_DEVICE);
60     XAxiDma_SimpleTransfer(&dma_instance_2, (UINTPTR)axi_dma_tx_2,
        sizeof(int8_t)*512, XAXIDMA_DMA_TO_DEVICE);
61     XAxiDma_SimpleTransfer(&dma_instance_3, (UINTPTR)axi_dma_tx_3,
        sizeof(int8_t)*262144, XAXIDMA_DMA_TO_DEVICE);
62
63
64 /* printf("Receive data from the dma");
65     XAxiDma_SimpleTransfer(&dma_instance, (UINTPTR)axi_dma_rx,
        sizeof(int32_t)*ELEMENTS_TO_RECEIVE, XAXIDMA_DEVICE_TO_DMA);
66     while(XAxiDma_Busy(&dma_instance, XAXIDMA_DEVICE_TO_DMA));
67     XTime_GetTime(&endHW);
68     Xil_DCacheInvalidateRange((INTPTR)axi_dma_rx, sizeof(int32_t)*
        ELEMENTS_TO_RECEIVE);*/
69
70     while(!XSlda_final_IsDone(&ip_instance));
71     XTime_GetTime(&endHW);

```

```
72
73   XTime_GetTime(&end);
74
75   printf("Time in microseconds for app execution: %llu\n", 2*(
       end-start));
76   printf("app execution took %.2f us.\n",
77         1.0 * (end - start) / (COUNTS_PER_SECOND/1000000));
78
79   printf("Time in microseconds for hw execution: %llu\n", 2*(
       endHW-startHW));
80   printf("Hw execution took %.2f us.\n",
81         1.0 * (endHW - startHW) / (COUNTS_PER_SECOND
       /1000000));
82
83   for(int k = 0; k < ELEMENTS_TO_RECEIVE; k++){
84       printf("scores[%d]: %d\n", k, scores[k]);
85   }
86 }
87 return 0;
88 }
```

LISTING 8.5: Main function

References

- [1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010.
- [2] 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](https://doi.org/10.1147/rd.33.0210).
- [12] Warren Mcculloch and Walter Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 127–147.
- [21] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536. URL: <https://api.semanticscholar.org/CorpusID:205001834>.
- [27] Kunihiro Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36 (1980), pp. 193–202. URL: <https://api.semanticscholar.org/CorpusID:206775608>.
- [28] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: [10.1109/5.726791](https://doi.org/10.1109/5.726791).
- [30] Michael McCloskey and Neal J. Cohen. "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem". In: *Psychology of Learning and Motivation* 24 (1989). Ed. by Gordon H. Bower, pp. 109–165. ISSN: 0079-7421. DOI: [https://doi.org/10.1016/S0079-7421\(08\)60536-8](https://doi.org/10.1016/S0079-7421(08)60536-8). URL: <https://www.sciencedirect.com/science/article/pii/S0079742108605368>.
- [32] James Kirkpatrick et al. "Overcoming catastrophic forgetting in neural networks". In: *Proceedings of the national academy of sciences* 114.13 (2017), pp. 3521–3526.
- [33] Matthias De Lange et al. "A continual learning survey: Defying forgetting in classification tasks". In: *IEEE transactions on pattern analysis and machine intelligence* 44.7 (2021), pp. 3366–3385.

- [34] Zhizhong Li and Derek Hoiem. “Learning without Forgetting”. In: (2017). arXiv: [1606.09282 \[cs.CV\]](#).
- [35] David Lopez-Paz and Marc’Aurelio Ranzato. “Gradient Episodic Memory for Continual Learning”. In: (2022). arXiv: [1706.08840 \[cs.LG\]](#).
- [36] Shaoning Pang, S. Ozawa, and N. Kasabov. “Incremental linear discriminant analysis for classification of data streams”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 35.5 (2005), pp. 905–914. DOI: [10.1109/TSMCB.2005.847744](#).
- [37] Tyler L. Hayes and Christopher Kanan. *Lifelong Machine Learning with Deep Streaming Linear Discriminant Analysis*. 2020. arXiv: [1909.01520 \[cs.LG\]](#).
- [38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet classification with deep convolutional neural networks”. In: *Communications of the ACM* 60.6 (2017), pp. 84–90.
- [40] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).
- [42] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.
- [45] Gao Huang et al. “Densely Connected Convolutional Networks”. In: (2017), pp. 2261–2269. DOI: [10.1109/CVPR.2017.243](#).
- [46] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: (2017). arXiv: [1704.04861 \[cs.CV\]](#).
- [48] Vincenzo Lomonaco and Davide Maltoni. “COPe50: a New Dataset and Benchmark for Continuous Object Recognition”. In: (2017). arXiv: [1705.03550 \[cs.CV\]](#).
- [52] Duvindu Piyasena, Siew-Kei Lam, and Meiqing Wu. “Accelerating continual learning on edge fpga”. In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2021, pp. 294–300.
- [53] Norman P Jouppi et al. “In-datacenter performance analysis of a tensor processing unit”. In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.
- [56] Xishan Zhang et al. “Fixed-point back-propagation training”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 2330–2338.

- [57] Yilun Chen et al. “Shrinkage Algorithms for MMSE Covariance Estimation”. In: *IEEE Transactions on Signal Processing* 58.10 (Oct. 2010), pp. 5016–5029. DOI: [10.1109/tsp.2010.2053029](https://doi.org/10.1109/tsp.2010.2053029). URL: <https://doi.org/10.1109%2Ftsp.2010.2053029>.
- [69] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California”. In: *IEEE Solid-State Circuits Society Newsletter* 12.3 (2007), pp. 19–20. DOI: [10.1109/N-SSC.2007.4785615](https://doi.org/10.1109/N-SSC.2007.4785615).
- [71] Khoa Pham et al. “Moving Compute towards Data in Heterogeneous multi-FPGA Clusters using Partial Reconfiguration and I/O Virtualisation”. In: (2020), pp. 221–226. DOI: [10.1109/ICFPT51103.2020.00038](https://doi.org/10.1109/ICFPT51103.2020.00038).
- [72] Chrisantha Fernando et al. “PathNet: Evolution Channels Gradient Descent in Super Neural Networks”. In: (2017). arXiv: [1701.08734](https://arxiv.org/abs/1701.08734) [cs.NE].
- [73] Arun Mallya and Svetlana Lazebnik. “PackNet: Adding Multiple Tasks to a Single Network by Iterative Pruning”. In: (2018). arXiv: [1711.05769](https://arxiv.org/abs/1711.05769) [cs.CV].
- [74] Arun Mallya, Dillon Davis, and Svetlana Lazebnik. “Piggyback: Adapting a Single Network to Multiple Tasks by Learning to Mask Weights”. In: (2018). arXiv: [1801.06519](https://arxiv.org/abs/1801.06519) [cs.CV].
- [75] Joan Serrà et al. “Overcoming catastrophic forgetting with hard attention to the task”. In: (2018). arXiv: [1801.01423](https://arxiv.org/abs/1801.01423) [cs.LG].

External Links

- [3] *Machine Learning Explained*. URL: <https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained>.
- [4] *AI vs ML*. URL: <https://cloud.google.com/learn/artificial-intelligence-vs-machine-learning>.
- [5] *Mathworks ML*. URL: <https://www.mathworks.com/discovery/machine-learning.html>.
- [6] *Mathworks Supervised*. URL: <https://www.mathworks.com/discovery/supervised-learning.html>.
- [7] *Stanford Reinforcement Learning*. URL: <https://web.stanford.edu/class/cs234/>.
- [8] *Deep Learning*. URL: https://en.wikipedia.org/wiki/Deep_learning.
- [9] *Mathworks Deep Learning*. URL: <https://www.mathworks.com/discovery/deep-learning.html>.
- [10] *IBM Deep Learning*. URL: <https://www.ibm.com/topics/deep-learning>.
- [11] *Neuron*. URL: <https://en.wikipedia.org/wiki/Neuron>.
- [13] *Artificial euron*. URL: https://en.wikipedia.org/wiki/Artificial_neuron.
- [14] *Artificial Neural Network*. URL: https://en.wikipedia.org/wiki/Artificial_neural_network.
- [15] *Perceptron*. URL: <https://en.wikipedia.org/wiki/Perceptron>.
- [16] *ANNs activation functions*. URL: <https://www.v7labs.com/blog/neural-networks-activation-functions>.
- [17] *Activation Function*. URL: https://en.wikipedia.org/wiki/Activation_function.
- [18] *Feedforward Neural Network*. URL: https://en.wikipedia.org/wiki/Feedforward_neural_network.
- [19] *Feedforward Neural Network*. URL: <https://deepai.org/machine-learning-glossary-and-terms/feed-forward-neural-network>.
- [20] *Loss Functions*. URL: <https://towardsdatascience.com/loss-functions-and-their-use-in-neural-networks-a470e703f1e9>.

- [22] *Understanding the Backpropagation Algorithm*. URL: <https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd>.
- [23] *Backpropagation*. URL: <https://en.wikipedia.org/wiki/Backpropagation>.
- [24] *Backpropagation 3Blue1Brown*. URL: <https://www.youtube.com/watch?v=IHZwWFHwa-w>.
- [25] *Convolutions 3Blue1Brown*. URL: <https://www.youtube.com/watch?v=KuXjwB4LzSA>.
- [26] *ContinualAI, Understanding Catastrophic Forgetting*. URL: <https://www.youtube.com/watch?v=UnCAdbtvZhc>.
- [29] *CNNs a brief history*. URL: <https://medium.com/appyhigh-technology-blog/convolutional-neural-networks-a-brief-history-of-their-evolution-ee3405568597>.
- [31] *The stability-plasticity dilemma*. URL: <https://www.frontiersin.org/articles/10.3389/fpsyg.2013.00504/full>.
- [39] *Alexnet the architecture that changed CNNs*. URL: <https://towardsdatascience.com/alexnet-the-architecture-that-challenged-cnns-e406d5297951>.
- [41] *VGG very deep CNNs*. URL: <https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/>.
- [43] *A practical experiment for comparing lenet alexnet vgg and resnet models with their advantages*. URL: <https://tejasmohanayyar.medium.com/a-practical-experiment-for-comparing-lenet-alexnet-vgg-and-resnet-models-with-their-advantages-d932fb7c7d17>.
- [44] *Resnet residual neural network*. URL: <https://viso.ai/deep-learning/resnet-residual-neural-network/>.
- [47] *An overview on mobilenet an efficient mobile vision cnn*. URL: <https://medium.com/@godeep48/an-overview-on-mobilenet-an-efficient-mobile-vision-cnn-f301141db94d>.
- [49] *Imagenet*. URL: <https://www.image-net.org/about.php>.
- [50] *CUB200-2011*. URL: https://www.vision.caltech.edu/datasets/cub%5C_200%5C_2011/.
- [51] *GPU GEMM architecture*. URL: <https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html>.
- [54] *Fx representation*. URL: <https://www.geeksforgeeks.org/fixed-point-representation/>.
- [55] *Digital Design book, Morris, Mano*. URL: <https://docs.google.com/file/d/0B8-drkZsESDnN2NmYTQxYjQtYTMwZi00N2IzLTkxNjgtZjI1NTZiN2FjNDli/edit?pli=1&resourcekey=0-Yk8bAsCt9I5epBNFTG8KMq>.

- [58] *NumPy documentation*. URL: <https://numpy.org/devdocs/user/whatisnumpy.html>.
- [59] *What is BLAS and LAPACK in NumPy*. URL: <https://superfastpython.com/what-is-blas-and-lapack-in-numpy/>.
- [60] *NumPy multithreaded parallelism*. URL: <https://superfastpython.com/numpy-multithreaded-parallelism/>.
- [61] *CuPy Documentation*. URL: <https://docs.cupy.dev/en/stable/overview.html>.
- [62] *Vitis HLS Guide*. URL: <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Navigating-Content-by-Design-Process>.
- [63] *Vitis HLS optimization directives*. URL: <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Optimization-Directives?tocId=RzqCJW4MAxgf0BgirykUsw>.
- [64] *Vivado Design Suite User Guide*. URL: <https://docs.xilinx.com/r/en-US/ug910-vivado-getting-started/What-is-the-Vivado-Design-Suite>.
- [65] *Vitis Application Acceleration Development Flow*. URL: <https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration>.
- [66] *Vitis Embedded Software Development Flow*. URL: <https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded>.
- [67] *Methodology For Accelerating Applications*. URL: <https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Methodology-for-Accelerating-Data-Center-Applications-with-the-Vitis-Software-Platform>.
- [68] *ZCU102 Evaluation Board User Guide*. URL: <https://docs.xilinx.com/v/u/en-US/ug1182-zcu102-eval-bd>.
- [70] *Amdahl's Law*. URL: https://link.springer.com/referenceworkentry/10.1007/978-0-387-09766-4_77.