Exploring the Capabilities of FPGA DSP Blocks in Neural Network Accelerators

by

Lenos Ioannou

Thesis

Submitted to the University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

School of Engineering

November 2021

Contents

List of Tables

List of Figures

Acknowledgments

First and foremost I would like to thank my supervisor, Dr. Suhaib Fahmy, for giving the opportunity to study for my PhD under his guidance. His feedback has been invaluable during my studies at Warwick, while our meetings and discussions helped me grow as a researcher and a person. He always took the time to help me improve my technical writing and to answer even my most tedious questions. His enthusiasm and support even in the most hectic periods will guide me forever.

I would also like to thank all the people I have met at the University of Warwick, especially the PhD students (Ryan, Alex and Kusuma), post-docs and other students at the WARC lab. Their company and discussions have made the daily routine better in so many ways.

I am also very thankful to my friends back home, who have always made the getaways from my studies more fun and relaxing.

Finally, I am eternally grateful to my family, especially my parents, Ntinos and Stella, for their unconditional support and understanding, in particular throughout my studies. I am also very thankful to my grandmother Rita, with whom I spent a lot of time growing up and helped shape the person I am today.

Declarations

1 Publications

Parts of this thesis have been previously published by the author in the following:

[1] L. Ioannou and S. A. Fahmy. Network Intrusion Detection Using Neural Networks on FPGA SoCs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 232–238, 2019.

[2] L. Ioannou and S. A. Fahmy. Neural Network Overlay Using FPGA DSP Blocks. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 252–253, 2019.

[3] L. Ioannou and S. A. Fahmy. Lightweight Programmable DSP Block Overlay for Streaming Neural Network Acceleration. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT)*, pages 355–358, 2019.

[4] L. Ioannou, A. Al-Dujaili, and S. A. Fahmy. High Throughput Spatial Convolution Filters on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(6):1392–1402, 2020.

[5] L. Ioannou and S. A. Fahmy. Streaming Overlay Architecture for Lightweight LSTM Computation on FPGA SoCs. *Submitted to: ACM Trans. Reconfigurable Technol. Syst.*.

2 Sponsorships and Grants

This research was funded by the UK Engineering and Physical Sciences Research Council (EPSRC), grant EP/N509796/1, and the School of Engineering, University of Warwick, UK.

Abstract

Neural networks have contributed significantly in applications that had been difficult to implement with the traditional programming concepts (e.g. computer vision, natural language processing). In many occasions, they outperform their hand coded counterparts and are increasingly popular in end user applications. Neural networks, however, are compute and memory demanding, making their execution in resource constraint devices more difficult, especially for real time applications. Custom computing architectures on Field-Programmable Gate Arrays (FPGAs) have traditionally been used to accelerate such computations to meet specific requirements. Nonetheless, most approaches in the literature do not consider in detail the underlying FPGA architecture, resulting in less efficient implementations. They additionally have focused on complex designs optimised for high throughput in a datacenter setting with access to large datasets in memory. Meanwhile real edge applications are often processing streaming sensor data and require consideration of efficiency. Detailed FPGA implementations involve time consuming low level design effort, which in turn result in long turnaround time. FPGAs have evolved over the years to include hard macro blocks, for example Digital Signal Processing (DSP) blocks, that map more efficiently widely used operations. In addition, FPGAs are often tightly coupled with embedded microprocessors in a System-on-Chip (SoC) arrangement that offers a complete system solution. This thesis explores the capabilities of FPGA DSP blocks in neural network accelerators. Within this context, practices and tools that improve turnaround time have been explored, drawing conclusions on how to exploit DSP blocks in a way that maximises performance and efficiency. Finally, the work in this thesis shows that designing overlays in an architecture-centric manner can result in high operating frequency, while scaling to better utilise FPGA resources.

Acronyms

ALU Arithmetic Logic Unit.

API Application Programming Interface.

ASIC Application Specific Integrated Circuits.

AXI Advanced eXtensible Interface.

BRAM Block Random Access Memory.

CLB Configurable Logic Block.

CNN Convolutional Neural Networks.

CPU Central Processing Unit.

CSD Canonic Signed Digit.

CUDA Compute Unified Device Architecture.

DA Distributed Arithmetic.

DBN Deep Belief Network.

DDoS Distributed Denial of Service.

DMA Direct Memory Access.

DNN Deep Neural Networks.

DNNDK Deep Neural Network Development Kit.

DoS Denial of Service.

DPU Deep-learning Processor Unit.

DRAM Dynamic Random Access Memory.

DSP Digital Signal Processing.

DT Decision Trees.

FF Flip-Flop.

FIFO First In First Out.

FIR Finite Impulse Response.

FPGA Field-Programmable Gate Array.

FPS Frames Per Second.

Gbps Giga bits per second.

GFLOPs Giga Floating-Point Operations per second.

GOPs Giga Operations Per second.

GPU Graphics Processing Unit.

GRU Gated Recurrent Unit.

HD High Definition.

HDL Hardware Description Languages.

HLS High Level Synthesis.

IDS Intrusion Detection System.

IoT Internet of Things.

IPC Instructions Per Cycle.

IPv4 Internet Protocol version 4.

LSTM Long-Short-Term Memory.

LUT Look-Up-Table.

MAC Multiply-Accumulate.

MAE Mean Absolute Error.

MIO Multiplexed Input Output.

ML Machine Learning.

NDAE Non-symmetric Deep Auto-Encoder.

NN Neural Networks.

PCA Principal Component Analysis.

PL Programmable Logic.

PR Partial Reconfiguration.

R2L Remote to Local.

RAM Random Access Memory.

ReLU Rectified Linear Unit.

RF Random Forests.

RISC Reduced Instruction Set Computing.

RNN Recurrent Neural Networks.

ROM Read Only Memory.

RTL Register-Transfer Level.

SAD Sum of Absolute Differences.

SIMD Single Instruction Multiple Data.

SoC System-on-Chip.

SOM Self Organising Maps.

SRL Shift Register Look-up-table.

SVM Support Vector Machine.

TPU Tensor Processing Unit.

U2R User to Root.

VHDL VHSIC Hardware Description Language.

VHSIC Very High Speed Integrated Circuit.

VLIW Very Long Instruction Word.

Chapter 1

Introduction

The emergence of Machine Learning (ML) has enabled a plethora of applications that would be more difficult to implement with the traditional programming methods. Although ML concepts have been reported in the literature for many years, the availability of large volumes of data, through big data, coupled with faster training turnaround time due to the availability of highly parallel compute platforms and other algorithmic and mathematical optimisations, have recently enabled ML models to outperform manually programmed solutions in more problem domains. Hence, ML has attracted the research interest of many disciplines, among them computer engineering, in which this thesis lies.

Machine Learning approaches span a wide range of learning techniques, supervised or unsupervised, and a variety of models, Self Organizing Maps (SOM), Support Vector Machines (SVM), Neural Networks (NNs), Decision Trees (DT), and many more [6]. The versatility and more complex structure of NNs has rendered them more capable to model non-linear tasks more accurately, compared to the other options. Therefore, NNs have been widely used in a wide spectrum of applications, from healthcare [7] to computer vision [8].

The operation of ML models comprises two phases, *training* and *inference*. During training, the ML model is formed and refined using a dataset to determine the best parameters to achieve a required task, iteratively refining to reduce the prediction error. During inference, the trained model is used to make predictions, based on the learned parameters, on new, unseen data. Training comprises the heaviest workload of the two and usually takes place offline on highly parallel computing platforms. Indicatively, it can take from a few hours to a few days, depending on model's topology. Inference can potentially take place on any compute device, from highly powerful servers to extremely constrained edge devices. Nonetheless, minor adjustments, i.e. fine-tuning, of the trained model's parameters may be made to better suit the specific use patterns of the device or user, either on device or centrally.

Due to the more central nature of training, inference has been the main target of various optimisations in an effort to meet the needs and capabilities of different devices across the computing spectrum. Edge devices have been a key focus since they impose stricter and more challenging attributes, e.g. latency, throughput, energy, etc. Moreover, due to the variable and often prohibitive transfer latency to the cloud, real time edge devices must perform their computations locally [9]. One approach to meet strict constraints has been with the use of custom computing architectures, either Application Specific Integrated Circuits (ASICs) [10, 11] or implemented on FPGAs [12–14]. Custom computing architectures can be tailored to meet a device's specifications, at the cost of flexibility compared to more general computing platforms, e.g. CPUs and GPUs. FPGAs have continued to improve in terms of performance and efficiency, primarily due to architectural evolutions that incorporate a variety of hard macro blocks. Moreover, complete System-on-Chip (SoC) solutions, featuring a low power processor tightly coupled with an FPGA fabric, are ideal for edge solutions in which a generic processor is supported by an accelerator on an FPGA.

Although, FPGAs are not superior to ASICs in terms of raw characteristics, their off-the-shelf availability, which in turn results in smaller turnaround time, reduced cost as well as their reconfigurable operation render them an excellent solution for acceleration, especially in domains where the computational algorithms continue to evolve, such as ML. We therefore have the opportunity to explore new approaches that better take advantage of the capabilities of modern FPGAs to maximise their effectiveness.

1.1 Motivation

The increasing ubiquity of interconnected devices at the edge has provided the means to automate various daily tasks through the use of sensors and actuators integrated with these devices. We therefore, have the ability to collect unprecedented volumes of data and the means to automate actions through the use of NNs. However, as the increasing interconnectivity of smart devices provides a plethora of even more advanced capabilities, it simultaneously renders efficiency more important. Edge devices are often battery powered, to enable portability, and of reduced processing power, supporting a subset of the instruction set of desktop computers. Custom computing architectures on FPGAs can therefore be used to provide real time performance and high energy efficiency at the edge.

NNs are extremely demanding in terms of memory requirements, to store the trained model's coefficients and intermediate results of computation, in addition to the significant computational workload they entail. Even a relevantly small NN may have such memory requirements that on-chip storage does not suffice, requiring frequent off-chip memory transfers that are costly in terms of energy and latency. The significant computational workload mainly consists of matrix-vector and matrix-matrix multiplications. In addition, edge devices are often dealing with data that streams from the various sensors or other components, e.g. camera, microphone etc.. Therefore ML computations should ideally be optimised around this streaming dataflow, which can add complexity to the design of custom architectures.

The design of custom computing architectures on FPGAs using Hardware Description Languages (HDLs), e.g. Verilog or VHDL, at Register-Transfer Level (RTL) involves time consuming low level design effort. However, machine learning workloads differ in their parameters and structure, therefore accelerator architectures should be scalable and easily reconfigured to different model parameters and configurations. To this end, High Level Synthesis (HLS) has emerged as an alternative to traditional RTL design, essentially raising the programming domain to a higher level language, e.g. C, with guided automated translation into an architecture. Another option is to build an abstracted overlay architecture that is fundamentally flexible enough to adapt to varying NN model topologies.

Most compute architectures on FPGAs are designed as static solutions, not fully taking advantage of the reconfigurable nature of the FPGAs. In addition, there is often very little consideration of the underlying FPGA architecture, resulting in implementations that operate at well below the frequencies that are theoretically achievable, hence not fully exploiting the capabilities of the device. Maximising device capabilities is key to meeting strict specifications in a challenging domain.

1.2 Aims and Objectives

This work in this thesis aims to explore the more efficient use of the underlying FPGA architecture and macro blocks for lightweight NNs on devices at the edge, while maintaining flexibility within this domain. This in turn is expected to result in a computing architecture that achieves higher operating frequency and hence performance, while offering higher energy efficiency through better resource utilisation. Lastly, since this thesis targets edge devices, all implementations are aimed to be integrated in an SoC environment, considering a streaming dataflow model.

Therefore, the aforementioned aims result in the following objectives:

- **Use of Digital Signal Processing (DSP) blocks:** NNs involve a significant amount of Multiply-Accumulate (MAC) operations that can be more efficiently mapped to the DSP macro blocks on modern FPGAs. This is expected to yield implementations that offer:

High operating frequency: which is expected to contribute to achieving higher performance in terms of latency and throughput.

Higher energy efficiency: since functions implemented on the FPGA's macro blocks consume less power than their equivalent implementations in fabric [15]. Moreover, higher operating frequency can contribute partly to energy efficiency, since leakage currents are clock independent [16].

- **Programmability-Abstractions:** Although FPGAs are reconfigurable, the lengthy design and compilation times result in less flexible deployments. Enabling more rapid deployment, through the use of higher level languages or coarse grained overlays, would improve the flexibility of FPGA design.

- **Streaming dataflow:** FPGA SoCs, and edge devices in general, often collect and distribute data in a streaming fashion, not using highly parallel bulk transfers that are used with more datacenter oriented interconnect such as PCI Express. Such dataflow architectures are more difficult to efficiently scale in terms of performance since data availability is less abundant. Therefore, an architecture that tailors the operation of its compute units and dataflow to a streaming arrangement would be better suited for the edge domain.

- **Parallelism:** NNs consist of highly parallel workload from which improved performance can be obtained by unrolling these computations. The unrolling scheme, however, must be tailored to the streaming dataflow.

1.3 Research Contributions

The research contributions of this thesis comprise computing architectures, implemented on FPGAs, that accelerate NNs, or part of their computations. The implemented architectures have been evaluated mainly in terms of performance and resource utilisation. Occasionally, additional features have been derived in order to normalise the varying capabilities between different FPGA devices and make more objective comparisons with previous work.

More specifically, the research contributions of the work in this thesis are as follows:

- An exploration an Intrusion Detection System (IDS) application using NNs on an FPGA SoC, implemented with HLS. The IDS demonstrated improved detection times while offering coefficient re-programmability along with the use of floating point operations, setting a baseline for the complexity of a parallel but not highly optimised architecture.

- An exploration of various large scale 2-D spatial convolution filters on modern image resolutions that is heavily optimised around modern FPGA DSP blocks. A detailed investigation of scalability for different filter sizes and image resolutions, along with their impact on operating frequency and resource utilisation, is performed. Comparisons made with previous work and equivalent implementations with HLS motivate the work that follows thereafter in this thesis.

- An FPGA overlay architecture, tailored to the compute patterns of specific NN layers and built around the concept of DSP block as a neuron. The overlay processes a streaming flow of data, is implemented in an FPGA SoC environment, and is runtime programmable. The overlay achieves high operating frequency and demonstrates improved performance compared to mobile, desktop CPU, and relevant previous work on FPGAs.

- An enhanced version of the aforementioned overlay architecture to support a wider variety of layer types including Long Short-Term Memory (LSTM) that require complex feedback structures. This overlay is shown to scale with a small frequency overhead, while extensive comparisons with previous work demonstrate the benefits of the proposed approach.

1.4 Thesis Organisation

Chapter 2 presents relevant background information on Machine Learning and Neural Networks in particular, followed by the different compute platforms and the various algorithmic optimisations, in addition to a literature review. This includes details on basic NN building blocks, including different layer types and activation functions. More information follows on various compute platforms, focusing mainly on modern FPGAs and the different methods and tools to enable more rapid deployment.

Chapter 3 demonstrates an NN application for network intrusion detection on an FPGA SoC device. Initially, the NN is trained in software, followed by architecture generation using HLS. The architecture is flexible, to allow coefficient modification at runtime, while using 32-bit floating point arithmetic throughout. The HLS generated design is integrated in an SoC implementation, alongside an ARM-A9 embedded processor, and has been functionally tested. The chapter concludes with comparisons with equivalent software implementations and previous work, showing improved detection time.

Chapter 4 explores FPGA implementations of spatial convolution filters, using Verilog HDL, focusing on modern image resolutions and respectively scaled filter sizes. The different design choices for the selected architecture are substantiated by comparisons on indicative designs, followed by an exploration on how the selected architecture scales and, finally, comparisons with HLS equivalent implementations. The work in this chapter motivates the transition from HLS to a more architecture-centric design approach that is better suited to the capabilities of the FPGA architecture, specifically DSP blocks and their runtime flexibility. Moreover, the extensive range of filter sizes demonstrates that the proposed architecture scales well in terms of frequency, laying the groundwork for the chapters that follow.

Chapter 5 presents an overlay implementation of feed forward NNs, using streaming dataflow with the FPGA DSP blocks acting as individual neurons. The implemented architecture achieves high operating frequency and better performance compared to software equivalents and previous work on HLS. The overlay has been integrated in an SoC implementation, operating at a baseline frequency, and has been functionally tested. Although the baseline NNs used in this chapter do not stress the scalability and the functionality of the overlay significantly, they serve as a stepping stone for the more complex overlay that follows.

Chapter 6 presents an expanded overlay streaming architecture that supports an additional set of NN layers, most notably LSTMs that require feedback, as well as more flexible activation functions. The overlay in this chapter is shown to scale at a low frequency overhead, while the extensive comparisons between the implemented overlays in this chapter and previous work demonstrate and quantify the effectiveness of the proposed approach.

Chapter 7 concludes the work presented in this thesis and discusses future work based on its findings.

1.5 Publications

The originality of the research contributions of this thesis is demonstrated through publications in the following peer-reviewed conference and journal proceedings:

1. L. Ioannou and S. A. Fahmy. Network Intrusion Detection Using Neural Networks on FPGA SoCs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 232–238, 2019 [1].

2. L. Ioannou and S. A. Fahmy. Neural Network Overlay Using FPGA DSP Blocks. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 252–253, 2019 [2].

3. L. Ioannou and S. A. Fahmy. Lightweight Programmable DSP Block Overlay for Streaming Neural Network Acceleration. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT)*, pages 355–358, 2019 [3].

4. L. Ioannou, A. Al-Dujaili, and S. A. Fahmy. High Throughput Spatial Convolution Filters on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(6):1392–1402, 2020 [4].

5. L. Ioannou and S. A. Fahmy. Streaming Overlay Architecture for Lightweight LSTM Computation on FPGA SoCs. *Submitted to: ACM Trans. Reconfigurable Technol. Syst.* [5].

Chapter 2

Background and Literature Review

Edge devices' functionalities have evolved over the years, from passive to more interactive, incorporating sensors and actuators to interact with the physical world in a smarter way. Meanwhile, the increasing interconnectivity of such devices has enabled the collection of unprecedented volumes of data, from which specific patterns can be extracted and used for better informed future predictions and user tailored operation. Machine Learning has provided the means to automate this learning process and in many cases has outperformed hand coded methods of extracting such patterns. Neural Networks are considered by many the most prominent class of ML models, and are nowadays increasingly used on wide spectrum of computing devices, from highly parallel platforms to resource constrained edge devices. As a result, there is great interest on the workload and size of Neural Networks and how these can be executed more efficiently on various platforms through various optimisations, either algorithmic or architectural. This chapter covers all aforementioned aspects in theoretical background and relevant literature review.

2.1 Machine Learning Motivation

ML algorithms were shown to generalise their learned patterns to new, previously unseen data. The latter, renders them very useful to simple day to day tasks, but more importantly, to more complex tasks that have greater impact, for example network security applications. Specifically, ML algorithms have the potential to detect new, zero-day, attacks or even modified known attacks that have been altered adequately to deceive security mechanisms. Both of which are very difficult to detect with hand coded rules. Therefore, there is plenty previous work in the literature that explored the use ML in security applications, in the context of an Intrusion Detection System (IDS). The work in [17] explored the use of Non-symmetric Deep Auto-Encoders (NDAE) and Random Forests (RF) for network intrusion detection, demonstrating great potential in their detection results and improved turnaround time compared to a Deep Belief Network (DBN). The use of Neural Networks in the same domain was explored in [18–23], obtaining high detection rates.

2.2 Real Time Signal and Image Processing Systems

Signal and image processing are very popular application domains of compute systems. The real time constraints that these domains usually pose, result in stringent system specifications. These specifications are often met with application specific architectures the accelerate parts or even complete digital signal processing algorithms. As a result, signal and image processing have sustained interest in real time response implementations by accelerating their computations. Finite Input Response (FIR) filter acceleration, for example, has been presented in [24] and [25]. Image processing acceleration, on the other hand, is becoming increasingly more complex due to the continuously increasing image resolutions used in vision systems. This results in even more stringent requirements, in terms of buffer memory and workload, that have to be met in order to achieve real time operation.

Image, and signal processing algorithms usually include multiplications with tunable parameters and a signal input (i.e. kernel window and image). These parameters have been traditionally defined by experts and involve signifiant human intervention. The emergence of ML has provided the means to define these parameters empirically, through the availability of datasets and without the need of expert knowledge. In addition, the automated training frameworks in the ML domain reduce human interaction to the minimum while yielding models that are competitive and often surpass algorithms defined by experts.

2.2.1 Convolution

Convolution, or 2-D spatial filtering, is fundamental operation in image processing. It comprises a kernel matrix, or filter window, that scans an input image at a given stride. During the scan, an output image pixel is generated by calculating the weighted sum of the overlapping kernel area, as depicted in Figure 2.1. The kernel dimensions and coefficients define the transformation of the image. For example the transformation can be edge detection, sharpening or even a specific feature in the image. Convolution has always been popular among the research community as it constitutes the foundation for many vision systems. Accelerating convolutions is therefore paramount for real time vision

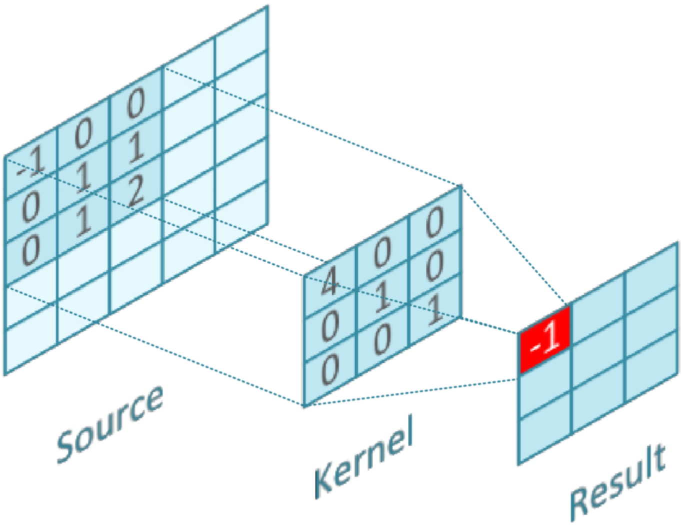


Figure 2.1: Convolution operation showing the source image, overlapping kernel window and result image [26].

systems, as shown in [27–32]. Other previous work have approached this task alternatively, mainly divided into multiplierless, that make use various theorems, optimisations and transformations to avoid the use of multipliers [33–37], and others that make use of distinct multiplications [24, 25, 38].

2.3 Neural Networks

Neural Networks are computational models inspired by human cognition, able to form complex non-linear functions from a given dataset. Their learning, similarly to any other ML model, can be further categorised in supervised and unsupervised. In supervised learning, a dataset with a set of inputs and their corresponding outputs (i.e. labelled data) is provided. The Neural Network is then trained to match the expected output given that specific set of inputs, or classify them to a given class. During unsupervised training, a dataset with only inputs is provided, with the task to extract any patterns from the input data. For example, an unsupervised model could automatically cluster its input data to categories with similar features. Supervised learning therefore may include expert intervention to derive the output labels, rendering the preparation of the used dataset a more time-consuming process compare to unsupervised learning. On the other hand, unsupervised training requires more data and more time to achieve satisfactory accurate predictions, which in turn require more compute and memory resources during training. Lastly, unsupervised learning can also be susceptible to dataset artefacts or erroneous spikes that may have been included due to the automated operation of the process. As a result, most trained models used for benchmarking in the literature are trained with supervised learning.

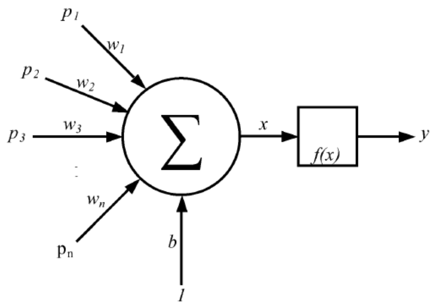


Figure 2.2: Neuron Structure, showing the inputs, weights, biases and activation function [12].

Neural Networks consist of compute entities called neurons, as shown in Figure 2.2. Each neuron calculates the weighted sum of its inputs, adds the result to an offset value (bias), followed by an activation function for non-linearity. Most commonly used activations functions are the Rectified Linear Unit (ReLU), sigmoid and tanh, as shown in equations 2.1 - 2.3.

$$relu(x) = max(0, x) \tag{2.1}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.3}$$

A layer is then formed with the use of multiple neurons, operating in parallel, and by extension, a Neural Network is built with a sequence of layers that propagate their results between them. The first layer of a Neural Network is the input layer, from which new data are fed to the network, whereas the last layer is the output layer, out of which the results of the network are generated. Any layers between the input and output layer are called hidden layers. Conventionally, Neural Networks with one or two hidden layers, in addition to the input and output layers, are called shallow. Networks with more than two hidden layers are considered deep networks. Various types of Neural Network layers exist that are tailored to different types on inputs. Depending on the layer type, the functionality of each neuron is somewhat different, however the core operation remains inherent. Neural Networks have been successfully applied in a broad range of fields, from automotive [39], to healthcare [7].

2.3.1 Fully Connected, or dense Layers

Fully connected, or dense, layers are the most commonly used type of layers as they can be used as an independent solution or be embedded in more complex Neural Network topologies. An indicative structure of a fully connected layer is shown in Figure 2.3. Inputs to fully connected layers are represented numerically and so symbolic (or categorical) inputs must be converted to suitable formats before being applied. Applications of NNs consisting solely of fully connected layers have been indicatively used in network security [12, 19, 20, 40, 41], healthcare [7], automotive [39], language processing [42] and gas classification [14].

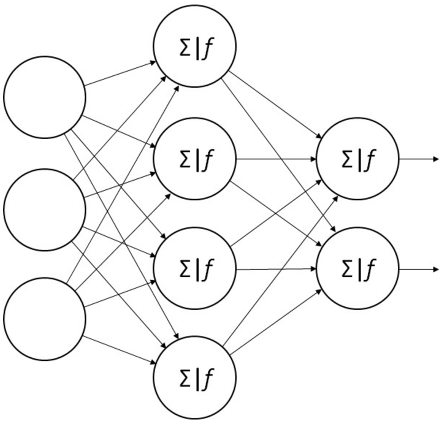


Figure 2.3: Artificial neural network structure.

2.3.2 Convolutional layers-CNNs

Convolution gained even more attention in the recent years due to the success of Convolutional Neural Networks (CNNs) [43]. CNNs mainly consist of convolutional layers that use convolutions abundantly to extract features from an input image. Convolutional layer weights (i.e. kernel coefficients) are defined automatically during training. The correlation between convolution and CNNs is demonstrated in [37], in which the convolution operation in CNNs is optimised by using power of two weights.

Convolutional layers incorporate the traditional 2-D convolution operation in image processing. Convolutions are more efficient at processing images compared to fully connected layers. For example, although an image could be flatted and used in a fully connected layer, it would require many more neurons and connections between them, resulting in higher memory and compute requirements. Convolutional layers act as feature extraction entities on the feature maps (i.e. images) that are propagated through the network. They are mostly used in tandem with other types of layers that perform the actual classification task on the extracted features. An example of a complete CNN is shown in Figure 2.4, which is trained for handwritten digit recognition [44]. The input images initially propagate through the feature extraction layers, that mainly consist of convolutional layers, followed by the classification part that consists of fully connected layers. Various CNN topologies have been proposed over the years, for example AlexNet[8], VGG [45] and GoogLeNet [46]. Each of these CNNs has pushed the state of the art while demonstrating improvements or tradeoffs between them. For example, a CNN topology may offer faster training turnaround, going deeper or reduce the overall workload, among others.

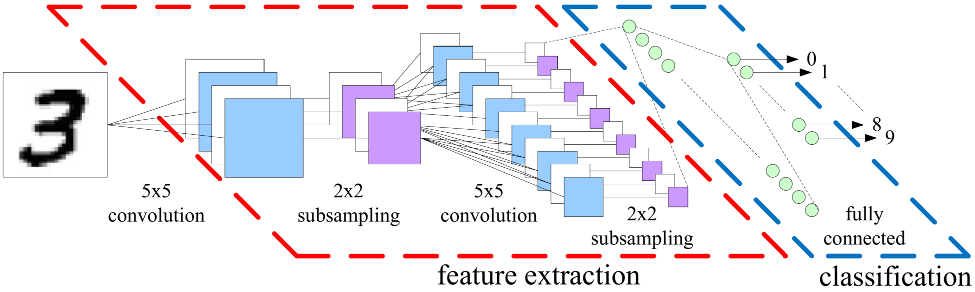


Figure 2.4: A typical CNN structure, showing the distinct feature extraction and classification parts [44].

The different layers in a CNN have significantly different workload and memory requirements. Indicatively, the analysis that takes place in [13] extracts the number of operations, Figure 2.5, and the number of weights, Figure 2.6, for each layer. In this case, the convolutional layers comprise the largest workload while benefiting from weight reuse, whereas fully connected layers have reduced workload but higher memory requirements.

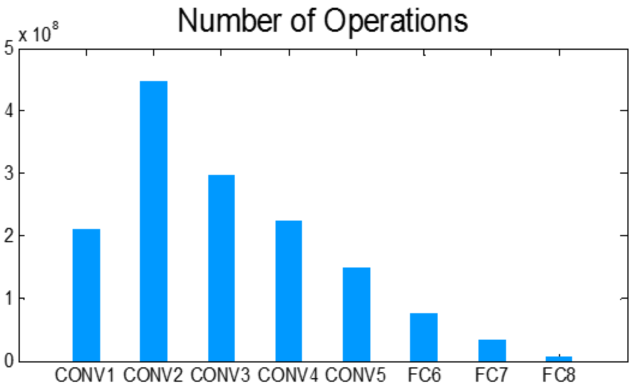


Figure 2.5: Per layer number of operations in AlexNet [13].

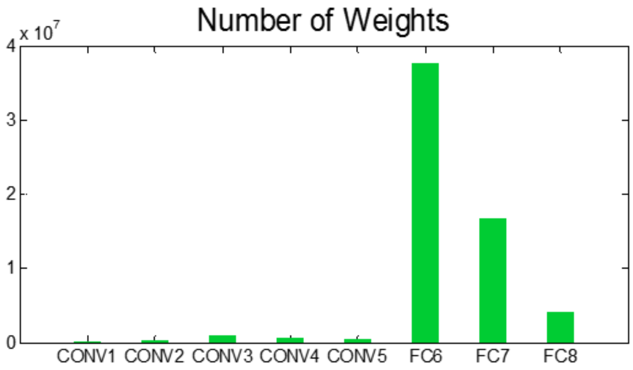


Figure 2.6: Per layer number of weights in AlexNet [13].

As a result of their popularity, toolflows have been proposed to map these CNN topologies on custom computing architectures [47]. In addition, although CNNs are primarily used for image recognition tasks, the work in [23] uses CNN topologies for a network security application. The authors have converted the network traffic features into images which have been used to train CNN networks. The proposed method however did not obtain better detection rate compared to simpler NNs comprising fully connected layers. Nonetheless, it demonstrates the versatility of neural networks in a broad range of domains.

2.3.3 Recurrent Layers-RNNs

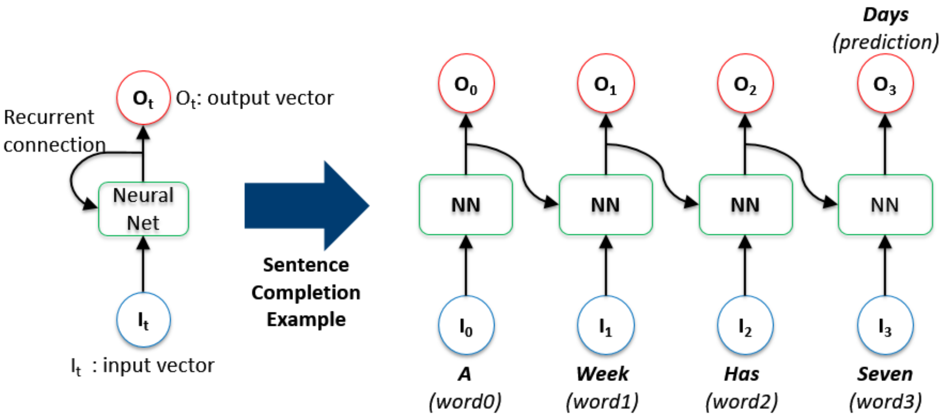


Figure 2.7: A recurrent unit, its unrolled computation over timesteps [48].

Recurrent layers are tailored for sequential or time series data applications. To enable this class of networks to extract any correlation between data sequences, they include feedback connections to previous outputs, which in turn translate to computing dependencies during their runtime. The feedback connections embed a memory element to the network, based on the data that has been previously propagated through the layer. An example of a recurrent unit unrolled over time is shown in Figure 2.7. The unit receives the first four words of a sentence serially and generates the fifth word, based on the previous ones. The initial structure of recurrent units however has been proven prone to vanishing and exploding gradients. The former case is when a gradient is very small, during training, and it continues to become smaller until it vanishes. The latter case is the exact opposite, referring to a gradient that is very big, creating an unstable model. As a result, variants have been proposed to overcome the aforementioned issues.

Long Short-Term Memory (LSTM) layers

Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs) are RNNs that have been proposed to overcome the vanishing and exploding gradient problems. Although both perform similarly in many tasks, the more complex structure of LSTMs theoretically allows them to learn more complex sequential patterns.

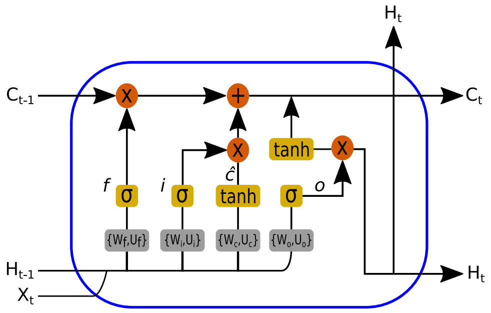


Figure 2.8: An LSTM unit.

Equations 2.4 to 2.9 describe the operation of an LSTM unit, which is also illustrated in Figure 2.8, where $\odot$ denotes element-wise multiplication, and W and U are the weights of current input features and the previous cell outputs respectively. More specifically, an LSTM unit consists of the forget gate (f_t), input gate (i_t), and output gate (o_t), along with the cell state(C_t), its partial result($\tilde{C_t}$), and the LSTM cell output (H_t). The forget gate controls the amount of information to discard from the previous cell state, the input and partial cell state define the new information to add to the cell state and the output gate defines the LSTM cell's output based on the current cell state. The main difference between LSTMs, and RNNs in general, and feedforward NNs, is the feedback connections from previous outputs (C_{t-1} and H_{t-1}).

An LSTM network can process a sequence of inputs, each of which can be a scalar or a vector. The sigmoid (σ) and tanh activation functions are most commonly used in this configuration. However, some flexibility in the application of activation functions is required to support different networks.

$$f_t = \sigma(W_f x_t + U_f H_{t-1} + b_f) \tag{2.4}$$

$$i_t = \sigma(W_i x_t + U_i H_{t-1} + b_i) \tag{2.5}$$

$$\tilde{C_t} = tanh(W_c x_t + U_c H_{t-1} + b_c) \tag{2.6}$$

$$o_t = \sigma(W_o x_t + U_o H_{t-1} + b_o) \tag{2.7}$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C_t} \tag{2.8}$$

$$H_t = tanh(c_t) \odot o_t \tag{2.9}$$

LSTMs have been successfully applied to weather forecast [49], network security [21, 22], optical character recognition [50, 51], speech recognition [52], character level text prediction [53–55], among others.

2.3.4 Hyperparameters and Evaluation

Hyperparameters are a set of parameters used during the training of a Neural Network that have an impact on how effectively the model learns from the input data. These parameters are usually defined experimentally as their impact differentiates according to the task at hand.

- **Neurons per layer and number of layers:** the number of neurons in a layer and the number of layers affect the learning capacity of the network. Therefore, depending on the complexity of the task at hand, different network topologies are explored to find the better suited one.

- **Batch size:** is the number of input samples after which the model coefficients (i.e. weights and biases) are updated during training.

- **Epochs:** an epoch is called after all samples in the training set have been propagated through the network once. Usually this process is repeated multiple times for the network to learn.

- **Learning Rate:** is the rate at which the coefficients are updated after the pass of a batch.

- **Loss function:** is the function with which the prediction error of a Neural Network is evaluated, which in turn affects the coefficient update during training.

- **Optimiser:** is the function, or algorithm, used to update the network's coefficients and reduce the prediction error of the model to ideally reach to the global minimum.

A Neural Network is frequently evaluated during training, e.g. after the pass of an epoch, showing the prediction error and, where possible, the percentage of correct classifications made. Although other metrics exist as well, accuracy is the most widely used one since it encapsulates the overall prediction capability of the model, while others focus on a specific attribute. For example, given a network that classifies its input data to two classes, e.g. class 1 and class 2, classification accuracy is calculated as follows:

$$accuracy = 100 * \frac{TP + TN}{TP + TN + FP + FN} \tag{2.10}$$

where:

- **TP:** True Positive, corresponds to a class 1 dataset entry has been correctly classified as such.
- **TN:** True Negative, corresponds to a class 2 dataset entry that has been correctly classified as such.
- **FP:** False Positive, corresponds to a class 1 dataset entry that has been incorrectly classified as class 2.
- **FN:** False Negative, corresponds to a class 2 dataset entry that has been incorrectly classified as class 1.

2.4 Compute Platforms

Compute platforms are the means by which Neural Networks are processed. These may be generic software programmable architectures, e.g. a Central Processing Unit (CPU) or a custom computing architecture implemented on an ASIC or FPGA.

2.4.1 Software Programmable Platforms

Software programmable platforms, CPUs and GPUs, have been the most popular for Neural Network computations. The ease of compiling an algorithm in software, coupled with the highly parallel operation of GPUs have rendered these platforms very user friendly. In addition, the availability Neural Network frameworks (e.g. Tensorflow [56], Keras [57], Caffe [58], Theano [59]), which have abstracted the low level details of NN implementations in software, acted as a catalyst for the wider use of these platforms.

Central Processing Units (CPUs)

The ubiquity of generic CPU compute architectures coupled with their low cost have rendered this class of units very appealing for NN execution. CPU capabilities may vary significantly in respect to the compute platform they are used in. For example, server CPUs are more capable compared to personal computer CPUs, both of which are more powerful compared to an embedded microprocessor that implements a Reduced Instruction Set Computing (RISC) architecture. Nonetheless, since embedded processors comprise the majority of edge devices' CPUs, there have been particular efforts in enabling a more efficient deployment to these devices through the use of compute optimisations. For example, Tensorflow [56] provides a *Lite* version that offers a range of optimisations for embedded processors, like the ARM Cortex-A72 on a Raspberry Pi 4.

Although the spectrum of CPU capabilities varies significantly depending on the targetted device, CPUs in general are not able exploit parallelism in NNs at significant extend due to their relevantly low core count and inefficiency in memory intensive computations. The work in [60] has shown that fully connected layer processing can become less efficient on CPUs due to their memory intensive patterns. This was demonstrated in an analysis conducted on AlexNet [8], a specific CNN topology. Two Intel Xeon E5-2650 CPUs running at 2.4GHz were used to measure AlexNet's Instructions Per Cycle rate (IPC). Fully connected layers achieved the lowest, compared to the other layer types, obtaining an IPC rate of less than 1, due to cache misses in all cache levels, causing a high number of stall cycles to fetch data from memory. Hence, we see that data flow optimisations are also important to fully reach the processing potential of the computing unit.

Therefore, despite the wide availability and ease of Neural Network deployment, CPUs are less likely to offer real time execution while also being less energy efficient for such tasks. As a result, latest CPU models include special NN processing units [61, 62], to offer more efficient processing for this class of algorithms.

Graphic Processing Units (GPUs)

GPUs attracted significant interest in the NN domain on both, training and inference ends. Their highly parallel compute architecture coupled with the availability of an abstracted Application Programming Interface (API), for example Nvidia's Compute Unified Device Architecture (CUDA), have greatly reduced execution times on both training and inference while maintaining the ease of software programmability. GPUs form factor has diversified over the years, providing solutions that range from powerful GPUs for central computing to GPUs in the embedded domain. Although Neural Networks' training is dominated, at the moment, by the use of powerful GPUs in workstations, the use of GPUs for inference, especially in the embedded domain, has not been very beneficial in terms of performance and efficiency, compared to custom computing architectures on ASICs and FPGAs. The latter is mainly due to the fact that custom computing architectures are more tailored to a specific application domain compared to the more generic architecture of GPUs. For example, GPUs offer limited parallelism in LSTMs due to their sequential components and dependencies to previous outputs. Therefore being underutilised when processing in streaming mode, requiring batch processing to achieve high throughput. Previous work in [52] showed that an LSTM implementation for speech recognition, that operates at 100MHz on a Xilinx Zynq XC7Z045 FPGA, is more energy efficient compared to a high-end NVIDIA GeForce Titan X GPU. The authors in [63] explored the partitioning and execution of large LSTM layers on FPGAs. Their proposed approach on a Xilinx Virtex 7 and a Zynq FPGAs demonstrated improved performance and energy efficiency compared to an Nvidia TITAN X Pascal GPU, in addition to a Intel Xeon E5-2665 CPU and previous work on FPGAs. The work in [55] proposed LSTM co-processors on Xilinx Zynq ZC7020, obtaining improved runtime and energy efficiency compare to an Nvidia Tegra TK1 GPU.

2.4.2 Application Specific Integrated Circuits (ASICs)

Application Specific Integrated Circuits (ASICs) have demonstrated superior performance and energy efficiency at the cost of flexibility. As reported in Chapter 1, FPGAs tradeoff between performance, flexibility and off-the-shelf availability have been the main advantages over ASICs. In addition, as Machine Learning is an actively explored domain, constantly pushing the state of the art, means that long fabrication times are likely to render ASICs outdated by the time they are produced. As a result, ASICs implementations in this domain are shown to maintain some flexibility with more generic architectures, for example, the systolic array architecture found in the Google TPU devices. As shown in Figure 2.9, the input data in a TPU systolic array flow from the left hand side, while the weights are loaded from the top. Compute units in between calculate partial sums which are accumulated at the lower end of the array.

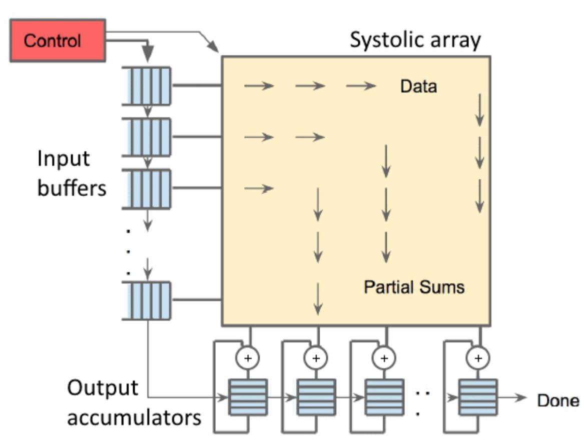


Figure 2.9: Systolic Array dataflow used in Google Edge TPU [64].

A systolic array implementation would therefore require a number of external memory channels to provide the required input and weights bandwidth. Its compute unit arrangement makes it efficient for matrix-matrix multiplications, which is ideal for CNNs that are inherently tailored for batch inference, but not very efficient in operations of other network types, e.g. matrix-vector or vector-vector. Therefore, in streaming processing, systolic arrays would be less efficiently utilised due to the lack of batching and shared weights. Moreover, the fixed architecture of an ASIC systolic array means that their efficiency is heavily based on the Neural Network's dimensions, with small networks to underutilise its compute resources. Additionally, the dependencies in LSTMs make it very difficult to utilise the pipeline parallelism in this class of architectures due to their more regular dataflow.

2.4.3 Field-Programmable Gate Arrays (FPGAs)

FPGAs are reconfigurable integrated circuits that can be found on various form factors that range from the embedded domain to high performance central computing. Their inner structure, depicted in Figure 2.10, comprises of a finite number of flexible routing resources and Configurable Logic Blocks (CLBs), or slices. Each CLB consists of small number of Look-Up-Table (LUT) memories, that can be used to implement any logic function, and synchronous memory elements, Flip-Flops (FFs). Each CLB is interfaced with a routing channel, which can be configured to I/Os of the CLB to the programmable interconnect. Routing channels are in turn interfaced with switchboxes which are able to make a connection between the available routing channels. FPGAs have evolved over the years to not only provide more configurable elements, but to also include hard macro blocks and interconnect for widely used operations, implemented directly in silicon. These macro blocks have enabled more efficient implementations, in terms of both performance and energy, while their dynamic

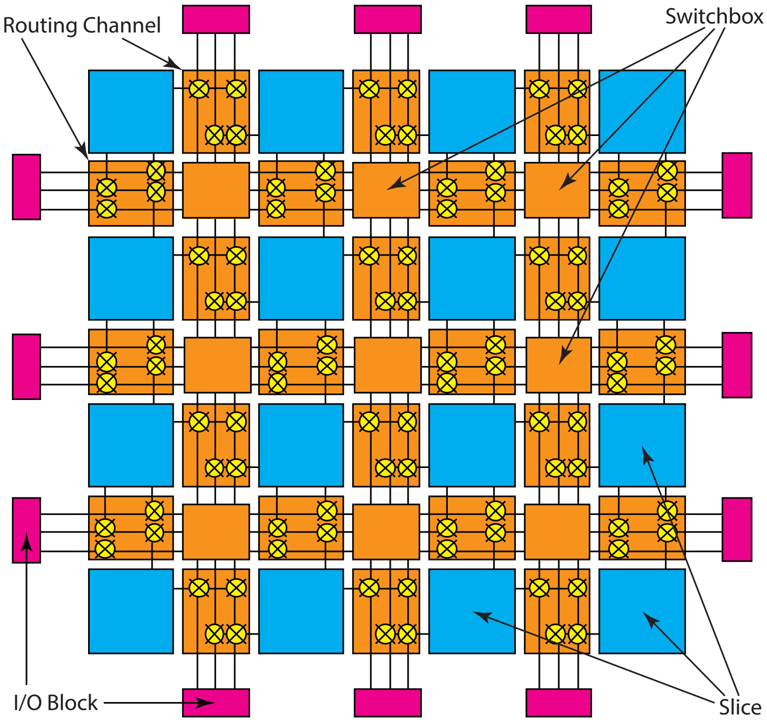


Figure 2.10: A part of an FPGA architecture, showing the various building blocks. [65].

programmability provides a degree of flexibility. Moreover, the reconfigurable fabric has also been integrated alongside an embedded processor, offering a complete system solution on an FPGA SoC. Depending on the family, the capabilities of FPGAs may vary, providing different interconnect capabilities, different number of macro blocks or more advanced LUT functionality, among others.

FPGAs have traditionally been used to accommodate custom computing architectures to accelerate workloads and achieve real time performance. Critical applications that require real time performance, among others, are network security algorithms. The faster the response in this class of algorithms means that less malicious packets enter the network, which results in a more effective defence mechanism. Accelerating traditional intrusion detection workloads on FPGAs has been explored in [66] and [67], while more modern ML approaches on FPGAs have been explored in [68–70]. In a similar way, FPGAs have also been used to accelerate vision computations for real time performance [28–32].

FPGA Macro blocks

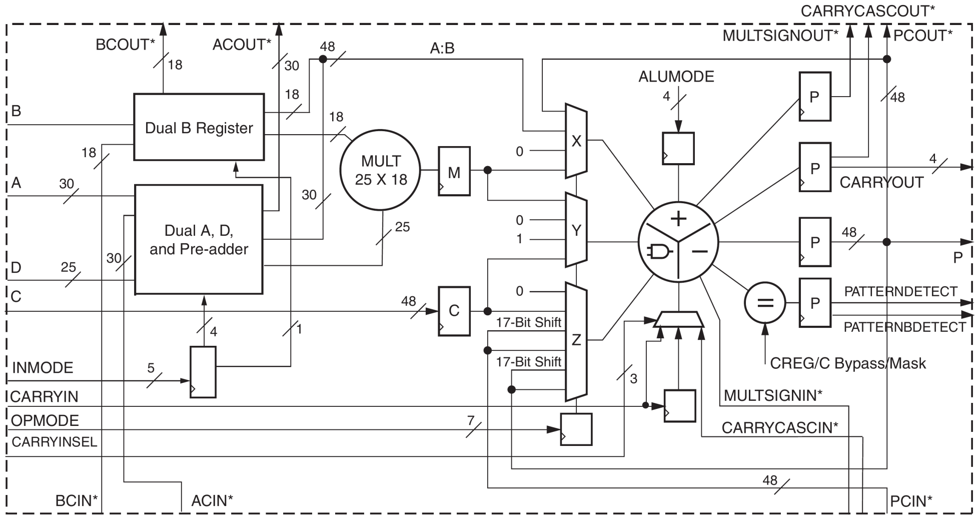


Figure 2.11: DSP48E1 compute block architecture, showing the various datapaths, compute units and configurations [71].

Hard macro blocks are distributed on the FPGA IC and can offer, for example, improved distributed memory storage, in the form Block RAM or most recently Ultra RAM, or advanced compute capabilities, with Digital Signal Processing (DSP) blocks. An average BRAM memory block has about 32 Kbit of memory which can be configured as 32K × 1 bit, 16K × 2 bit, etc. DSP blocks are able to perform more complex computations, that are widely used in signal processing applications. An indicative DSP block architecture is shown in Figure 2.11. It consists of three main compute blocks, a pre-adder, a multiplier and an Arithmetic Logic Unit (ALU). The operation of DSP blocks can be dynamically configured at runtime, for example, to execute various ALU operations or select the different input registers to the ALU. Thus providing some degree of flexibility through programmability.

FPGA SoCs

FPGA SoCs provide a complete system solution, featuring an embedded microprocessor tightly coupled with an FPGA on the same IC, as shown in Figure 2.12. These devices are therefore ideal for edge computing, combining the high-level management functionality of embedded processors with the compute acceleration of a custom architecture on an FPGA. The embedded microprocessor can not only be used for light computing but to also configure and control the FPGA dynamically after deployment. Many previous work implementing co-processors for Neural Networks on FPGAs have targetted SoC platforms in a manner that the microprocessor manages the data transfer, control and runtime coordination [53–55, 72]. Others have implemented

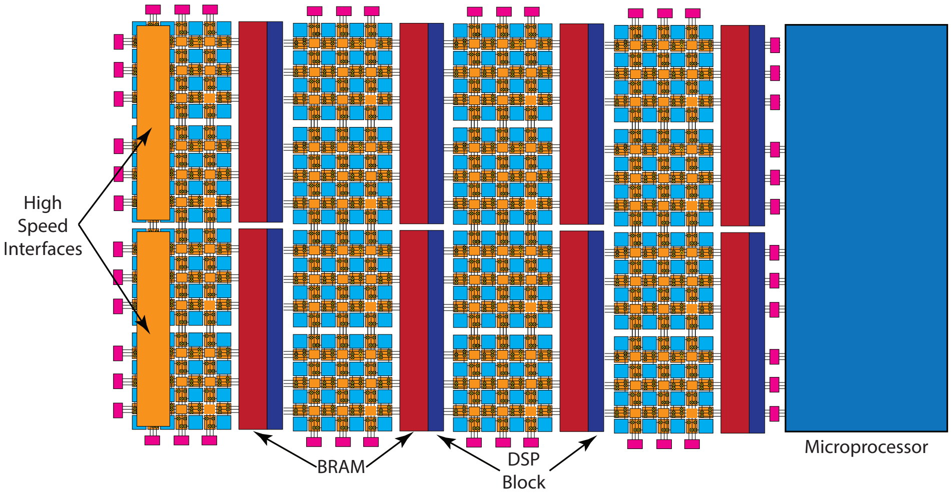


Figure 2.12: FPGA SoC architecture showing the reconfigurable fabric along with a microprocessor [65].

complete compute architectures on the FPGA fabric, benefiting from all the advantages of custom compute architecture, while leaving the microprocessor free and potentially deal with other tasks [73].

2.5 Compute Optimisations

Neural Networks' ubiquity has led to a significant interest in optimising this class of algorithms for more efficient processing. Previous work in the literature has analysed NN models and shown that they are typically over-parametrised, thus incorporating significant redundancy. Various compute optimisations have been explored, described in the following subsections, as a result of this observation, that aim for a more efficient use of compute and memory resources.

2.5.1 Scheduling - Batch Inference

NNs are typically demanding in terms of workload and memory bandwidth. Even lightweight networks are typically too large to be fully unrolled on the compute units of embedded custom compute architectures or fit in their on-chip memories. Generic and custom compute architectures may therefore employ more complex scheduling techniques in order to make these computations more efficient in terms of compute resources utilisation and memory transfers [50, 63]. The most popular technique to process these networks is in batches, i.e. on a group of inputs rather than one input at a time. Similarly to the batch size hyperparameter during training. Specifically, a large NN is usually partitioned according to the capabilities of the targetted compute unit. Each partition weights are cached to the compute unit and computations take place for a number of inputs, generating partial results of the NN. Consequently, the following partition is loaded and cached to the compute unit, calculating its own partial results based on the previously generated ones. This process is repeated until the whole Neural Network is processed. Batch processing in essence alleviates the overheads of loading weights for each input inference and is ideal when there are huge volumes of stored data. This method is very useful to accelerators with high speed PCIe interconnect (i.e. GPUs), in which the high transfer bandwidth is constantly filled with the available stored data. The availability of data in GPUs is paramount since these devices need to cache adequate data in their local memories to keep their compute cores occupied, and maintain high throughput. Otherwise, the available compute cores will be underutilised, resulting in poor performance and efficiency. Batch processing, however, is not always suitable for real time processing with streaming flow of data, e.g. data collected from a sensor on an edge device. The latter calls for custom compute architectures that are tailored to a streaming dataflow, making arrangements accordingly to be more efficient.

2.5.2 Pruning

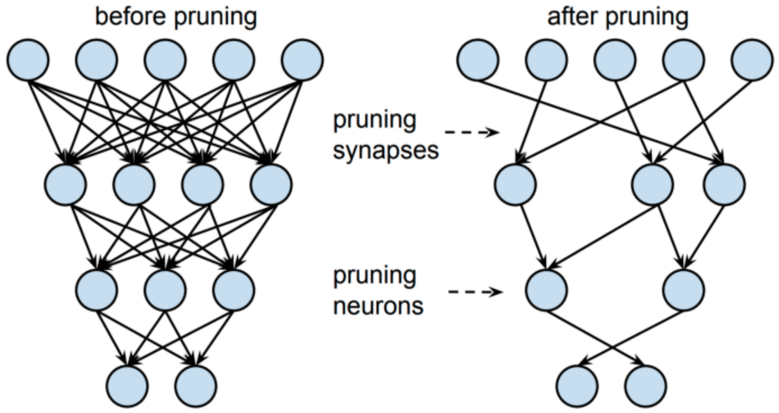


Figure 2.13: Pruned Neural Network example, showing weight (synapse) and neuron pruning [74].

Pruning involves the detection of weights, also known as synapses, or neurons whose impact to the Neural Network is minor and essentially remove them, as shown in Figure 2.13. Weight pruning can be implemented by setting these weights to zero, resulting in a sparse network, whose topology remains the same. Weight pruning can be very effective when a compute architecture can take advantage of its sparse nature and skip computations with zero values, which will in turn reduce runtime, in addition to its smaller size. The effectiveness of the compute architecture however, also depends on the degree of sparsity in the network. Neuron pruning, on the other hand, would result in a smaller network topology which will remain dense. Although the regularity of dense models has a better support in compute platforms, neuron pruning has typically a more negative impact on the networks' accuracy. Pruning in general can be more effective on serial and low core edge computing devices (i.e. embedded CPUs) in which computations can be skipped at their core. GPUs on the other hand may not be very effective due to their highly parallel nature. Specifically, previous work in [75] achieved 1.5× inference time acceleration with 90% of pruning rate. Alternatively, custom parallel compute architectures need to implement supporting logic, tailored datapath and compute units to effectively take advantage of pruned networks and reduce inference time [76].

2.5.3 Reduced Precision-Quantisation

The over parametrisation of NNs can also be exploited by reducing the precision of the computations. Instead of using floating point 32 bit types, the computations may use fixed point representation on reduced wordlength (e.g. 8 bit) that can be more efficient. More specifically, although most modern CPUs have bridged the execution time gap between floating and fixed point computations, fixed point arithmetic is simpler and will always be accessible on severely constrained devices at the edge, where a floating point unit may not to be implemented. The more complex floating point unit is also expected to consume more energy compared to its fixed point equivalent. In addition, the wordlength reduction results in less memory requirements in terms of storage and bandwidth, resulting in reduced external memory accesses which are slower and energy consuming. A form of quantisation is shown in [37], in which the authors quantised the weights and biases to power of two values in order to avoid the use of multipliers. On more advanced generic compute architectures, multiple reduced precision computations can be fused instead of one full precision by incorporating the Single Instruction Multiple Data (SIMD) paradigm. Nonetheless, this optimisation is more beneficial on custom compute architectures that can shrink their datapath, compute and memory resources to the specific reduced precision wordlength.

2.6 Enabling Faster Deployment on FPGAs

Although FPGAs are reconfigurable, detailed hardware implementations in RTL involve time consuming low level design effort. Moreover, the lengthy back-end tool compilation times, add to a less rapid and dynamic deployment of custom computing architectures on FPGAs. To this end, various techniques and paradigms have been proposed over the years, which are described in the following subsections. In addition, the emergence of new domains, like Neural

Networks, have also created a gap between training a network and mapping it to a compute architecture on an FPGA. As a result, automated toolflows have also been proposed that mainly take advantage of rapid deployment techniques to provide an end-to-end framework that maps a Neural Network on an FPGA.

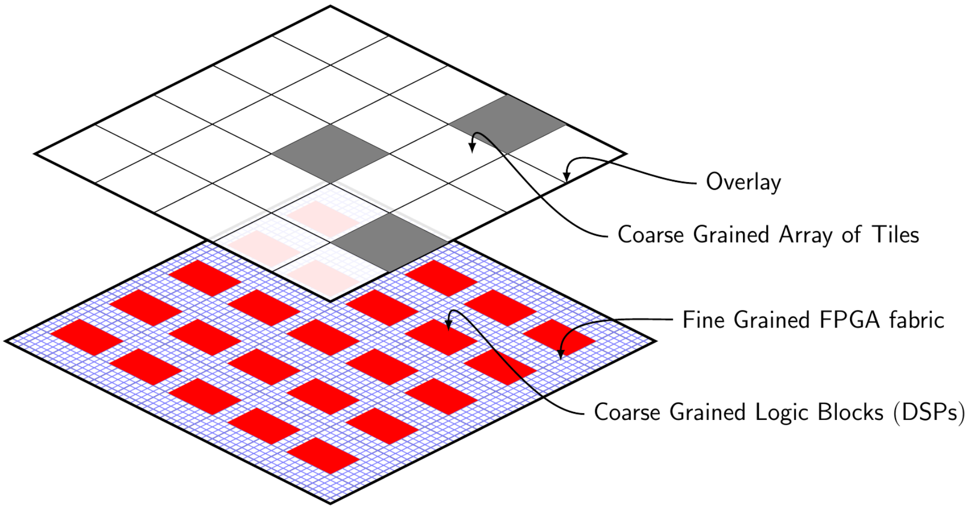
2.6.1 High Level Synthesis (HLS)

High Level Synthesis (HLS) has enabled the generation of custom compute architectures from a higher level language, for example C/C++. Moreover, behavioural verification through a high level written testbench is also faster, compared to an HDL written one. HLS also features architectural optimisations that enhance the performance of the architecture in terms of latency, throughput, area and resource utilisation. Parallelism, for example, can be exploited by unrolling compute loops while memory resources can be formatted in different ways using various partition factors. All these optimisations can take place by using simple instructions in the high level code. HLS has effectively reduced the overall design time, by automatically translating the high level code to low level HDL architecture. The use of higher level language, however, translates to less low level design optimisations. As a result, HLS generated compute architectures may be less efficient in terms of power, performance and resource utilisation [77, 78]. Nonetheless, the lengthy backend compilation time is still required in order for the FPGA bitstream to be generated. In addition, although HLS is written in a high level language, functional knowledge of digital design is still required, since high level programming concepts in generic compute architectures do not exist in digital system design. For example, the concept of dynamic memory allocation in software programming does not exist in digital design, i.e. the memories are fixed in the architecture and cannot be dynamically allocated [65]. HLS is increasingly used in many published pieces of work, for example in [14] to implement an NN accelerator in an SoC design and in [50, 51] to implement an LSTM variant, among others.

2.6.2 Overlays

Overlays have been proposed as a way of enabling high level programming with rapid compilation and predictable performance on FPGAs. When designed in an architecture-centric manner, overlays can achieve near the theoretical maximum frequency supported by underlying FPGA architecture, while scaling to large overlay sizes [80]. Meanwhile, compilation does not involve the FPGA backend flow and so can be very fast, lightweight and vendor independent.

Overlays enhance flexibility in custom computing architectures by forming a coarser grained abstraction on top of the FPGA fabric, as shown in Figure 2.14. As a result, overlays do not need to repeatedly go through the lengthy



Overlay
Coarse Grained Array of Tiles
Fine Grained FPGA fabric
Coarse Grained Logic Blocks (DSPs)

Figure 2.14: Coarser grained overlay architecture on top of the finer grained reconfigurable fabric [79].

compilation time required by the backend toolflow. Performance can be more predictable as it is closely tied to the fixed performance of functional units, and routing overhead can be reduced by taking into account the regularity of the required data movement [80]. The authors in [81] present a family of overlay architectures and associated design methodology. By using datapath merging, they minimise the added overhead to support various computations while also providing optional adjustable flexibility through a secondary interconnect network. Their experiments demonstrate faster runtime compilation and reduced area utilisation, though resulting in reduced operating frequency due to the slower operators occurring in the same context as faster ones. Further performance improvement in overlays can be obtained when tailoring the architecture to heavily take advantage of the high performance DSP blocks, that are abundant in modern FPGAs [80].

2.6.3 Neural Network Toolflows

The emergence of Neural Networks has created the need to automate their mapping to the various compute platforms in order to shorten their design time, by abstracting their low level computations to higher level building blocks. In order to bridge the ease of deployment gap between software programmable platforms and FPGAs, automated toolflows have been proposed that provide an end-to-end mapping of neural networks to custom architectures.

The various toolflows that have been proposed in the literature have approached this task in different ways. The work in [82] has explored the use of roofline model to perform a design space exploration in HLS, based on an input CNN topology. The proposed toolflow outputs a custom compute architecture based on the design space exploration and specified user constraints. Thus a separate compute architecture is generated for every CNN, using the lengthy backend toolflow each time. The latter, in turn, results in the reconfiguration of the FPGA with a different bitstream for each CNN. In a similar context, hls4ml provides a tool for end-to-end FPGA implementations of ML models [83]. Specifically, the tool automates the generation of HLS based accelerators from high level Python programming language. User specific optimisations can be applied to tailor the generated architecture for a specific use case. However, this method also generates a separate compute architecture for each model, requiring a lengthy backend toolflow run for each model.

An accelerator implemented in an SoC architecture that consists of a tunable number of compute clusters is proposed in [84]. The compute clusters use 16 bit wordlength, are flatten in one dimension and are time-multiplexed to process a neural network. The compute acceleration is supplemented by a compiler that translates a neural network to a series of instructions, executed by the custom architecture. This work therefore favours programmability, over a more custom architecture, while providing a degree of performance and area customisation by offering a tunable number of compute units.

The work in [85] takes advantage of the redundancy in neural networks to initially generate a binarised network equivalent for given CNN topology. Binary neural networks are networks in which part or all computations are converted to single-bit values. This extreme quantisation has led to significant efficiency improvement in all aspects, performance, resource utilisation and energy, at the cost of flexibility. The proposed solution also comprises a framework that automates the process and focuses on generating a custom compute unit for each layer, which are then pipelined to form the complete network. Each processing unit's performance is tailored to match the throughput of preceding and following units, providing a balanced performance throughout the network. The latter avoids inefficiencies that may occur as a result of different throughput or latency between the layers in a network due to their different workload. Binarising a model results in reducing its compute and memory requirements significantly, enabling them to fit on-chip. The proposed framework generates a compute architecture for each unit, which in turn results in the reconfiguration of the FPGA for each network. Based on FINN, FINN-L is introduced in [51], which is a library extension of the former that supports a variant of LSTM networks.

The Xilinx Deep Neural Network Development Kit (DNNDK) is an example of a vendor flow for accelerating NN inference on an accelerator architecture on FPGAs. It comprises a more generic NN computing architectures, like the Deep-learning Processor Unit (DPU), to offer a more balanced performance acceleration to flexibility and area ratio. DNNDK includes model compression, by using data quantization and pruning, to more efficiently process NN inference. The various optimisation techniques have contributed to more efficient FPGA implementations of NNs [86].

2.6.4 Summary

Neural Networks, and Machine Learning in general, are constantly driving a growth of new applications that are shown to be more accurate than their hand coded equivalent ones. As a concept, they are ideal in a data driven processing era from which huge volumes of knowledge can be extracted in an automated manner. The various NN layers and learning techniques provide a wide spectrum of learning abilities for different data patterns. As a result, there is great interest in accommodating this class of algorithms in a wide range of devices, from powerful servers to edge computing. The latter domain presents the most challenges and calls for a more systematic approach. FPGAs have demonstrated their suitability in accommodating custom computing architectures, compared to other approaches. Tightly coupled microprocessors on FPGA SoCs offer a complete system solution which renders them ideal for computing at the edge, combining the generic CPU architecture with a custom one on the FPGA. In addition, the abundance of DSP blocks in modern FPGAs provide the means to unroll highly parallel computations. DSP blocks can offer high performance while consuming less power than their equivalent ones in fabric. Architecture centric approaches were shown to more efficiently utilise all aforementioned FPGA resources to provide solutions that offer improved performance and energy efficiency. Nonetheless, detailed hardware implementations in RTL involve time consuming low level design effort. Although various automation tools have been proposed by research groups and vendors, these still require lengthy hardware recompilation for each NN topology. This calls for using new methods that reduce design turnaround time or reuse the low level design in a flexible, parametrised and scalable manner. The overlay approach enables mapping to an independent intermediate architecture that does not require low level hardware compilation for different networks and does not rely on vendor tools for application mapping.

Chapter 3

Accelerating Neural Network Based Network Intrusion Detection on FPGA

3.1 Introduction

The Internet of Things (IoT) is driving an exponential growth in connectivity between lightweight embedded systems. These devices are often severely computationally constrained, being designed to fulfil a single task well. This increased networking presents a challenge, however, in terms of network security, since these devices can expose a wider attack surface on account of not being as rigorously engineered as more complex systems. Indeed, the use of IoT devices as a tool in cyberattacks was exemplified by the Mirai malware in 2016, among other cases.

Traditional network security has aimed to provide confidentiality, integrity, and availability of resources to authorized users. This has often occurred in more controlled network environments such as corporate networks, where firewalls serve as a secure point of interface with open networks. Even in such cases, the possibility of an internal system being compromised requires monitoring for attacks of all traffic, even from within the network.

Intrusion Detection Systems (IDSs) collect and analyse information from the systems within a network for malicious attack detection. Detection can be logged as an event of interest or trigger a defense mechanism to deal with the event in real time. Mainstream IDSs use pattern matching, string matching, multi-match packet classification and regular expressions for operation [87]. These computationally complex approaches are often implemented using hardware accelerators on FPGAs or ASICs, or run on highly parallel computing platforms such as multi-core processors or GPUs to enable them to process network traffic at the high rates required. Hence, such complex systems are usually integrated within the network infrastructure of large organisations.

The limited computing power of embedded systems means IoT devices will often not incorporate significant security capabilities at the nodes, making them an ideal target for malicious attacks. With such devices being deployed in less controlled environments, and without access to significant infrastructure, more lightweight approaches to such security mechanisms are required.

In this chapter, a Network Intrusion Detection approach based on Machine Learning is explored, specifically Neural Networks (NNs), that provides flexibility to evolve to emerging attacks. This chapter demonstrates how this can be implemented on a lightweight Xilinx Zynq FPGA SoC to process packets at line rate while enabling model parameter updates to adapt to changing requirements.

3.2 Background

Intrusion Detection Systems (IDSs) can be divided into two categories, according to the detection method used:

- **Signature (or misuse) based:** Captured data is compared against a database containing signatures of known attacks.

- **Anomaly based:** Captured data is compared against a model of the expected normal behaviour of the system. If a deviation is observed then an attack has been detected.

Signature based IDSs are widely used in commercial systems because of their accurate detection of known attacks, while anomaly based systems are prone to generating false classifications. Signature based IDSs, however, fail to detect unknown (zero-day) attacks. There can also be a significant delay for a new attack to be detected and its signature generated and distributed in an update [41]. Moreover, signature based systems must consider a large database of signatures, requiring substantial memory and computational power. Hybrid implementations of signature and anomaly based IDSs present a more robust approach since one method complements the other, though these still require significant computing power.

Intrusion Detection has been an appealing domain for Machine Learning (ML) algorithms in general. The strongest incentive lies in the ability of ML algorithms to generalize their learned pattern to new, unknown data. Thus, ML algorithms have the potential to detect new, zero-day, attacks and modified known attacks. It is also worth considering that IoT, as a developing domain will entail evolving (normal) traffic patterns as it finds more uses, so the safe patterns of communication are themselves evolving, and hence an adaptable approach to intrusion detection is needed.

Many ML approaches can be computationally intensive, as described in Chapter 2, hindering their adoption in embedded systems and compromising real time detection [41]. Neural Networks, and specifically Deep Neural Networks (DNNs) suffer from high computational complexity and complex training. Hence the work in this chapter focuses on (shallow) NNs to limit the computational complexity of the proposed system in order to achieve real time detection. Shallow NNs have successfully been applied in a broad range of fields, from the automotive domain [39], to healthcare [7], and are a very good fit for simple event classification or detection. Furthermore, their flexible topology enables tradeoffs between detection accuracy, performance and area, resulting in a highly customizable architecture for hardware implementation.

As the functionality of Machine Learning models is defined during training, the dataset used becomes very important. A flawed dataset means that the ML model will extract flawed patterns, that may not be applicable or representative of the intended application. This will in turn result in very poor accuracy when the trained model is deployed to classify new data. Datasets used for intrusion detection fall into two broad categories, private (or custom) and public datasets. Privately generated datasets may contain more realistic data for training and testing as in most of the cases they are created from the specific scenario that are to be applied to. Moreover, they can be tailored to a specific attack detection by manipulating the number of records in each class accordingly, while public datasets may lack a sufficient number records for a specific attack type. Proprietary and commercially sensitive datasets, however, are not available to researchers. Publicly available datasets, on the other hand, are widely used and, as a result, thoroughly tested [88]. They constitute a safer choice to avoid potential flaws and, more importantly, they provide a means to compare with previous work using the same datasets.

3.3 Related Work

Network security has sustained interest in the research community and IDSs using a variety of approaches have been proposed. Acceleration of pattern matching on FPGAs has been explored in [66, 67]. The work in [89] proposed an approach using Principal Component Analysis (PCA) with features extracted from network traffic, which was tested on the publicly available KDD Cup 1999 dataset. The IDS was implemented on a Xilinx Virtex II Pro FPGA and achieved a 23.76 Gb/s throughput with an attack detection rate of over 99%. In [68], the authors present an energy efficient implementation of Decision Trees (DTs) on an Altera Cyclone IV. Their work covers two test cases: the first classifies the NSL-KDD dataset using 9 manually selected features out of 41, achieving a 96.5% accuracy on the train set and 77.8% on the test set.

The second detects probe attacks on a custom dataset, misclassifying only 3 out of the 37548 instances in the test set. The hardware implementation of the probe attack detection DT is 15.4× better in throughput while consuming only 0.03% the energy of its software equivalent on an Intel Atom CPU. The authors further expanded their work using their custom dataset to evaluate 3 machine learning classifiers in a similar manner in [69]. In this case, their fastest classifier in hardware was 926× faster while consuming 0.05% the energy of its equivalent in software. The work in [70] showed how security primitives could be built into network controllers to enable enhanced security.

In broader work in neural network implementations, the work in [17] combines deep and shallow learning for Network Intrusion Detection based on Non-symmetric Deep Auto-Encoders (NDAE) and Random Forests (RF), tested on the KDD Cup 1999 and NSL-KDD datasets. This approach demonstrates promising detection results with less training time compared to a Deep Belief Network (DBN) implementation. The NN in [18] detects Distributed Denial of Service (DDoS) and DoS attacks offline. The authors use a custom dataset to train a three-layer (shallow) NN for binary classification (normal-DoS/DDoS) and test it in a simulated IoT network, demonstrating a 99.4% accuracy. In [19], two NNs are trained on the UNSW-NB15 and NSL-KDD datasets to detect DoS attacks using only input features relevant to such attacks. The authors determined the number of neurons in the hidden layer experimentally, and demonstrated a DoS detection accuracy of 99% on NSL-KDD and 97% on UNSW-NB15.

The work in [40] presents two NNs trained on the NSL-KDD dataset to detect all 4 types of attack in the dataset (DoS, Probe, R2L and U2R). The first NN categorizes records between normal and malicious, while the second classifies the malicious records into types (5-categories). The authors experimentally determined the number of neurons in the hidden layer as well as whether to use all features or a reduced set. On the test set, for binary classification, the best accuracy of 81.2% was obtained using a subset of the input features, while for attack classification the best accuracy of 79.9% was obtained using all features. The authors in [41], similarly use two shallow NNs trained on the KDD Cup 1999 dataset for binary and attack type classification. The NNs use 36 of the 41 features demonstrating an average precision of 98.86% for binary classification and 95.05% for the attack type classification.

A detailed review on IDSs that employ deep learning is presented in [90], the most relevant of them to this chapter found in [20, 22, 23]. In [22], the authors used a Recurrent Neural Network (RNN) on the NSL-KDD dataset using all provided input features for binary and attack type classification. They determined the optimal number of hidden nodes and learning rate in each case experimentally. For binary classification, the authors obtained 99.81%

and 83.28% accuracy on the train and test set respectively using 80 hidden nodes and a learning rate of 0.1. Their proposed 5-category classification model obtains 99.53% and 81.29% accuracy on the train and test set respectively using 80 hidden nodes and learning rate of 0.5.

Although CNNs are primarily used for image recognition tasks, the work in [23] proposes an approach that uses CNNs to classify the NSL-KDD dataset. The authors apply an image conversion technique that maps all the input features of each record in the dataset to an image. The input features are initially transformed to a binary vector space and then to an 8×8 grayscale image. The authors used Tensorflow to implement 2 popular CNN models, ResNet 50 and GoogleNet, obtaining 79.14% and 77.04% on the test set for binary classification respectively.

Lastly, the authors in [20] present a Deep Neural Network (DNN) approach using 6 raw features out of the 41 in the NSL-KDD dataset, achieving 91.62% and 75.5% accuracy on the training and test set respectively. Using the same number of raw features, the authors applied their methodology to Deep RNNs in [21] obtaining 89% accuracy on the test set.

The topology configurations of the NNs described above are summarised in Table 3.1, where available.

Citation	**Configuration**
Tang et al. [20]	6-12-6-3-2
Hodo et al. [18]	6-3-1
Idhammad et al. [19]: UNSW	6-7-1
Idhammad et al. [19]: NSL-KDD	5-6-1
Ingre and Yadav [40]	29-21-2
Ingre and Yadav [40]	41-23-5

Table 3.1: Network configurations in related work.

3.4 Experimental Methodology

In the context of intrusion detection, Neural Networks have the potential, ideally, to be updated after deployment or tailored (fine-tuned) to a specific device's network traffic. These updates can be applied through new coefficients for the same model topology, assuming that the model has the capacity to support this.

Section 3.3 shows that tailoring an NN to detect only a single type of attack or all the attacks in one category can result in better accuracy. Moreover, selecting the most relevant features from the dataset decreases the dimensionality and this in turn enables NNs to perform better. Hence, the proposed NN is trained to detect all types of attacks in one category, binary classification

(Normal-Anomaly), using a selected subset of the available features.

3.4.1 NSL-KDD Dataset

The publicly available NSL-KDD dataset is used, a labelled dataset for supervised learning. It is an updated version of the KDD Cup 1999 dataset, addressing its shortcomings [91]. While the dataset is not directly related to IoT applications, it is widely used, enabling comparisons with previous work. The approach presented here can be applied to any future public dataset which can be used to retrain the network for IoT specific traffic patterns. The dataset is divided into the *train* and *test* sets which contain data for normal and malicious traffic. Each entry comprises 41 features categorized into 3 groups [91]:

- **Basic features:** features that are extracted from a TCP/IP connection.
- **Traffic features:** features that are generated within a window of the last 100 connections, to enable detection of longer probe attacks. These features provide an element of time-domain memory. Traffic features are further categorized into service and host based.
- **Content features:** features that are extracted from the packet's data and provide the means to detect attacks with infrequent sequential patterns.

The train set contains 22 attack types, divided into 4 main categories: DoS (Denial of Service), Probe, R2L (Remote to Local), and U2R (User to Root). In the test set, there are 17 additional attacks that fall into the same 4 categories. In this way, the ability of the NN to generalize its learned pattern to unknown data is put under test.

In order to fairly train the model, categorical features are mapped to a one-hot encoded representation for the training phase, mitigating the possible bias introduced by ad-hoc numerical mapping.

3.4.2 Software Implementation

TensorFlow [56] was used to train an NN with 29 input features, 21 hidden neurons and 2 output neurons, similar to that by Ingre and Yadav [40]. Of the 41 input features, Bajaj and Arora [92] concluded that 8 of them have little or no impact in attack detection, while Ingre and Yadav [40] noticed that the values of 4 other features are close to 0. The selected features span all types of features in the dataset. This enables the NN to extract patterns in the time domain using Traffic Features, thus avoiding the use of more computationally complex machine learning models that do so with raw features, such as Recurrent Neural

Networks (RNNs). Out of the 29 selected features, 3 are categorical, and hence the number of input layers increases to 110 after one-hot encoding.

The relatively simple and inexpensive Rectified Linear Unit (ReLU) activation function is used, that can be easily implemented with a comparator as shown in equation 2.1, instead of more complex functions that include divisions and exponents, such as the sigmoid and tanh functions, as shown in equations 2.2 and 2.3.

The proposed NN was trained with the Adam optimizer, using the cross entropy loss function (that also includes softmax) with weights and biases randomly initialized. Training hyperparameters, such as the learning rate and batch size, were determined experimentally. For fair comparison, the same randomly initialized weights and biases are used for all the experiments. Subsequently, experimental runs were made using 3 learning rates (0.01, 0.001, 0.0001) on four different batch sizes (32, 64, 128, 256) for a total of 5 epochs. The classification performance of each run is evaluated using accuracy, as described in Section 2.3. The highest accuracies obtained after one epoch are summarised in Table 3.2. While the proposed NN is trained on the train set and tested on the test set, the results in Table 3.2 are selected by prioritizing the accuracy obtained from the test set across runs.

Batch Size	**Learning Rate**					
	0.01		**0.001**		**0.0001**	
	Test	**Train**	**Test**	**Train**	**Test**	**Train**
32	77.61	89.15	**80.52**	**96.02**	80.37	89.09
64	73.16	94.71	80.64	94.05	80.29	93.62
128	76.65	93.09	79.01	96.62	79.80	91.99
256	77.56	94.49	80.84	94.22	77.47	94.06

Table 3.2: Accuracy results for training parameters.

From the results in Table 3.2, the learning rate of 0.001 and batch size of 32 provides the optimal combined accuracy across the test and train sets. This results in the confusion matrix of the test set in Table 3.3.

Predicted Class	**Actual Class**	
	Normal	**Malicious**
Normal	9257	3937
Malicious	454	8896

Table 3.3: Test set classification results.

Compared to other work in the literature that use the NSL-KDD dataset, with which a direct comparison can be made, the proposed model accuracy is close to that by Ingre and Yadav [40], where the authors report 99.3% and 81.2% accuracy on the train and test sets respectively. It is worth noting that the authors in this case normalized the dataset prior to its use. While data normalization has been proven to enhance the accuracy of NNs, it also entails additional workload during inference. Tang et al. [20] use a DNN with 6 input features, reporting 91.62% and 75.75% accuracy on the train and test sets respectively. This shows that deep models that use a small subset of the input features do not necessarily outperform shallow models that use more features. All the referenced systems in this chapter that use the NSL-KDD dataset are summarised in Table 3.4, along with their configurations.

Citation	**ML Model**	**Classification**	**# Features (out of 41)**	**Accuracy %**	
				Train Set	**Test Set**
[68]	DT	N/A	9	96.5	77.8
[40]	NN	Binary	29	99.3	81.2
[40]	NN	5-Cat.	41	98.9	79.9
[20]	DNN	Binary	6	91.62	75.75
[21]	D-RNN	Binary	6	N/A	89
[22]	RNN	Binary	41	99.81	83.28
[22]	RNN	5-Cat.	41	99.53	81.29
[23]	CNN-ResNet50	Binary	41	N/A	79.14
[23]	CNN-GoogleNet	Binary	41	N/A	77.04
Proposed	NN	Binary	29	96.02	80.52

Table 3.4: Accuracy comparisons on the NSL-KDD dataset.

3.4.3 Hardware Implementation

Unlike previous work, the aim of this chapter is to build a fully functional embedded IDS to perform these classifications in real time on network data. Hence, the trained NN was used to build a working hardware system for this purpose. Vivado HLS (version 2016.4) was used, targeting the Xilinx Zynq Z-7020 FPGA as found on the Xilinx Zedboard, to implement the intrusion detection hardware. This is a modest FPGA SoC device including a flexible FPGA fabric tightly coupled with an ARM Cortex-A9 processor subsystem, as shown in Figure 3.1. This system is designed to act as an IoT gateway, securing the network for a group of less capable devices. The peripherals, e.g. Ethernet Phy and SD card, are connected through Multiplexed I/O (MIO) interconnect to the ARM core and 512 MB of DDR3 memory is attached through the DRAM controller. This flexible connectivity enables runtime processing of network data by forwarding packets to the accelerator, or processing them in software. For testing and verification, it allows the test set and model coefficients to be stored on an SD card, to be transferred to memory and then to the accelerator over DMA.

Most work on optimising FPGA implementations of neural networks con-

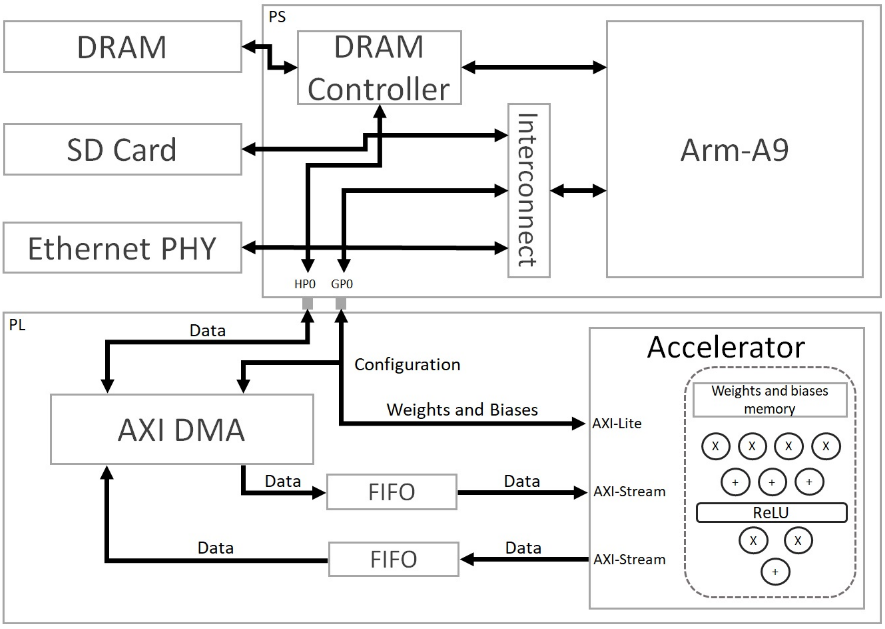


Figure 3.1: Overview of the Xilinx Zynq based system architecture.

siders fixed network parameters. Network pruning, data quantization, and reduced arithmetic precision can be exploited to trade off performance, power consumption, and detection accuracy [47, 93, 94]. However, this comes at the cost of flexibility as any change in network parameters requires a new design exploration and hardware implementation process. The architecture presented in this chapter is designed to be flexible, by allowing the coefficients to be modified at runtime, thereby enabling the same hardware to be used to detect different or evolving attacks without the need for additional design space exploration or hardware optimisation.

Vivado HLS allows us to exploit the inherent parallelism in the NN structure using pragmas to unroll loops for maximum parallelism and performance, without the need for low level Hardware Description Language (HDL) design. The inputs and intermediate results are represented in single precision floating point (IEEE-754), as the architecture is designed to retain flexibility to accept newly trained model parameters. The accelerator operates in one of three modes: IDLE, LOAD, or COMP. It starts in the IDLE mode where it can make a transition to LOAD or COMP. Transitions between states are triggered from the ARM core over AXI-Lite since these are not time critical operations.

In LOAD mode, the coefficients (weights and biases) of the model are modified to update the NN at runtime, which is done over AXI-Lite using the 4 accelerator inputs:

- **mem_sel:** selects the memory bank to configure. (i.e. first layer weights, first layer biases, etc.)

- **dimA:** indexes the first dimension of a 2D array, or the only dimension in a one-dimensional array.
- **dimB:** indexes the second dimension of a 2D array.
- **coeff_in:** value of the coefficient to be stored.

The digram of the proposed IDS is shown in Figure 3.2.

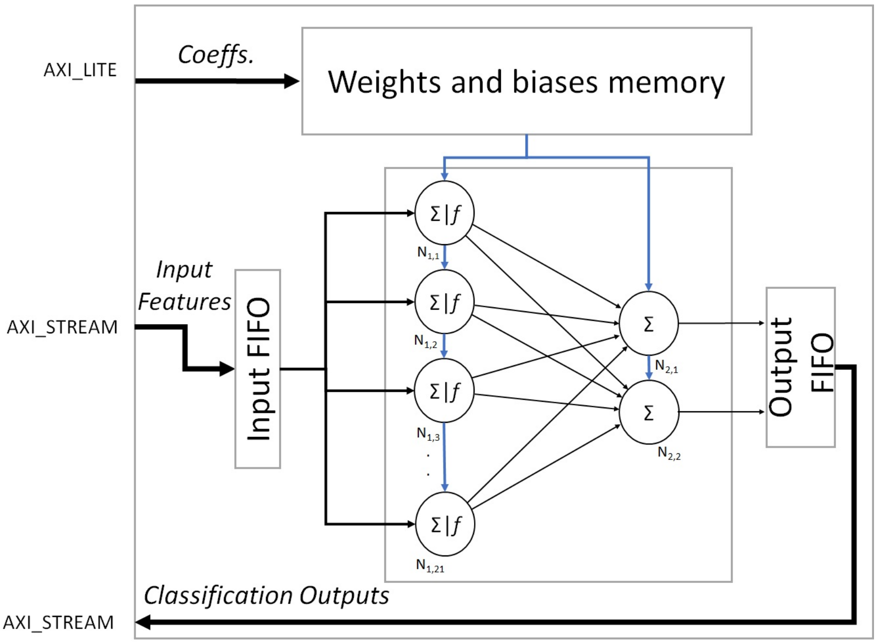


Figure 3.2: Intrusion Detection System diagram, showing the various memories, neurons and connectivity.

FPGAs support flexibility through reconfiguration by loading alternative bitstreams that modify the hardware on the FPGA [95]. One method for using different NN models would be to generate multiple bitstreams and load them as needed. However, this would entail the separate design and compilation of these optimised hardware models and would not allow for easy modification of model parameters to deal with emerging attacks. The Xilinx Zynq allows the programmable logic (PL) configuration to be changed by the processor system in software, taking around 30 milliseconds. One way to reduce this time is to partition a section of the PL for this accelerator and reconfigure only that, through Partial Reconfiguration (PR), and using an optimised reconfiguration controller to reduce the time to below 10 milliseconds [96]. Hence, the work in this chapter retains full flexibility by implementing a general datapath with reprogrammable coefficients, rather than tightly optimising the datapath around a fixed set of coefficients.

The time needed for the configuration of all 2375 coefficients was measured to be 2.273 ms. This includes the time needed for the ARM core to iterate through the data, increment its indexing variables, configure the accelerator accordingly and make the appropriate checks as indicatively shown in Listing 3.1. Updating the coefficients is not considered a time-critical operation as one configuration of the IDS is expected to be active for a large volume of network data. Nonetheless, the proposed approach offers competitive reconfiguration time compared to reconfiguring the hardware, while offering a much more flexible implementation that allows coefficients to be updated directly, from software without the need for vendor tools and a full hardware compilation.

```
//Sets the accel to the LOAD state
Set_write_en_V(&ids_nn,0x1);

//Indexes the memory of the 1st layer weights
Set_mem_sel_V(&ids_nn,0x0);

//Loads the weights of the first layer
for (i=0;i<(num_inputs-3);i++) {
    //Indexes the weights memory
    Set_dimA_V(&ids_nn,(u32)i);
    for (j=0;j<neurons_1st_layer;j++) {
        //Indexes the neuron's memory
        Set_dimB_V(&ids_nn,(u32)j);

        //Sets the weight
        Set_coeff_in(&ids_nn,float_to_u32(weights_layer1[i][j]));

        //Reads the value from the accel.
        temp_float=u32_to_float(Get_coeff_in(&ids_nn));

        //Checks that the value has been set
        while(temp_float!=weights_layer1[i][j]) {
            temp_float=u32_to_float(Get_coeff_in(&ids_nn));
        }
    }
}
```

Listing 3.1: Setting weights using the ARM processor.

The Intrusion Detection process takes place in the COMP state. To mitigate the increased complexity due to the one-hot encoding, the fact that only one of each one-hot encoded features is used at a time is exploited. During inference, integer representation is used for each attribute and in each case only the index of the active attribute is needed. The index of the active attribute is used as an address to a Look-Up-Table, that outputs the corresponding weight. This restores the number of input features needed for inference from 110 to 29, while also avoiding redundant multiplications by 0 caused by the inactive attributes in each one-hot encoded feature. Meanwhile, any multiplication by 1 of each active attribute is replaced with a low latency table look-up. The 29 input features along with the 2 output results (corresponding to the normal/malicious score), are interfaced with the ARM core through 2 separate AXI-Stream ports with the data transferred sequentially in consecutive clock cycles.

The timing results of the implemented design from HLS are shown in Table 3.5 while the resource utilisation on the Xilinx Zynq device is shown in Table 3.6.

Frequency (MHz)	**Latency (Clock Cycles)**	**Initiation Interval (Clock Cycles)**
76	237	29

Table 3.5: Timing results for NN accelerator.

The initiation interval of 29 clock cycles is bounded by the number of input features that need to be read through the AXI-STREAM port.

	LUTs	**FFs**	**DSPs**	**BRAM**
Utilised	26463	56478	111	88
Available	53200	106400	220	280
% Utilisation	50	53	50	31

Table 3.6: Resource utilisation on the Xilinx Zynq Z-7020.

The proposed system, shown in Figure 3.1, uses 2 FIFOs on each AXI-Stream port of the accelerator to act as buffers. Data is transferred to and from the AXI-Stream ports through the AXI-DMA that is interfaced with the PS using the HP0 (High Performance 0) port. The HP0 port, in turn, using the DRAM controller, transfers data to and from DRAM. The configuration of the accelerator coefficients as well as the configuration of the AXI-DMA take place using AXI-Lite ports, which are interfaced with the PS through the GP0 (General Purpose 0) port. In order to test the full system, the test dataset along with the weights and biases obtained from the trained model were stored on an SD card and made available to the ARM core using the FAT filesystem library.

3.5 Results and Evaluation

To evaluate the performance of the proposed IDS in practice, Vivado (version 2016.4) has been used to implement the system as shown in Figure 3.1. The AXI-TIMER IP, operating at 100 MHz, was used to measure the execution time. To evaluate the accuracy of IDS in practice, the coefficients and test dataset in the SD card were read from the ARM core, transferred to DRAM and then fed to the accelerator. Consequently, in COMP mode, the dataset was read and fed to the accelerator. In order to provide a reference for comparison, the execution time of the proposed IDS on a single core of the ARM-A9 (bare-metal) was recorded. To demonstrate and evaluate the benefits of utilising the Look-Up-Table mechanism, the execution time of the unoptimized software implementation on the ARM core is also provided. This version of the NN uses 110 inputs at the input layer and goes through a number of redundant multiplications as described in the Section 3.4.3. This work is also compared with the test time of the NN used for DoS detection by Idhammad et al. [19], as there is adequate information for an objective comparison to be made. Although the authors use a different dataset and focus only on DoS attacks, the focus is only with the execution time and corresponding workload.Idhammad et al. [19] use the Keras and Theano frameworks in Python, running on an Intel Core i3 2.4GHz CPU under Debian Linux 8. The authors used 43748 records to test their NN. For a fair comparison, the authors obtained test time of 0.466 seconds is normalised according to the number of test records in the NSL-KDD (22544). The execution time of the three methods for the classification of the test set is shown in Table 3.7.

ARM-A9 [a] **@667MHz**	**ARM-A9** [b] **@667MHz**	**Accelerator** [b] **@76MHz**	**Idhammad et al. [19]**
4751.440ms	1458.1ms	9.018ms	240.136ms

[a] Unoptimised, 110 inputs. [b] Optimised, Look-Up-Table.

Table 3.7: Execution time.

The execution time of the proposed accelerator includes the time needed for the input data to be transferred to the accelerator from the DDR memory and the results to be written back to DDR memory. The use of a Look-Up-Table mechanism yields 69% reduction in the software execution time. A straightforward comparison between the proposed accelerator, which operates as a streaming engine in this case, and the optimised execution on the ARM Cortex-A9 shows a 161.7× improvement in the execution time. Compared to the unoptimized software version, the HW implementation operates 526.9× faster.

Comparing only the execution time of the proposed NN, considering the proposed optimised implementation with a 29-21-2 configuration, and the work by Idhammad et al. [19], the proposed accelerator performs about 26.6× faster. Meanwhile, the proposed optimised model on the ARM core is about 6× slower. In this case, however, the workload of the NN in [19] with a 6-7-1 configuration is significantly smaller compared to the proposed 29-21-2 configuration. Taking into account the number of multiplications and additions in each layer, as those are the most computationally intensive operations, it is estimated the NN in [19] requires 49 multiplications and 57 additions. Whereas this work includes a total of 651 multiplications and 674 additions. This amounts to 13.3× the multiplications and 11.8× the additions of the NN used in [19], while delivering 26.6× its performance. Overall, the proposed approach is able to detect more types of attack: DoS, Probe, R2L and U2R, at a faster detection rate compared to the work by Idhammad et al. [19] which focuses on detecting only DoS attacks.

3.5.1 Network Throughput and Detection Rate

A considerable aspect of an IDS is whether it can make decisions on packets at a suitable rate to ensure detection does not lag the start of an attack significantly. Ideally, such a system should be able to flag malicious packets before many of them have entered the network, so that evasive action can be taken. The time required to classify a single data record (interpacket interval), calculated by normalizing their execution time, on both the ARM core and the accelerator is shown in Table 3.7. In addition, the required minimum transmission size for IPv4, which is 576 bytes according to the Internet Protocol [97], is taken into consideration to generate the results in Table 3.8.

Transfer Rate (Packets/Second)	**Platform**	**Interpacket Interval(µs)**	**Detection Rate (Packets/Classification)**
1Gbps	ARM-A9	64.678	14.036
(217,014)	Accel	0.4	0.0868
10Gbps	ARM-A9	64.678	140.360
(2,170,139)	Accel	0.4	0.8680

Table 3.8: Detection rate in packets.

At 1Gbps, 217,014 packets per second of the minimum packet size can be transferred when the network is saturated. The accelerator offers a detection rate within a small fraction of a packet (0.0868 packets). On the other hand, the ARM core can only process one in 14 packets. While the Zedboard does not offer a 10G Ethernet interface, the performance for such a setup that might be deployed in an edge datacenter, interacting with IoT devices, is also evaluated. Newer Zynq UltraScale+ development boards do offer 10G Ethernet, meaning that the proposed design could be ported to such boards for more complex networks. At 10Gbps, a maximum of 2,170,139 packets per second can be transferred. The detection rate in this case is still within a single packet (0.8680 packets), which is 16.2× faster than the ARM core at 1Gbps and 161.7× faster at 10Gbps. The ARM core at 10Gbps only processes one in 140 packets.

These results demonstrate the benefit of the proposed hardware accelerated NN detection mechanism in terms of scaling to faster networks, while still offering the flexibility needed to accept updated model parameters for emerging threats. Porting to newer FPGA SoC devices such as the Zynq UltraScale+ would also likely offer significant runtime improvements.

3.6 Summary

This chapter presented an approach for network intrusion detection using NNs on FPGA SoCs. The topology of the NN maintained moderate computational complexity for a hardware implementation that can be deployed on a modest Xilinx Zynq device. It also allowed runtime configuration of neural network parameters to enable updates and address emerging attacks. The hardware implementation has been generated with HLS. Meanwhile, a low level optimisation on one-hot encoded features coupled with unrolling parallel operations have lead to a real time response implementation. The NN topology was trained with TensorFlow [56] using the NSL-KDD dataset, and obtained at best 80.52% accuracy on the test set. The proposed hardware accelerator performed 161.7× faster than software execution on the Zynq ARM core which has allowed it to detect malicious packets within a single packet window for 1Gbps and 10Gbps.

Chapter 4

High Throughput Spatial Convolution Filters using FPGA DSP Blocks

4.1 Introduction

Image and, by extension, video processing entail intensive computations on a large stream of input pixels. A full HD color video streaming at 60 frames per second (FPS) requires a processing throughput of over 124 million pixels per second for each channel. This rate, coupled with the numerous operations required per pixel in a typical vision flow, result in many GOPS for real time processing. Exploiting parallelism is therefore paramount to achieve real time system implementation [98]. Spatial filtering, or 2-D convolution, is a fundamental operation used in the initial stages of many vision applications, and as a result, its efficiency significantly impacts higher layers in these applications. Meanwhile, the resolution of images and videos is increasing, with 4K video now commonplace, quadrupling the computational requirements compared to full HD. In addition, the increasing popularity and wider use of Convolutional Neural Networks (CNNs) in a plethora of applications [43] makes spatial convolution even more important. As deeper CNNs with more neurons per layer are developed, the memory required to store weights and biases grows significantly. Hence, most CNN acceleration architectures buffer pixel and weight data in off-chip memory, which breaks the streaming model that is more relevant for real time streaming video applications. Moreover, some CNNs make use of varying convolution window strides, which reduces computational complexity compared to a streaming filter that processes overlapping windows. Therefore, most optimisations applied in CNN implementation do not typically apply to streaming video processing, and yet the performance requirements continue to scale. In contrast, optimisations applied to real time vision systems can be applied to CNN architectures that operate in streaming manner.

The work in this chapter shows how the use of high performance DSP blocks in generalized filter architectures can achieve high throughput that meets real time constraints. Specifically, the flexible filter designs use the DSP blocks for the pixelwise multiplication, alongside various different adder tree designs. In order to achieve high throughput, all filter architectures are extensively pipelined. Compared to previous work, the proposed filter implementations focus on maximising their throughput on FPGAs while maintaining dynamic coefficient adaptability through external register access. Specifically, the proposed architecture is built to fully exploit the DSP block resources on modern FPGAs while managing the required data buffering and architectural pipeline for 2-D filtering, and being scalable to large filter and frame sizes. Optimisation around the FPGA architecture enables the proposed filters to achieve higher operating frequency and, as a result, higher throughput compared to published previous work. The proposed architectures achieve high throughput by operating at near the DSP block theoretical maximum frequency, while also maintaining an adaptable convolution architecture with coefficients that can be updated at runtime. Moreover, this chapter demonstrates how the baseline architecture scales to three widely used video resolutions: HD (1280×720), full HD (1920×1080) and 4K (3840×2160), on a range of filter sizes that span from 5×5 to 25×25. Lastly, the proposed design is compared with equivalent filters generated using High Level Synthesis (HLS) and with previous work found in the literature.

The work in this chapter extends some initial exploration of these concepts on small filters that was discussed in [99] and has been published in:

- L. Ioannou, A. Al-Dujaili, and S. A. Fahmy. High Throughput Spatial Convolution Filters on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(6):1392–1402, 2020 [4].

4.2 Related Work

High performance image-processing on FPGAs has been an active field of research, mainly due to the ability of FPGAs to exploit fine and coarse grained parallelism, allowing for tradeoffs between performance and area [27]. The reconfigurability of FPGAs also means that they can provide the flexibility often desired in vision systems. Their high throughput processing, ability to exploit parallelism, and flexibility have led to the wide use of FPGAs in real time vision systems [28–30] and to implement a variety of filter structures [31, 32].

A typical vision processing flow moves from pixel-level operations to more abstracted algorithms on less dense and structured data, where software implementations can be a better fit due to ease of programming and irregular data access patterns. Ideally a real time vision system would therefore couple the high performance of a hardware accelerator, to take advantage of massive parallelism in low level operations, with the programmability of a processor for higher level operations. FPGA SoCs, like the Xilinx Zynq, couple an embedded processor with flexible reconfigurable fabric on the same silicon, with high throughput connectivity between them. FPGA platforms with PCIe connectivity can also be used within a workstation environment alongside a more capable CPU. The reconfigurability of FPGAs, including partial reconfiguration, also allows them to support dynamic vision systems where the hardware can adapt at runtime to changing conditions [95]. Hence, FPGAs are ideal for implementing the full computer vision stack including higher level software and low-level hardware in a broad range of domains, from distributed embedded computing to high performance servers. Within this context, generalised convolution architectures are explored to maximise the throughput of low-level operations within a typical vision flow for high bandwidth video streams.

Convolution, or 2-D spatial filtering, is computed by initially performing a pixelwise multiplication of each pixel within a window with a corresponding coefficient, followed by a function that aggregates these products to produce a single output [100]. Both of these functions may vary for different filter applications. The coefficients used in the pixelwise multiplication define the filter's operation, which can be, for example, noise removal, image sharpening, blurring/smoothing, or feature extraction. With real time vision systems typically deployed on streaming images, input data flow becomes simpler, not requiring storage of complete frames. It is, however, important that the computations within the convolution meet the real time constraints to avoid becoming a bottleneck.

Increasing image resolution in mainstream use means that the dimensions of convolution windows must also scale in order to maintain their effectiveness. For example, a 3×3 filter applied to a 1280 × 720 image is equivalent to a 9 × 9 filter for a 3840 × 2160 image, assuming that spatial equivalence is required. This, consequently, results in increased workload on more input pixels that in turn requires more demanding processing in order to maintain the same frame rate.

Most previous work on FPGA-based spatial filters has focused on optimisations based on the use of fixed coefficients or coefficients constrained to a specific range [35]. Such optimisations are most effective for filters in which the coefficients consist of zeros or ones, or other powers of two, since each multiplication can be replaced with a shift, resulting in no use of multipliers [37], and much improved hardware area efficiency. This comes at the cost of flexibility as those systems are fixed to a single purpose. Fixed filter implementations, however,

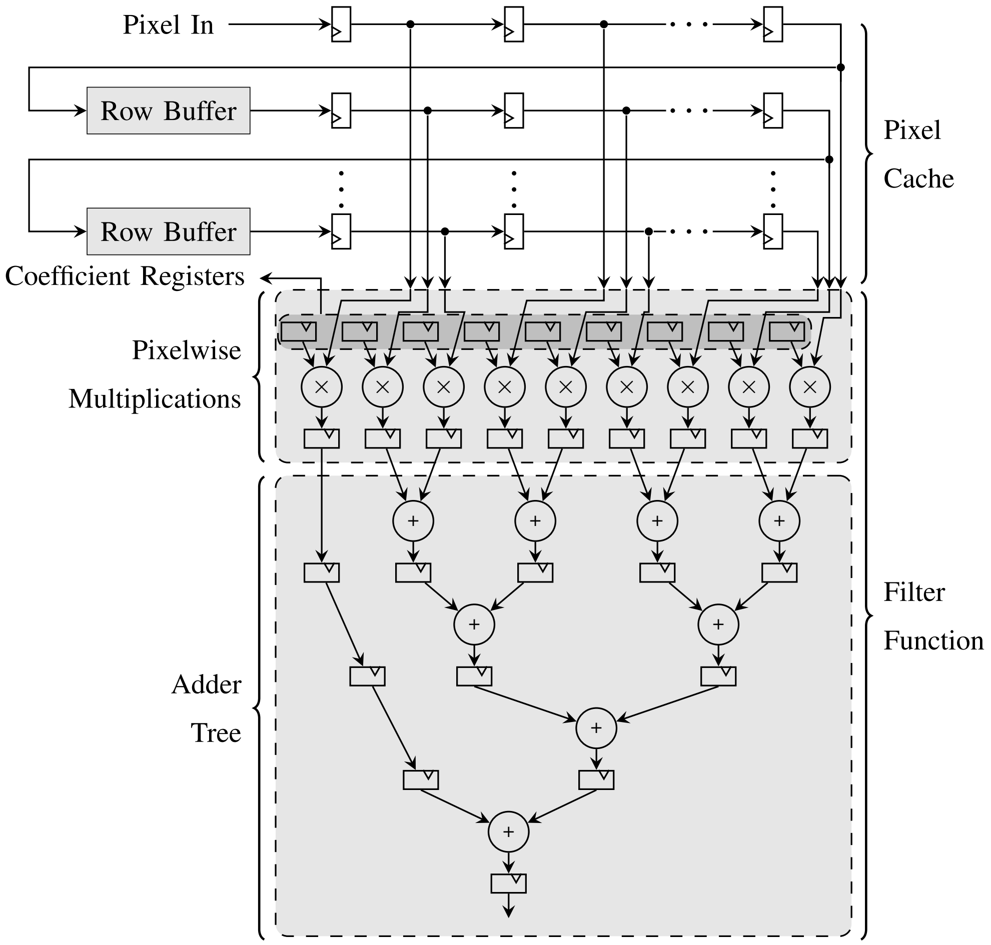


Figure 4.1: Filter architecture diagram, showing the various functional blocks.

are not ideal for smart vision systems, in which filters at the lower layers should be flexible enough to dynamically adapt to different requirements. Flexible convolution architectures use generic multipliers while providing external access to the coefficient registers to support dynamic adaptability.

4.3 Generic Filter Architecture

A typical filter architecture and its functional blocks are depicted in Figure 4.1. It operates in streaming mode, receiving a new pixel from the source image in each clock cycle, in raster scan order. To compute an output pixel, all pixels within the corresponding input window must be available. For $w \times w$ filter sizes, where w is an odd number, pixels from w rows are required to compute each output pixel. This requires a row buffer with the ability to store $w - 1$ rows plus w pixels (since only w pixels are needed from the last row). Full frame buffering is, therefore, not needed for streaming images, something that would consume significant area for large frame sizes, and possibly require frequent off-chip memory accesses, which can have a significant performance and power consumption overhead.

The pixels in the $w \times w$ filter window from the input image, formed around the calculated pixel, are stored in the pixel cache block as shown in Figure 4.1. The pixel values are then fed to the pixelwise multiplications block within which they are multiplied in parallel with the corresponding coefficients, which are stored in registers and can be configured at runtime. This enables modification from the higher layers of a complete vision stack as described previously. The filter's operating mode is controlled by a state machine that cycles between idle, coefficient update, priming, processing, and flushing modes. Output data are streamed at the same rate as the input pixels, maintaining simple data movement with no need for storing complete frames.

Numerous approaches toward more efficient filter designs have been proposed in the literature. These mainly target the core underlying multiply-accumulate (MAC) operations and are generally divided into two categories, multiplier-based and multiplierless filters. Multiplier-based filters directly map the multiplication to hardware multipliers. Park et al. [25] proposed a sharing scheme targeting vector–scalar multiplications through decomposing FIR filters. Bougas et al. [24] use internal pipelining in multiplier arrays to fold FIR filters. Ma et al. [38] reduce the computational complexity of 2-D convolutions by splitting large filter windows to a sequence of convolutions with smaller window sizes.

Multiplierless filters avoid the use of multipliers through various arithmetic transformations and representations. These include programmable canonic signed-digit (CSD) representation in [101], and distributed arithmetic (DA) in [33] and [34] that replaces multiplications with lookup table memories and adders. The Bachet weight decomposition theorem is used in [35] to similarly replace multipliers with ROMs and adders. Other multiplierless methods tailor their architecture to the filter function with hardwired shifts [36] or make use of powers-of-two weights for multiplierless CNN inference [37].

4.3.1 Boundary Handling

While the convolution window scans the input image, its computation becomes more complex when it targets pixels at the edges. This calls for particular handling or padding as the convolution window requires pixels that do not exist, as shown in Figure 4.2. An alternative is to restrict the sliding window within the valid region of the input frame, which results in an output frame of reduced size compared to the input frame. A scalable architecture should be able to support the addition of border management schemes without a significant impact on performance.

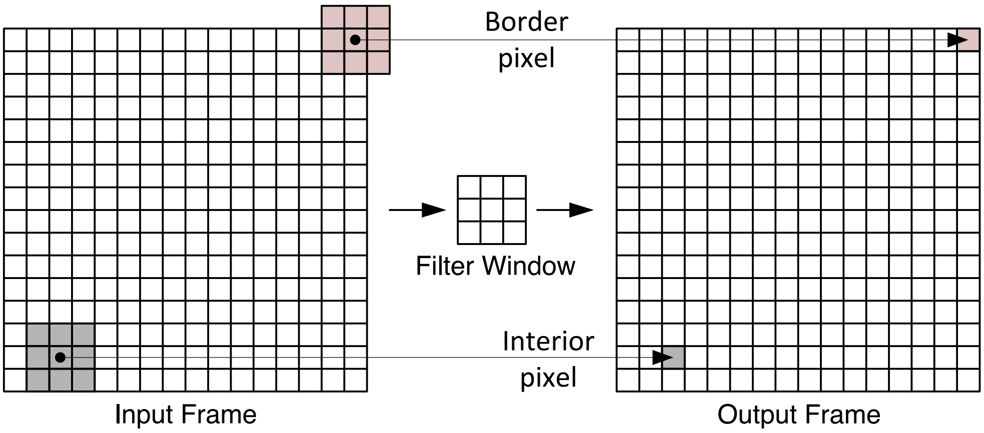


Figure 4.2: 2-D filter operation showing indicative examples for interior and border pixels.

4.4 FPGA DSP Block Architecture

FPGA DSP blocks have evolved significantly since their introduction into FPGA architectures, from simple multiplier blocks to fixed MAC capability, to programmable multiplication and arithmetic/logic in modern devices. While DSP block availability was limited in older FPGA platforms, even low end FPGAs today include sufficient DSP blocks to implement large filters. DSP blocks have also evolved to support wider input lengths, additional input ports, incorporation of a pre-adder, pattern detection capability and SIMD support. For instance, the DSP48E1 block on Xilinx 7 Series FPGAs supports multiplication, MAC, multiply-add, add-MAC, and three-input-add functions, among others. Interconnect between DSP blocks has also been improved, with modern FPGAs now offering dedicated cascade interconnect between DSP blocks that allows wider computations and chaining of DSP blocks to form 1-D FIR filters without using the logic fabric, hence achieving higher performance. Although connectivity enhancements benefit 1-D FIR filter implementations, 2-D structures cannot fully exploit these features, so data movement and buffering must be done manually in the logic fabric. Dynamic programmability is another feature of the DSP48E1, where it is possible to adjust ALU function ($ALUMODE$), operation mode ($OPMODE$) and input selection ($INMODE$) dynamically at runtime. This has led to their use in lightweight soft processors [102] and flexible overlay architectures [80, 103] where their functionality can be modified on a cycle-by-cycle basis or to reconfigure functional units, all while achieving high throughput.

However, to exploit these features and achieve high performance, designs must be optimized at a low level as synthesis tools are often unable to infer the best structures for complex designs [104]. Indeed, since DSP blocks can be clocked significantly higher than the typically achievable frequency in complex designs, it is possible to share them in a time-multiplexed manner through multipumping, where they are clocked at a multiple of the surrounding logic and multiple operations are mapped to a single DSP block per cycle with suitable buffering [105].

4.5 Filter Architecture

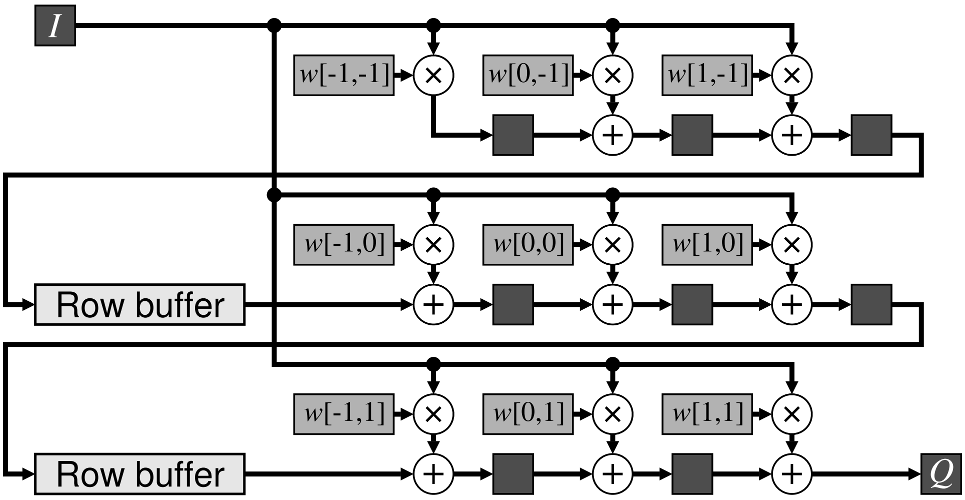


Figure 4.3: Transposed filter diagram, showing the various compute blocks and pipeline stages [98].

The proposed filter architectures reflect the general architecture described in Section 4.3, with a variety of architectural optimizations to achieve high throughput. The details of the filter design are discussed in this section.

The general structure of a filter, comprising pixel cache, coefficient multiplication, and adder tree was shown in Figure 4.1. Filters can also be implemented in transposed form, shown in Figure 4.3, in which the incoming sample is multiplied by all coefficients and products are summed serially through the delay line. Transposed form architectures have the advantage of being pipelined by default [100] compared to the manual pipelining required in direct form. FPGA DSP blocks have the required functionality and connectivity to enable 1-D transposed form filters to be implemented using only DSP blocks with no external logic. And hence, these designs have reduced resource utilisation and improved performance. For 2-D filters, however, data is buffered across multiple rows and as a result buffering is more complex and cannot be implemented directly using DSP blocks. Direct form architectures require a separate adder tree, which consumes additional resources and power. The adder tree depth depends on the filter size and scales by log_2 of the filter size w. Although the proposed architecture is in direct form, a comparison is made against a transposed form implementation for completeness.

4.5.1 Filter Function

The utilisation of DSP blocks in the proposed architecture has been made through direct instantiation. Although FPGA vendor tools are able to infer the use of DSP blocks from RTL code, their efficiency decreases for more complex structures as this automated inference does not fully exploit all DSP block features or always suitably pipeline them [105]. Through direct instantiation, the low-level DSP block mapping is controlled to ensure high throughput.

4.5.2 Pixel Cache

The filter cache consists of row buffers and individual registers for the pixels in the active window, which are connected to the coefficient multipliers. The number of row buffer units and individual registers depends on the filter size while the length of the row buffers depends on frame width. Hence, higher frame resolution is more demanding in terms of buffering, utilising more memory elements. Shift register look-up-tables (SRLs) are one way of more efficiently implementing line buffers on FPGAs, since pixels in the line buffers do not need to be accessed until they reach the filter window. By utilising a suitable coding style, it is possible to ensure the line buffers are implemented using SRLs [106]. To demonstrate the impact of this optimization, a six-row buffer (for a 7×7 filter size and 1280 wide frame) utilises over 61 000 flip-flops or only 110 flip-flops and 1920 LUTs when implemented using SRLs. The savings introduced through use of SRLs contributes to the scalability of the proposed architecture. SRLs do impact achievable frequency, lowering it from 700 to 600 MHz for this isolated experiment, but this is in line with the capabilities of the DSP block and so not a limiting factor in the proposed architecture, while offering a significant area saving. Moreover, the higher resource utilisation of the register based implementation can have an adverse impact on the placement and routing process in a larger design, resulting in longer routing delays for other parts of a design [106].

4.5.3 Adder Tree

In a direct form filter implementation, DSP blocks configured as multipliers, as shown in Figure 4.4, are used to calculate the pixelwise products while a separate adder tree follows to sum these products up and generate an output pixel value. Three different types of adder trees are explored, as shown in Figure 4.5. More specifically, the three layouts are as follows.

1. *DSP Layout:* The adder tree comprises directly instantiated DSP blocks configured as wide adders. Since this operation is mapped directly to

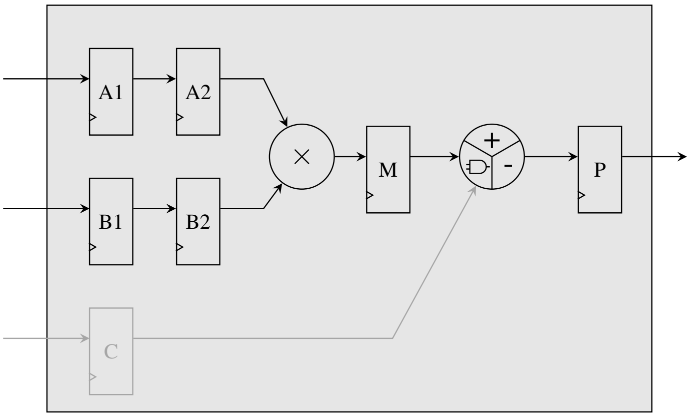


Figure 4.4: DSP48E1 block diagram for multiplication.

	Architecture		
	DSP	**LOG**	**DSPCOMP**
Number of Inputs	2	2	6
Basic Units	1 DSP48E1	LUTs	2 DSP48E1s LUTs
Latency	3	1	10
Number of adders for $w = 7$	48/36*	48	10
Number of stages for $w = 7$	5	5	3

*without/with SIMD mode

Table 4.1: Adder tree layout resource consumption.

silicon, it is used as a baseline for maximum performance. In this case only the post-adder in each DSP block is used.

2. *LOG Layout:* The adder tree is mapped to the FPGA logic fabric. This results in a more balanced utilisation of the device resources. Each adder in this layout is followed by a register, resulting in a pipelined architecture that is mapped to LUTs and registers.

3. *DSPCOMP Layout:* A compression component, mapped to the FPGA logic fabric, is used to reduce the depth of the adder tree while also using fewer DSP blocks. More specifically, the logic-based compressor (6:3) takes six operands and generates three partial sums, which are then summed using two DSP blocks.

Using hardened DSP blocks for the multiplier means that wordlength can be chosen to use the maximum available within the structure without impacting area significantly. The filter architecture uses 14 fractional bits for the coefficients. For the pixelwise multiplications, the input pixels are mapped to the 25-bit inputs of the DSP48E1 blocks while the coefficients are mapped to

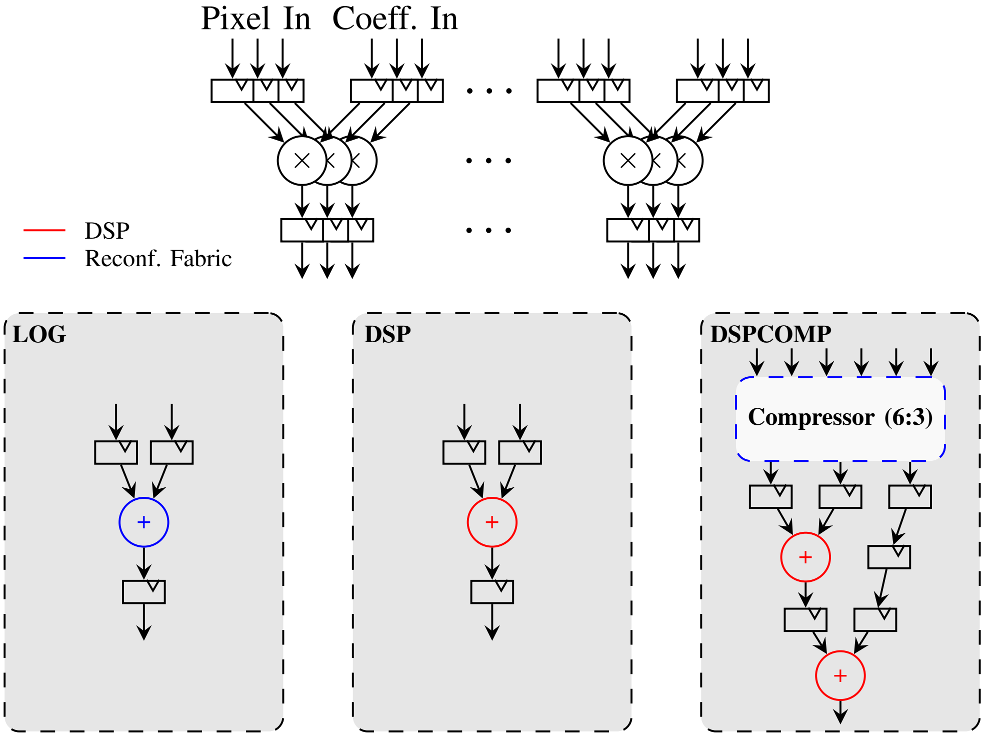


Figure 4.5: Alternative adder tree layouts: LOG, DSP, and DSPCOMP.

their 18-bit inputs. Since each pixel wordlength is fixed to 8 bits, the remaining bits in each 25-bit input are set to zero. Since these must be multiplied by the correct matching coefficients, it is not possible to reuse these additional input bits, but the loss in terms of hardware is minimal since the DSP block is not fracturable. In every case, the output of each multiplication is reduced from 48 to 24 bits and then fed to the adder tree. In the first stage of LOG and DSP-based adder trees, 24-bit additions are performed, while in the following stages, 48-bit additions are performed. The reduced input wordlength at the first stage is taken advantage of by adjusting the adders' wordlength in the LOG filter, and by mapping two 24-bit additions to a single DSP block in the DSP filter, reducing the number of DSP blocks utilised. In the DSPCOMP, 45-bit additions are performed throughout the adder tree due to the fixed wordlengths. The 24-bit inputs are sign extended accordingly before being fed to the DSPCOMP adder tree.

Information about the DSP block utilisation and latency for each adder tree layout is summarised in Table 4.1, and dataflow is shown in Figure 4.5. To achieve high throughput, each adder tree is extensively pipelined. The adders implemented in DSP48E1 blocks have a latency of three clock cycles, as shown in Figure 4.6, the compression logic requires two additional clock cycles to generate its partial sums while adders mapped on the FPGA fabric have a latency of a single clock cycle.

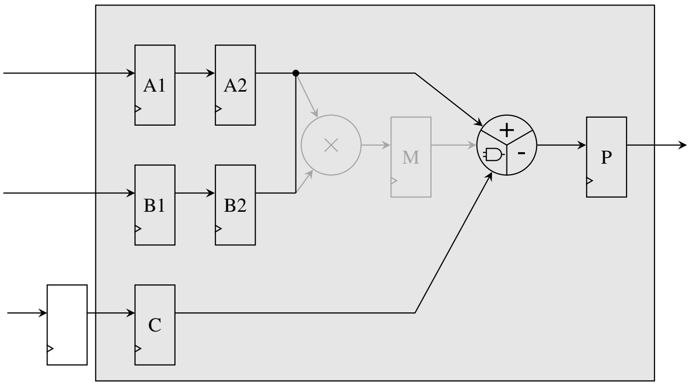


Figure 4.6: DSP48E1 block diagram for addition.

Transposed form filters are formed by reversing the signal flow while rearranging the building blocks of direct form filters accordingly. This results in an inherently pipelined adder tree [100]. Transposed form filters in 1-D are fully supported by the DSP blocks, since an MAC operation can be mapped to a single block, and therefore requires no external logic. Although in 2-D structures some logic is required, this does not fully nullify the savings introduced.

The estimates of DSP block utilisation in a transposed form filter function for a $w \times w$ convolution window are outlined in Table4.2. The DSP block utilisation shows those used in the individual pixel multiplications and the adder tree separately. In the direct form design with DSP adder tree, the dual 24-bit SIMD mode (two two-input adders) is used at the first stage of the adder tree to pack two additions in a single DSP block, as described earlier. Although the 25-bit pre-adders in DSP blocks usually offer a more efficient utilisation of the DSP resources, they can be used only in the first stage of the adder tree where 24-bit additions take place, so mapping two 24-bit additions to the 48-bit post-adder was preferred. This leads to the same DSP block utilisation, while maintaining a more straightforward interconnect with no need for delay buffers. In the direct form design with LOG adder tree, the adder tree is mapped to the FPGA fabric, using no DSP blocks. The compressor (6:3) in the direct form design with DSPCOMP adder tree generates three partial results that are summed up using two DSP blocks, at the expense of some logic utilisation, compared to the five DSP blocks required in the direct DSP. The proposed architecture does not include any mechanism to handle overflow and as a result inaccuracies may occur in cases where the intermediate results overflow.

4.6 Border Management Techniques

Border management techniques handle the undefined regions of an input pixel stream to produce an output of the same size. Spatial filters can be

	DSP Block Usage		**DSP blocks for**
	Mult. Block	**Adder Tree**	$w = 7$
Direct			
DSP	w^2	$\frac{w^2-1}{4} + \frac{w^2-1}{2}$	85
LOG	w^2	-	49
DSPCOMP	w^2	$\lceil 2\times \frac{w^2-1}{5} \rceil$	69
Transposed	w^2		49

Table 4.2: DSP Block usage for different configurations for a filter size of $w \times w$.

implemented without border management, generating output images with reduced size. In 1-D filters, this affects only the very beginning of the input signal, and the first outputs can be ignored. In two dimensions, however, this affects every output frame, reducing frame size. Although this may not be an issue for all applications, there are occasions where this can be problematic, such as when a sequence of filters is used to process an image. In CNNs for machine learning, border management is not usually required, since as data propagate through the neural network, the convolutions are applied to more abstracted features and these edge pixels have little or no impact. Filters with no border management have simpler control logic and data flow, allowing a straightforward implementation of a transposed form filter.

The complexity introduced by border management has resulted in a body of work on mitigating its effects. A review of 2-D border handling methods on FPGAs is presented in [107], where the authors also introduce a novel border handling management scheme with overlapped priming and flushing. In this method, registers acting as temporary pixel buffers and multiplexers are used to reduce the time overhead in handling border pixels. Bailey and Ambikumar [100] proposed two novel border handling mechanisms, transformation coalescing, and combination chain modification, that reduce the complexity of border handling in transposed form filters while taking advantage of the inherent pipelining of the transposed form structure. Another approach in [108] considers symmetric extension for 1-D signal border management by exploiting the SRL16 shift register primitives in Xilinx FPGAs to skew data. This technique, however, is not ideal for DSP block based filters as it introduces shift registers between the multiplication and addition, preventing efficient mapping to DSP blocks.

The most widely used techniques for handling border pixels are *border neglecting, wrapping*, *function change, constant extension*, *border duplication*, and *mirroring with* and *without duplication*. *Mirroring*, for instance, is used in [107–109], while detailed description of all these methods can be found in [98]. *Border neglecting* requires no additional logic but results in a reduced output frame which can be troublesome in small resolutions or cascading filters. All other methods generate an output an image of the same resolution as the input one, however the *wrapping* method benefits from its small control logic at the cost of possible discontinuities and artefacts. *Function change* method is difficult to generalise to all filters while requires complex control logic. *Constant extension* and *border duplication* suffer from discontinuities, artefacts and additional control logic. Lastly, the *mirroring* technique's only drawback is the additional control logic. Figure 4.7 shows how *constant extension*, *border duplication* and *mirroring* techniques behave on an image.

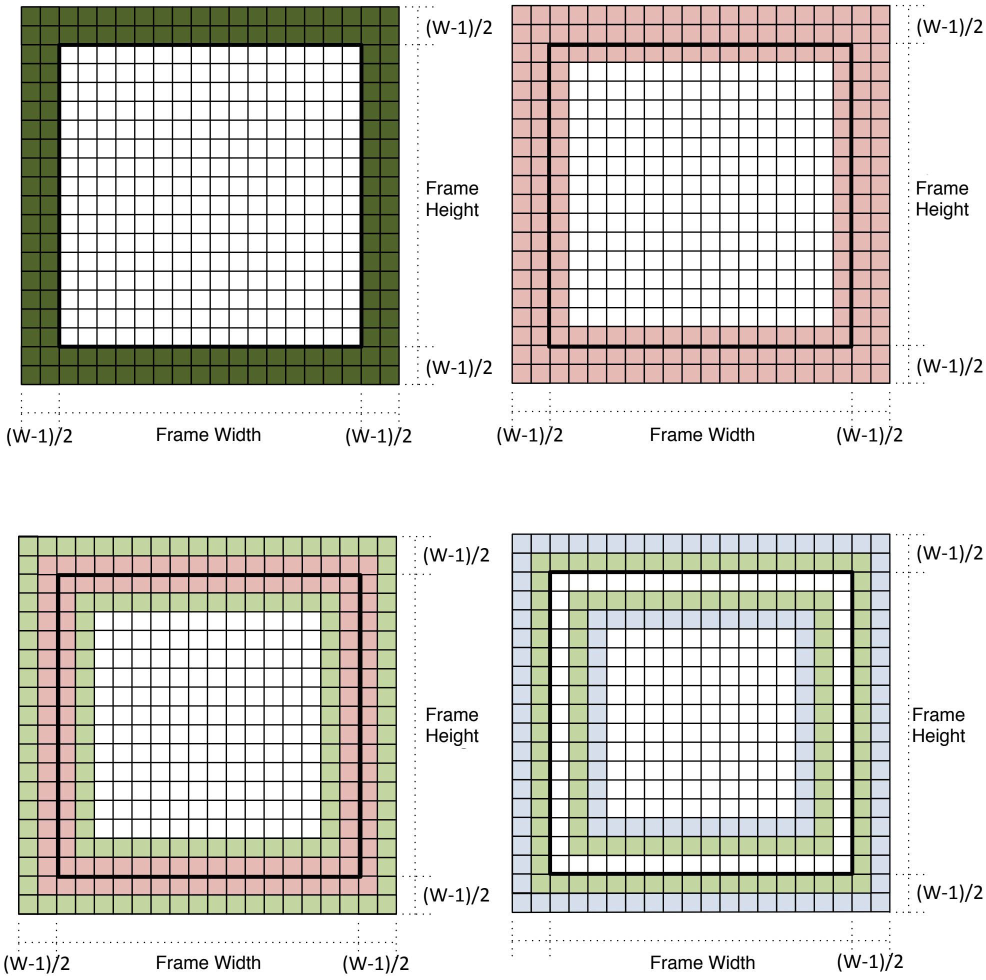


Figure 4.7: Border management techniques (top left: constant extension, top right: border extension, bottom left: mirroring with duplication, bottom right: mirroring without duplication) [99].

These techniques can be implemented in hardware in a number of ways, as described in detail in [107]. *Direct window input* and *cached priming* add extra stalling cycles when processing border pixels, reducing their efficiency. This complicates the streaming data flow and, as a result, the datapath control logic in real time systems. In addition, *direct window input* uses a complex address generation logic which adds to its control logic complexity. In contrast, *cached priming* uses a less complex address generation logic, however, this method requires extra multipliers for its operation.

The *overlapped priming and flushing* scheme on the other hand, both naive and the scheme proposed in [107], preserve the regular streaming flow at the cost of additional logic. More specifically, additional multiplexers are required to make the replacement values immediately available. The naive scheme uses extra row buffers along with the additional temporary registers within the window pixel cache, requiring additional memory components. All these techniques are modifications to the pixel cache block in the filter architecture.

Although border handling techniques and their optimisation are out of the scope of this thesis, this section concludes that the proposed architecture can be extended to include these functionalities without significant overhead on performance or efficiency. This is demonstrated by an indicative implementation that includes *overlapped priming and flushing* scheme, proposed in [107], in Section 4.7.3.

4.7 Proposed Architecture Results

This section presents the implementation results of the proposed filter architecture, showing operating frequency and throughput as well as area and latency (which represents the number of clock cycles required for the first output pixel to be generated), for different design parameters. The proposed architecture is initially investigated in detail using an indicative filter with a 7 × 7 window for 1280 × 720 frames, which is used to make comparisons between the adder tree types in direct form, the direct and transposed forms, against an HLS equivalent, and the impact of border management on area and performance. Subsequently, the architecture is shown how it scales on three frame sizes, 1280 × 720, 1920 × 1080 and 3840 × 2160, for 11 different filter sizes, ranging from 5 × 5 to 25 × 25. Finally, the proposed architecture is compared against published work in the literature. All the proposed designs were implemented in Verilog HDL, using Vivado 2018.2, targeting the Xilinx Virtex 7 XC7VX690 on the VC709 development board and the results presented in Sections 4.7.1– 4.7.6 are post place and route.

4.7.1 Adder Tree Designs in Direct Filter Structure

The effect of three different adder trees is explored, **DSP**, **LOG**, and **DSP-COMP**, as described in Section 4.5 without considering border management.

Adder Tree **Design**	**Freq.** (MHz)	**Latency** (Cycles)
DSP	535	7713
LOG	525	7700
DSPCOMP	530	7725

Table 4.3: Frequency and latency of direct form filter implementations with different adder tree designs for 1280×720 frame, 7×7 filter and no border management.

Table 4.3 summarises the operating frequency and latency for the three designs, all offering high throughput at similar frequencies. The LOG filter is slightly slower with marginally improved latency.

Table 4.4 shows the resource utilisation for all designs. All filters use 49 DSPs for the pixelwise multiplication, as shown earlier in Figure 4.1. The LOG design does not use any DSPs in the adder tree, while the DSP design uses 36 and the DSPCOMP design 20 DSP blocks when $w = 7$. These results correspond with the estimates in Table 4.2. When comparing the total resource utilisation, the DSPCOMP utilises the most registers while ranking second for LUT and DSP utilisation. Since all filters operate at almost the same frequency, DSPCOMP can be considered the least efficient when considering utilised area. The LOG filter utilises about 73% more LUTs compared to DSP while using 42% fewer DSP blocks and approximately the same number of registers. Considering the availability of such resources in modern FPGAs, DSP blocks are the least abundant, while the register-to-LUT ratio is 2 to 1. Therefore, the resource mix of the LOG filter better mirrors the FPGA architecture and utilises the fewest DSP blocks. Trading 1456 LUTs and 23 registers for 36 DSP blocks is a net positive in area terms based on the approximate 120:1 ratio of resources on the device. This configuration allows better replication of parallel filters for different streams while not utilising more DSP blocks than are necessary, and achieving almost identical performance.

4.7.2 Direct Versus Transposed Form Architectures

Table 4.5 summarises the resource utilisation, maximum operating frequency, and latency of the direct form (with LOG adder tree) and transposed form architectures. The transposed form filter structure combines the pixel cache and filter function into a single module so separate results are not shown. In terms of performance, both filters have similar latency while the direct form operates at a slightly higher frequency. Direct form uses significantly more registers than the transposed form while using about half as many LUTs. The majority of LUTs in the transposed form filter are utilised by the combined

Modules	**Adder Tree**	**Resource**		
		Regs	**LUTs**	**DSPs**
Coeff. File	DSP	735	0	–
	LOG	735	0	–
	DSPCOMP	735	0	–
Control Unit	DSP	49	59	–
	LOG	49	63	–
	DSPCOMP	49	59	–
Pixel Cache	DSP	440	1920	–
	LOG	440	1920	–
	DSPCOMP	104	1920	–
Filter Func.	DSP	4516	12	85
	LOG	4539	1464	49
	DSPCOMP	8954	1120	69
Total	DSP	5768	1991	85
	LOG	5791	3447	49
	DSPCOMP	9870	3099	69

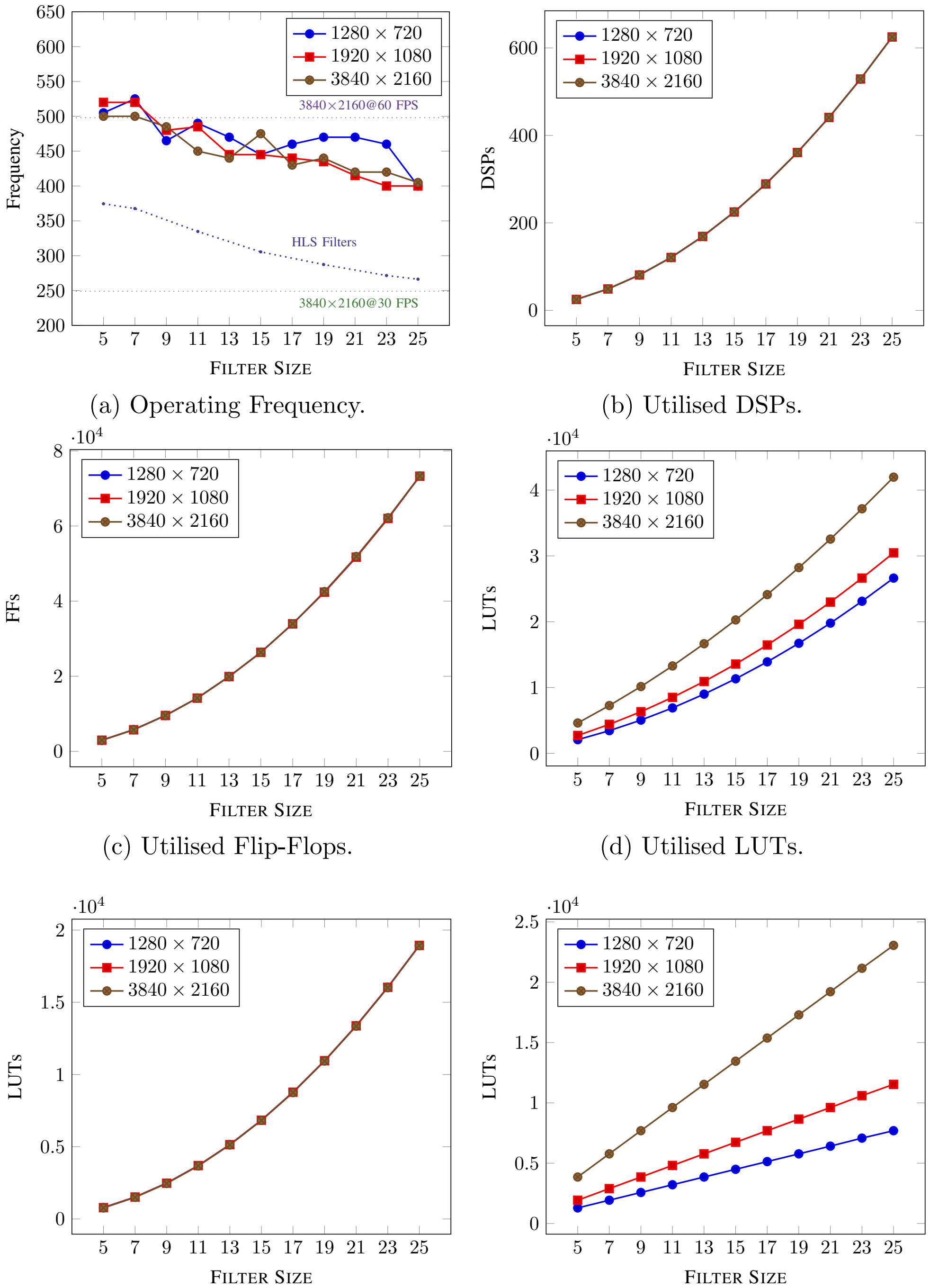
Table 4.4: Resource utilisation of direct form filter implementations with different adder tree designs for 1280×720 frame, 7×7 filter and no border management.

Module	Direct LOG			Transposed		
	Regs	**LUTs**	**DSPs**	**Regs**	**LUTs**	**DSPs**
Coef. File	735	0	-	735	0	-
Control Unit	49	63	-	83	42	-
Pixel Cache	440	1920	-	918	7200	49
Filter Func.	4539	1464	49			
Total	5791	3447	49	1763	7242	49
Freq. (MHz)		525			505	
Latency (Cyc)		7700			7691	

Table 4.5: Direct and transposed form implementation summary with 1280×720 frame and 7×7 filter.

filter function and pixel cache modules. The LUTs in this case are solely used as shift registers (SRLs) for buffering, affected primarily by image width. The transposed form architecture does not require an adder tree, instead using an adder chain that can be packed into the same DSP blocks that implement the individual multipliers. As a result, the DSP utilisation of both filters is the same. Additional logic is still required, however, for the 2-D transposed form filter as the dedicated cascade wires offered by the DSP blocks are only suitable for 1-D structures. The direct form LOG design implements the adder tree in the FPGA logic fabric.

While both filters have similar performance, with the direct form operating at slightly higher frequency, their resource utilisation varies significantly. The resource utilisation of LOG has a better register to LUT ratio as discussed previously. The transposed form filter architecture is also less extensible to support border handling as pixel values within the window are already accumulated with other pixels. This issue has been discussed in [108], where the use of shift registers is proposed as a solution. This approach however separates the multiplication and addition, making the resource utilisation similar to the direct form. Two novel border handling techniques are proposed in [100] to reduce the border handling complexity in transposed form filters. These methods result in designs of similar complexity to the direct form ones. Hence, transposed form filters must sacrifice their efficiency to offer scalability and support for border handling. The work in this subsection shows that direct form filters can be suitably pipelined to achieve equivalent performance. Finally, as discussed earlier, the resource utilisation mix of the direct form filter better mirrors the resource availability on modern FPGAs.



(a) Operating Frequency.

(b) Utilised DSPs.

(c) Utilised Flip-Flops.

(d) Utilised LUTs.

(e) Utilised LUTs as Logic.

(f) Utilised LUTs as shift registers.

Figure 4.8: Implementation results of the proposed filter architecture on three image resolutions, each on 11 filter sizes.

4.7.3 Direct Filter Structure With Border Management

Although border management is not the focus of this chapter, the use of the overlapped priming and flushing scheme is explored, as presented in [107]. As some applications may require the use of border management, this section demonstrates the extensibility of the proposed architecture to support this

Modules	**Direct LOG**		
	Regs	**LUTs**	**DSPs**
Coef. File	735	0	–
Control Unit	63	65	–
Pixel Cache	652	1657	–
Filter Func.-LOG	4542	1464	49
Total	6020	3186	49
Freq. (MHz)		515	
Latency (Cycles)		3860	

Table 4.6: Direct LOG architecture for 1280 × 720 frame and 7 × 7 filter with border policy from [107].

feature. Moreover, the overhead introduced, as a result of the increased design complexity, is quantified. Table 4.6 summarises the implementation results of the direct form LOG filter with border management on a 7 × 7 filter for a 1280 × 720 frame size.

Compared to the direct form LOG design without border management, the border extension architecture uses fewer LUTs and more registers, while DSP block utilisation remains the same. Of particular interest is the LUT reduction in the Pixel Cache module within the border management architecture. Its LUT utilisation amounts to 376 LUTs for logic and 1281 LUTs for SRLs, meanwhile the same module without border extension uses all 1920 LUTs as SRLs. The straightforward flow of data without border management enables the synthesis tool to map the Pixel Cache into SRLs. In contrast, border management requires additional logic for its implementation and its more complex data flow is mapped to both SRLs and registers, contributing to higher utilisation of slice registers. Border management reduces frequency marginally due to the increased complexity of the design and increased routing congestion. The latency of the first output pixel is decreased, as expected, since required pixels are present in about half as many clock cycles as in the baseline design. For example, when a 7×7 filter is used, only four of the first seven lines need to be buffered for the computation to start, since the out of frame pixels in the window are replicated from those pixels. Without border management, no output pixel is produced until seven lines are buffered.

4.7.4 Comparison With Vivado HLS Filters

HLS is increasingly gaining popularity due to its higher level design abstraction compared to HDL, enabling faster design time and functional verification of hardware accelerators. Ease of implementation in HLS, however, can come at the cost of reduced performance and possibly poorer resource efficiency, especially when considering processing patterns that map well to low-level architectural features like the DSP blocks. Vivado HLS 2018.2 was used, with the image processing libraries provided by Xilinx, to explore how resource utilisation and throughput scales for filters generated from high level code, assuming a 1280×720 frame size. Pragmas were also used to unroll and pipeline the computation of the HLS filters in order to enable streaming processing, reading, and outputting a pixel in each clock cycle. The achieved frequency for these designs is plotted with a dotted line in Figure 4.8a. All filter coefficients are configurable, resulting in the same functionality and DSP block utilisation as the proposed filter architecture. Coefficient wordlength was set to 18 bits, and inputs and outputs were set to 8 bits with 16-bit intermediate results. The reduced wordlength compared to the proposed architecture as described in Section 4.5, reduces area somewhat, but allows the tool to generate high throughput filters for more competitive comparison. Table 4.7 summarises the relative change in resource utilisation and frequency for the HLS filters compared to the proposed architecture, for the same parameters, compared to the proposed filter architecture as the baseline. The reduced wordlengths result in reduced resource utilisation compared to the proposed filter architectures. This becomes more apparent as the filter size increases, resulting in deeper and wider adder trees, which in turn increases the difference in overall resource utilisation. The purpose of this comparison, however, is primarily throughput and HLS filters have at best 25% lower throughput and up to 40% less for larger filters. This demonstrates the effectiveness of the proposed architecture for high throughput applications, where some area overhead can be tolerated.

	Filter Size ($w \times w$)						
	5	7	11	15	19	23	25
Regs	−36.83	−41.29	−59.71	−65.07	−66.04	−67.54	−68.20
LUTs	−15.79	−22.74	−22.72	−28.71	−34.19	−39.91	−42.86
DSPs	0	0	0	0	0	0	0
Freq. (MHz)	−25.84	−29.97	−31.68	−31.34	−38.86	−40.96	−33.42

Table 4.7: Relative resource utilisation and frequency for Vivado HLS filters.

4.7.5 Scalability Analysis

Spatial filters are widely used in configurations with different filter and frame sizes as required for a variety of vision applications. Therefore, performance and resource utilisation are explored while scaling the proposed architecture to three frame sizes, 1280 × 720, 1920 × 1080 and 3840 × 2160, with 11 filter sizes ranging from 5 × 5 to 25 × 25. Results are illustrated in Figure 4.8a– 4.8f. Operating frequency varies from 525 to 400 MHz, decreasing as the filter size increases, due to the critical path resulting from a wider adder tree and routing of more coefficient products. The frequency fluctuations are a result of the critical path moving through various parts of the adder tree as the architecture grows, leading to routing congestion. Meanwhile, frame size has minimal impact on operating frequency as it primarily affects the size of the line buffers, which are not in the critical path. These results compare favorably with the DSP theoretical maximum frequency of 650 MHz on this Xilinx Virtex 7 device [110].

DSP block and flip-flop utilisation are dependent on filter size rather than frame size. DSP blocks are explicitly instantiated for the multiplication of window pixels with the filter coefficients, and hence are filter size dependent. Flip-flops are mainly used for pipelining the computational datapath, which in turn depends on the filter size. LUT utilisation is more complex, depending on both the filter and frame size. To further analyse the scaling pattern, LUTs utilised as logic are shown in Figure 4.8e while the LUTs utilised as SRLs are shown in Figure 4.8f. LUTs as logic are primarily in the adder tree and additionally in the control logic. LUTs as shift registers are used primarily in the line buffers and as a result their utilisation depends primarily on the number of lines that need to be buffered and also on the width of those lines.

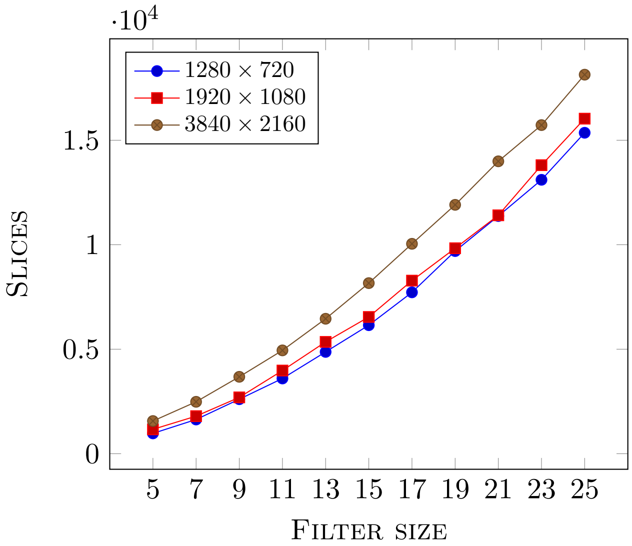


Figure 4.9: Slice utilisation for each filter implementation.

In order to investigate whether the individual resources in Figures 4.8b to 4.8f are efficiently used within each FPGA slice, the overall slice utilisation for each filter is graphically depicted in Figure 4.9. We observe that the overall slice utilisation patterns are similar to those of the individual resources. Therefore, it is expected that the overall slice utilisation does not hinder scaling of the proposed filter architecture due to inefficient individual resource utilisation.

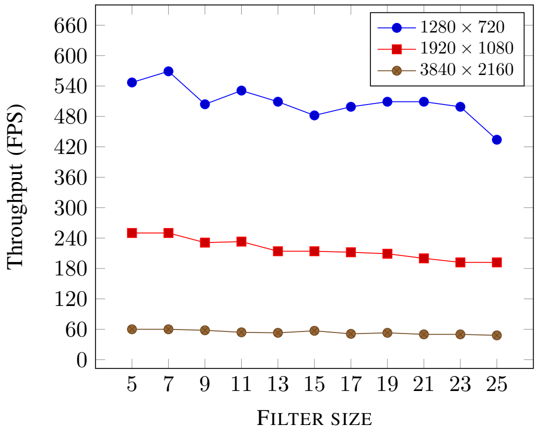


Figure 4.10: Achievable frame rates for varying filter and frame sizes.

Finally, Figure 4.10 shows how the operating frequency translates to throughput in FPS. All designs perform well over the 30 FPS required for real-time processing, even on 4K videos. 60 FPS is achieved by the majority of designs, except 9×9 and larger filters on 4K frames, achieving 58 FPS for the 9×9 filter, and as low as 48 FPS for the 25×25 filter. To determine whether newer FPGAs would allow processing at 60 FPS, the 25×25 filter for 4K frames was implemented on a Zynq Ultrascale+ ZCU102, successfully satisfying the 500-MHz constraint for 60 FPS. This suggests that the performance of this architecture scales well with newer FPGA devices.

4.7.6 Comparisons With Previous Work

Table 4.8 summarises previous relevant work in the literature. The FPS column in the same table has been extrapolated under the assumption that these architectures generate one pixel per cycle, in streaming processing flow, in cases where this attribute was not reported.

Licciardo et al. [35] implement a multiplierless design that emulates the IEEE-754 floating point standard through the use of fixed point adders and additional logic to manage exponent alignment. Their proposed filter architecture is tailored to a fixed set of coefficients and a fixed range of input values using the Bachet weight decomposition theorem. Ortega-Cisneros et al. [111] present a 3×3 filter with fixed coefficients obtaining at best 318-MHz frequency. The work in [34] presents a multiplierless, coefficient independent filter that

	Kernel Size	**FPGA Platform**	**Freq. (MHz)**	**Resolution**	**FPS**	**Fixed Kernel**	**Notes**
[100]	5×5	Cyclone V	175	1024×768	223		Transposed form, DSP-based, No border ext.
[100]	5×5	Cyclone V	152	1024×768	193		Direct form, Overlap prime and flush [107], Two-stage pipeline
[100]	5×5	Cyclone V	174	1024×768	221		Transposed form, Zero ext. using Transform Coalescing
[100]	5×5	Cyclone V	185	1024×768	235		Transposed form, Zero ext. using Combination Chain
[100]	5×5	Cyclone V	173	1024×768	219		Transposed form, Constant ext.
[100]	5×5	Cyclone V	159	1024×768	202		Transposed form, Duplication
[100]	5×5	Cyclone V	188	1024×768	239		Transposed form, Two-Phase duplication
[100]	5×5	Cyclone V	180	1024×768	229		Transposed form, Mirroring
[100]	5×5	Cyclone V	178	1024×768	226		Transposed form, Mirroring with duplication
[35]	3×3	XC7V	213	640×480	692	×	Multiplierless, Emulates IEEE-754, Optimized for fixed set of coeffs. with fixed input range
				1920×1080	102	×	
				3840×2160	25	×	
[34]	7×7	V4LX160	175	1920×1080	84		Multiplierless, Zero-padding, Flexible Coefficients
	11×11	V4LX160	181	1920×1080	87		
	22×22	V4LX160	177	1920×1080	85		
	30×30	V4LX160	171	1920×1080	82		
[34]	7×7	V4LX160	183	1920×1080	88		Use of Multipliers
	11×11	V4LX160	142	1920×1080	68		
	22×22	V4LX160	149	1920×1080	72		
[111]	3×3	Stratix V	318	1024×720	410	×	Fixed Kernel
[112]	5×5	V5LX330	115	-	-	-	Neural Network related
[113]	7×7	V4SX35	200	-	-	-	Neural Network related
[114]	13×13	V4LX25	50	-	-	-	-
[115]	4×4–25×25	Stratix III	≤115	1920×1080	≤55		Fixed point
[115]	4×4–13×13	Stratix III	≤114	1920×1080	≤55		Floating point

Table 4.8: Summarised previous work on 2-D spatial filters.

also utilises a mechanism for zero padding at the borders. They also compare against a baseline architecture that uses embedded multipliers and other work in the literature, with the most relevant to the work in this chapter in [112–114]. Bailey and Ambikumar [100] explored border handling in transposed form filters. More specifically, they explored the additional cost of border management, the cost of different border extension methods in transposed form filters and the scaling of their proposed optimal border extension mechanism. The scaling exploration takes place on a morphological filter at seven filter sizes, ranging from 3×3 to 15×15, for a frame size of 1024×768. They also provide implementation results for the overlapped priming and flushing method presented in [107] on a direct form filter, a method used in this chapter for comparison in Section 4.7.3. In this particular case, the authors use a two-stage pipeline on the combination tree and further improvements can be obtained with a more heavily pipelined architecture. Meanwhile, the proposed designs, fully pipeline the adder tree. The work in [115] explores the performance and energy consumption of FPGAs, GPUs, and multicore processors for sliding window applications. The authors in this case implement three applications, sum of absolute differences (SAD), 2-D convolution, and correntropy, on all platforms. They explore how their architectures scale on a range of filter sizes, each for three frame sizes: 640 × 480, 1280 × 720, and 1920 × 1080. For the FPGA analysis, an Altera Stratix III E260 on a GiDEL ProcStar III board was used. Their 2-D convolution architectures use 16-bit fixed point or 32-bit floating point representations, obtaining operating frequencies of 104–115 and 103–114 MHz respectively. That work concludes that FPGAs are more power efficient, compared to GPUs and multicores, while providing significant performance improvement for large input sizes.

The implementation results of the filter architectures in this chapter demonstrate significant throughput improvement compared to previous work in the literature, while being flexible to adapt to varying coefficients dynamically. It is also worth noting that the filter architectures used as references for comparisons in this chapter, including the transposed form filter and border management enabled design, also demonstrate substantial improvements compared to previous work.

4.8 Summary

This chapter presented a detailed discussion on 2-D spatial convolution filter design for FPGAs. It proposed a scalable direct form architecture that was shown to be extended to support border management, while it offered high throughput. The latter was achieved by the architectural optimisations driven by the underlying FPGA architecture, specifically the DSP blocks. Various adder tree designs have been compared, alongside a comparison against transposed form implementations and an extended design with border management, that used the overlapped priming and flushing scheme. The proposed architecture was scaled to a wide range of filter sizes, and frame sizes up to 4K. It was shown to offer throughput of over 60 FPS in most of these cases while it was also shown to achieve improved performance on more recent FPGA devices. The proposed designs were optimised around the features of modern FPGA DSP blocks, used through explicit instantiation, to achieve high throughput. Comparisons with HLS filter equivalents showed that the proposed architecture centric design was able to use the underlying FPGA resources more efficiently and obtained better performance compared to more generic methodologies. This has been achieved through the direct instantiation of primitive blocks which was enabled by the use of low level HDL language. In addition, extensive comparisons with previous work showed that all proposed filter designs offered significant throughput improvement while being flexible to adapt to different coefficients dynamically.

Chapter 5

Lightweight Streaming Neural Network Overlay using FPGA DSP Blocks

5.1 Introduction

Various NN topologies have demonstrated good performance in specific domains, for example, Convolutional Neural Networks are widely used in computer vision while Recurrent Neural Networks work well for time-series data. Hybrid NN structures are also used for more complex tasks, for example, an LSTM network, can be used after a CNN to generate captions for images [116]. Fully Connected, or dense layers, are often embedded in the last parts of these networks to implement the classification or regression task. Alternatively, NNs comprising only dense layers can be used for less complex tasks such as specific event detection [1]. As a result of the increasing efficacy of NNs, there is a growing interest not only in improving their accuracy, but to also accelerate this class of workloads for real time performance.

The inherent parallelism and computational regularity in NNs have been taken advantage of in highly parallel computing platforms, such as multicore CPUs and GPUs, and in custom computing architectures on FPGAs and ASICs. The ease of accelerating NNs in highly parallel computing platforms, through the availability of a number of frameworks, coupled with their fast compilation, have driven wider use of such platforms. Custom computing architectures offer additional advantages in terms of datapath and numerical representation optimisations, offering improved energy efficiency, which in turn makes them ideal for power-constrained platforms at the edge, where multicore CPUs and GPUs are unlikely to be suitable.

Most previous work on FPGAs has focused on accelerating the generic matrix-vector operations used for NN inference. Weight pruning and quantization have also been widely used to effectively reduce the memory requirements of models. High Level Synthesis (HLS) has contributed significantly in reducing accelerator design time, but still requires a lengthy backend toolflow. Other work has focused on bridging the gap between software programmable platforms and FPGAs by proposing automated toolflows [47]. The majority of these take advantage of the higher abstraction layer offered by HLS to also provide design space exploration, resulting in hardware implementations tailored to user requirements and platform capabilities. An example of NN acceleration vendor flow on FPGAs is Xilinx Deep Neural Network Development Kit (DNNDK). In the same context, Xilinx Vitis enables compilation of accelerators from higher level standard frameworks. Much of the published work targets more capable FPGAs on servers, with high bandwidth PCIe interconnect [16, 48, 117, 118].

Although research in the embedded domain has also flourished, many of these efforts either rely heavily on batch processing to generate high throughput, thus underperforming on single network inference and streaming data applications, or time multiplex complex compute units, thus not fully exploiting parallelism. Other optimisation methods include extreme quantization, even down to single bit data, and pruning. Although neural networks have been shown to tolerate such optimisations, these come at the cost of flexibility in the compute architecture while requiring additional design space and accuracy exploration, in addition to quantisation aware training. Finally, the majority of published work does not consider the FPGA architecture in detail, so fails to maximise achievable frequency [104]. This results in lower performance than what should be achievable and poorer energy efficiency since leakage power is clock independent [16]. In contrast, FPGA implementations that achieve high operating frequencies do so at the expense of flexibility and thus modifications to network topology or coefficients require a new compilation. Some work on large scale matrix multiplication on datacenter FPGAs has demonstrated near theoretical maximum performance, but relies on large FPGA fabrics to enable full unrolling of NN computations [16, 117].

Many end user applications rely on processing in both embedded and datacenter domains. For example, the widely used voice assistants, such as Amazon Alexa and Apple Siri [42, 119], process natural language both at the edge device and in the datacenter. The edge device is responsible for wakeword detection, e.g. “Hey Siri”, through the use of lightweight NNs, while the words that follow are processed in the cloud. At the edge, hybrid processing is usually employed to maximise efficiency which includes the use of a low-power, always-on processor along with the device’s main processor. A lighter network runs on the low-power processor and once this network generates a value that exceeds a threshold, the main microprocessor is woken up to run a more complex network. Depending on the device’s capabilities, these networks can indicatively be 5 layers deep, using fully connected layers with either 32, 128 or 192 neurons [42]. Hence, we envisage a growing need for edge devices to accommodate lightweight or moderate sized NNs, to support offloading of further processing to the cloud. This can be a result of the constrained resources of the edge device, or to protect the Intellectual Property of an organisation by adding a layer between the edge device and their trained Neural Network. Architectures for this purpose should be self-contained and flexible enough to support different NN structures dynamically. Most previous work has proposed co-processors that rely on a host processor to coordinate their operation and manage data transfers or only considered one layer type, offloading others to general purpose architectures.

As described in Section 2.6.2, overlays can enable high level programmability with rapid compilation and predictable performance. Architecture-centric overlays on FPGAs were shown to achieve high frequency while scaling to large overlay sizes [80]. In addition, compilation to the overlay does not involve the backend flow and is therefore fast, lightweight and vendor independent.

The work in this chapter has been published in:

- L. Ioannou and S. A. Fahmy. Neural Network Overlay Using FPGA DSP Blocks. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 252–253, 2019 [2].
- L. Ioannou and S. A. Fahmy. Lightweight Programmable DSP Block Overlay for Streaming Neural Network Acceleration. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT)*, pages 355–358, 2019 [3].

5.2 Related Work

Previous work on overlays, as described in Section 2.6.2, has shown that a coarser grained architecture on top of the finer FPGA fabric can reduce the long toolflow compilation time, enabling a more dynamic flexibility. Overlays' performance can be more predictable, as it is closely tied to the fixed performance of its functional units, while routing complexity can be reduced by tailoring them to the required data movement [80]. The work in [81] has shown that, through various optimisations, faster compilation runtime and reduced area utilisation can be obtained at the cost of lower frequency. Performance improvement has been obtained in [80], by using the high performance DSP blocks, that are abundant in modern FPGAs. More generic compute architectures on FPGAs, for example the Xilinx DPU supported by the DNNDK, provide a more balanced acceleration [86]. DPU can take advantage of various NN optimisations to provide better efficiency, that have generally shown many benefits at the cost of model accuracy [120]. Meanwhile, the analysis presented in [60] has shown that fully connected layer processing on a CPU is not very efficient due to cache misses that result in frequent off-chip memory transfers.

5.3 Serial and Fully Parallel Multiply Accumulate Operation Comparisons

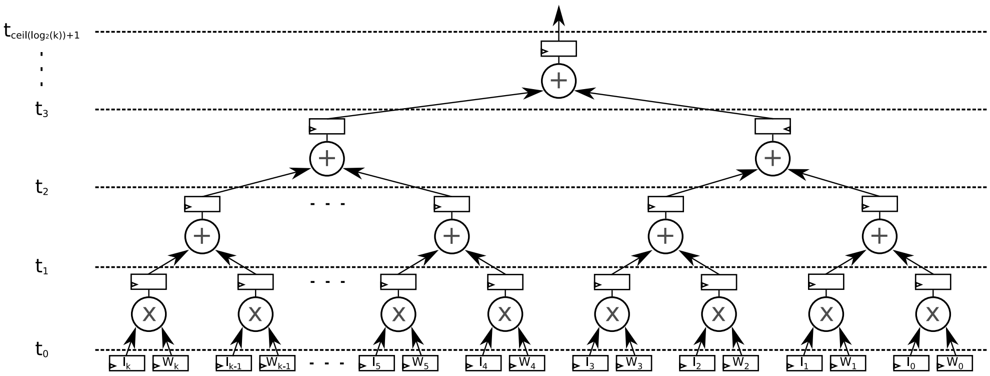


Figure 5.1: Fully Unrolled Multiply-Accumulate tree Architecture.

While fully unrolling the individual multiplications followed by an adder tree, as shown in Figure 5.1, has been widely used to take advantage of parallelism in multiply-accumulate operations in FIR filters and convolutions [4], it is not always ideal when considering streaming applications with a high degree of parallelism, for example in neural networks. FIR filters and convolutions are usually of smaller dimensions, compared to neural networks, having less coefficient storage requirements and workload to accelerate. The former does not hinder on-chip storage while the latter calls for more parallelism. MAC tree architectures are less flexible and adaptable to varying filter dimensions, being underutilised when computing a smaller filter or requiring complex partitioning of larger filters in order to fit. Meanwhile, latency for each pass remains the same. The computation at each layer of a neural network can be decomposed into a large vector-matrix multiplication that enables the sum of products for each neuron in the current layer with each in the previous layer and the corresponding weights. But this full unrolling is costly in terms of hardware, and the scale of these matrix multiplication units can hamper achievable frequency. Furthermore, this typically results in a layer-wise operation that necessitates significant transfers on/off chip between layers. These overheads are amortised by batch processing. Neural Networks, however, typically contain sufficient numbers of neurons to offer a coarser grained level of parallelism to exploit, where each neuron is mapped to a computational element, and its own results are calculated serially. This offers less dense signal connectivity and enables the compute units to operate at high frequency, while also affording flexibility to different network parameters, and avoiding memory transfers.

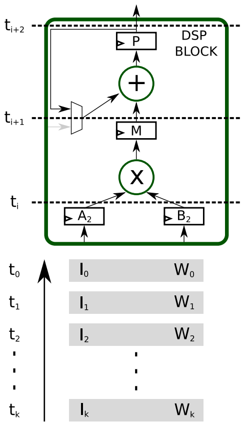


Figure 5.2: Serial Compute Architecture, using DSP blocks.

The serial multiply-accumulate operation, inherently abundant in neural networks, can be more efficiently mapped to a single DSP block, fully utilising the multiplier and adder, while also requiring less memory for the intermediate results since they are consumed in a single register throughout the flow of inputs. This method is shown in Figure 5.2. In contrast, adders in direct MAC trees are usually implemented either in fabric or in DSP blocks. In the first case, the adders consume FPGA resources that could otherwise be used to support the neuron operation (e.g. Control logic, memories in LUTRAMs, registers) while also consuming more power since functions implemented in a DSP block use less power compared their equivalent implementations in logic [15]. In the second case, using only the adder in a DSP block, underutilises the DSP block capabilities, leaving the multiplier unused, which may have significant impact to neurons and networks with large number of inputs. Systolic array implementations can efficiently utilise both components of the DSP block, but require considered data scheduling at the inputs to the array.

	k=100		***k=128***		***k=256***	
	Latency	DSP Blocks	Latency	DSP Blocks	Latency	DSP Blocks
Serial Compute	101	1 (100)	129	1 (128)	257	1 (256)
MAC tree	14	149	14	191	16	383

Table 5.1: Latency and resource utilisation of the two compute methods.

Table 5.1 shows the DSP utilisation and latency for each of the two MAC architectures for a neuron with k inputs, as depicted in Figures 5.2 and 5.1. For simplicity, the input data from the previous layer are assumed to be already loaded in the registers while routing complexity is not considered, both of which favour the MAC tree architecture. Each multiplier along with each adder that follows in the first row of the MAC tree is considered to be mapped to a single DSP block, using their cascade interconnect. To enable direct comparisons between the two compute methods, each of the adders that follow is assumed to be mapped to a DSP block instead of FPGA fabric. This would enable the MAC tree to achieve higher operating frequency, as close as possible to the serial compute frequency, at the cost of underutilising the DSP block capabilities by not using the multiplication while requiring two clock cycles latency for each addition. Starting with the MAC tree, its latency scales according to the next greatest power of two of k. The MAC tree offers latency improvements that range from 6.2× to 15.1× while consuming 148× to 382× more DSP blocks. The benefits of the MAC tree in terms of latency are disproportional compared to the resources used, thus less efficient. Although it might be argued, under ideal circumstances, that the the MAC tree is able to generate a new output every clock cycle whereas the serial compute every k cycles, this can be compensated with the coarser, per neuron, parallelism. For example, for k inputs, exploiting parallelism for at least k neurons (numbers reported in brackets in Table 5.1) in a layer results in having exact same throughput while consuming 27% to 33% fewer DSP blocks. Assuming, that there is sufficient parallelism within a layer to do so.

5.4 Implementation

5.4.1 Overlay

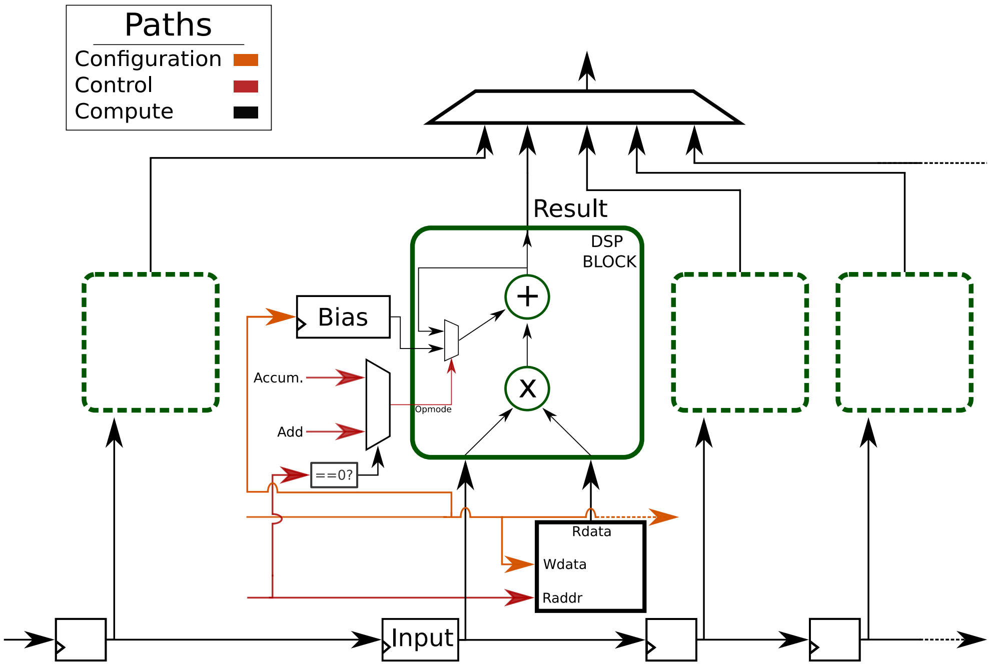


Figure 5.3: Diagram that shows configuration, control and compute paths for each neuron compute unit.

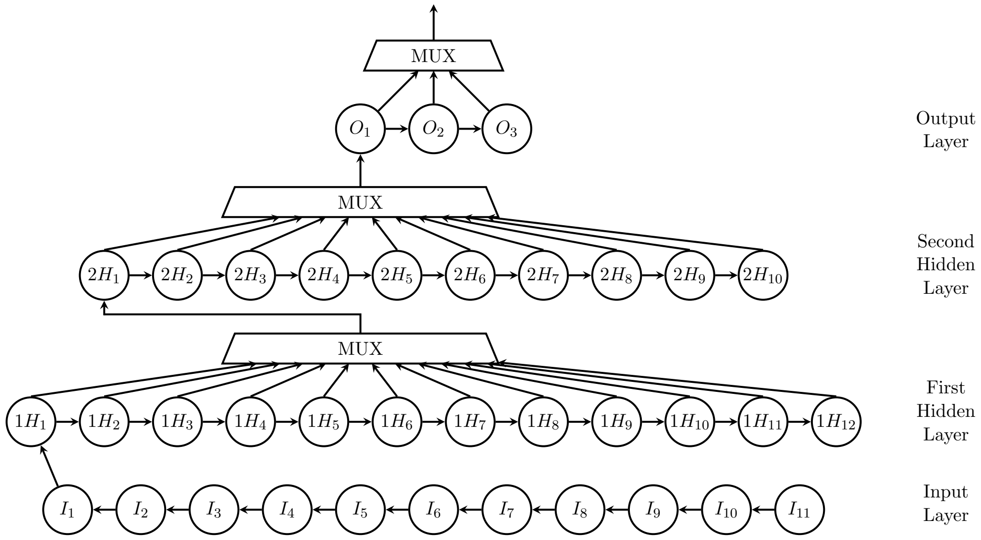


Figure 5.4: Proposed neural network overlay architecture, mimicking the structure of the network.

The overall proposed overlay aims to take advantage of DSP blocks' capabilities, while also being flexible, allowing the configuration of coefficients for rapid neural network iteration. The overlay is also able to adjust its processing latency to the required topology. It is tailored for deployment at the edge by maintaining low resource utilisation while operating at near the theoretical maximum frequency of the platform. The latter minimises the impact of clock independent leakage current, resulting in improved energy efficiency. More specifically, the overlay takes advantage of the efficient mapping of the multiply-accumulate operation, the main computation in neural networks, on DSP blocks. Instead of targetting peak performance however, by fully taking advantage of the parallelism, in this overlay, each neuron's operation is mapped to a single DSP block. As a result, each DSP block calculates its output sequentially, thus enabling a flexible and programmable architecture. Moreover, by being able to operate at a relevantly high frequency, the performance overhead is somewhat mitigated. The overall proposed overlay diagram is shown in Figure 5.4, while a more detailed compute unit architecture and its datapath are presented in Figure 5.3.

The input data along with a valid signal stream into the overlay serially. Each neuron receives a new input which is subsequently passed to the next neuron in the same layer. In the overlay architecture, one neuron in each layer generates an output in any given timestep. When there are more neurons in a layer than in the previous layer, this requires stall cycles to ensure that only a single neuron from the previous layer outputs to the shared bus. A programmable stall mechanism introduces these stall cycles automatically where necessary to maintain the regular dataflow. Before receiving any inputs, the overlay is configured by setting the number of neurons used at each layer (the reset address for each layer), when to stall and for how many clock cycles, along with the network weights and biases. The weights are stored in LUTRAMs while the rest of the configurations in registers.

After the overlay is configured, the input data flows into the first layer, from neuron to neuron, as shown in Figure 5.3, along with a valid signal that is used to enable the address counter. The address counter increments accordingly and addresses the LUTRAM where the weights are stored, feeding the corresponding weight of each input to the DSP block. The counter resets when it reaches its configured reset address, enabling the proposed overlay to adjust its latency, and as a result its performance, to the topology of the configured network. The address counter is also used to alternate between two DSP opmodes. Instead of resetting the accumulation register at the beginning each iteration, the DSP block OPMODE changes to add the multiplication's product to the bias (C input of the DSP block). Avoiding, as a result, redundant additions with zero, replacing them instead with the bias additions that would normally take place after all the weighted inputs have been accumulated. For the rest of the computation, a different DSP block OPMODE is used to accumulate the product. When the address counter of a neuron reaches its reset address, meaning that the computation of the neuron has completed, a pulse is generated. The pulse is delayed by three clock cycles for synchronization, and fed to a state machine that generates the enable signal for the first neuron in following layer. The enable signal subsequently propagates from neuron to neuron similarly to the first layer.

Meanwhile, a multiplexer between two layers, addressed by the counter of the first neuron in the following layer, selects the appropriate input from the previous layer. Each input to the multiplexer is reduced from 48 bit, the output of the DSP block, to 27, the input of the following DSP block, by selecting the appropriate bit range according to the fixed point representation used. The selected output is then passed to another multiplexer that implements the ReLU activation function, by checking whether the MSB is set to 1, and passing the input to the next layer accordingly.

5.4.2 Stall Mechanism

To make the processing and data flow stall for a number of clock cycles, the *valid* input signal is manipulated accordingly. The stall mechanism, shown in Figure 5.5, is configured externally before processing takes place. The valid signal is connected to the *en_in* port, while the rest of the ports, are connected to the external component, i.e. the ARM core. The stall mechanism takes two 5 bit inputs along with their active high configuration signals and stores their

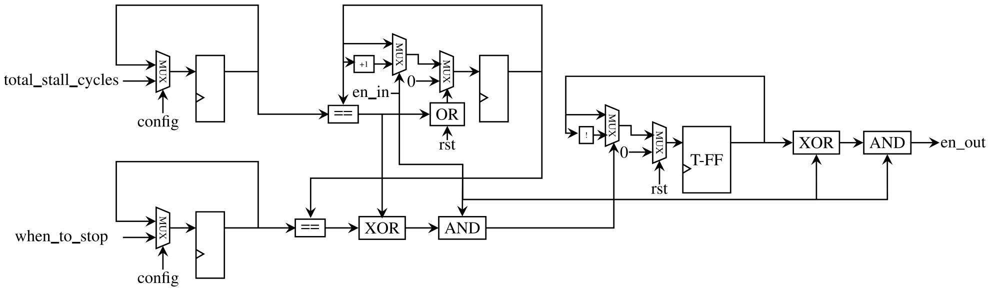


Figure 5.5: Programmable stall mechanism enabling variable sized networks to be implemented.

values to registers. The *total_stall_cycles* input takes the total number of clock cycles (processing cycles + stall cycles), while the *when_to_stop* input takes the number of processing cycles.

The counter increments as long as *en_in* is active, which means that input data flows to the accelerator. It resets to zero when it is synchronously reset or when it reaches the *total_stall_cycles*. The counter output is used to detect whether it has reached the point where it has to stall (*count==when_to_stop*) or whether it has reached the point to stop stalling (*count==total_stall_cycles*). The output flip-flop inverses its output accordingly, and combined with *en_in*, controls when to stall the overlay. Where not needed, the stall component can be disabled by setting both, *when_to_stop* and *total_stall_cycles*, to the same value. This causes the XOR gate not to generate an active pulse to trigger the T flip-flop.

5.4.3 Dataflow and Compute Timing Diagram

As described in Section 5.4.1, input data flow in the overlay serially, triggering the neuron compute units the one after the other. Due to the weight depth of each neuron within a layer being equal, neurons are expected to fire one after the other, propagating the serial dataflow to the following layers. This means however that if the number of inputs to a layer is less than the number of neurons in that layer, stall cycles are required to maintain this serial firing. The aforementioned data flow and stall operation are graphically depicted over time in Figure 5.6.

5.4.4 Case Study

Table 5.2 summarises the datasets and networks used in this case study. The networks were trained with Tensorflow [56] to obtain the accuracies shown in the same table. These were chosen to represent a range of application domains and to match or exceed the complexity of NNs that have been more widely targetted for acceleration, for instance in [73] and [39]. The proposed overlay,

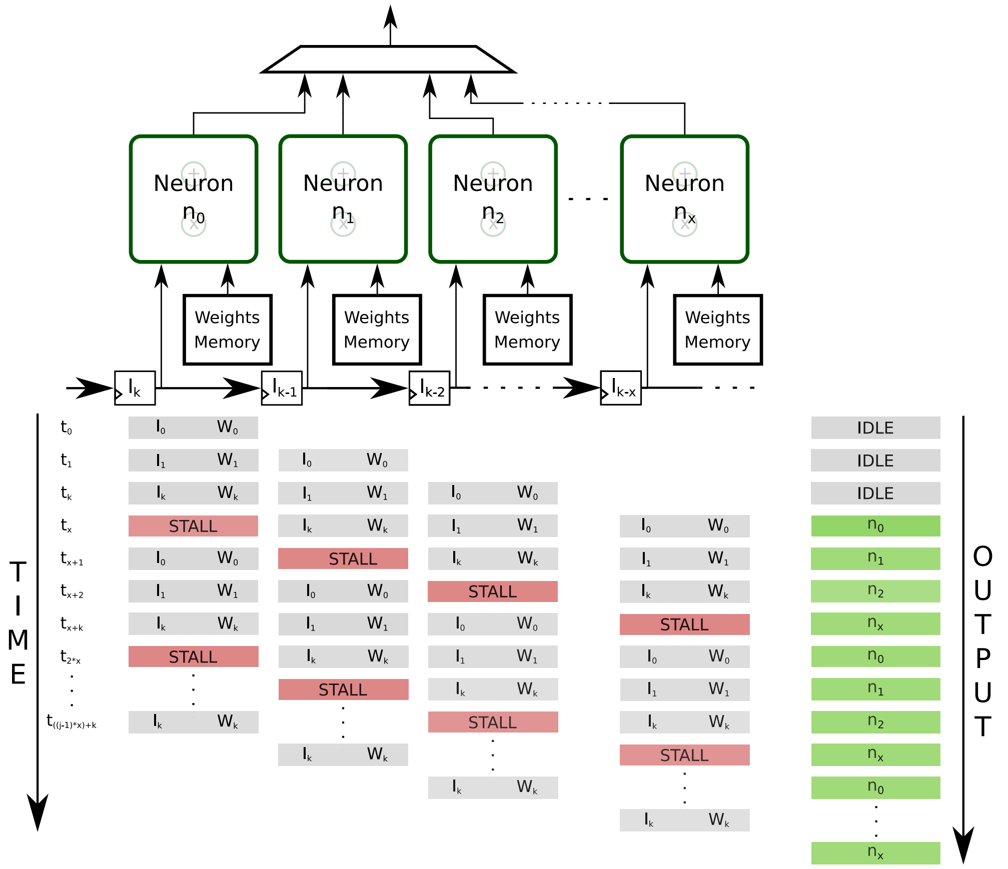


Figure 5.6: Diagram that shows the dataflow and compute allocations over time-steps.

designed for inference, does not implement an activation function at the output layer, since the required comparisons can be more flexibly made in software and raw outputs can used as feedback for fine-tuning.

The proposed overlay is tailored to the features of the DSP48E2 block on the Zynq Ultrascale+ ZU7EV. This DSP block comprises a 27×18 bit multiplier with a 48 bit accumulator/adder. After exploring the networks and datasets in Python, a representation with 12 fractional bits has been decided as it results in no accuracy reduction. Although quantisation has been shown to reduce coefficient wordlength at the cost of tolerable loss overhead, it has not been used in order to avoid additional complexity in the training steps. The overlay uses 18 bit weights, 48 bit biases, which can be configured externally, and 27 bit inputs. By analysing the topologies of the NNs included in this case study, an overlay with a 11-12-10-3 configuration has been implemented along with the stall mechanism using Verilog. The proposed architecture has been behaviourally simulated and verified against the expected output in each dataset. The design has then been synthesised and implemented using Vivado 2018.2 and all the results are post place and route.

The proposed architecture can perform computations at maximum fre-

Dataset	NN Topology	Train Entries	Acc. Train	Test Entries	Acc. Test
Customer Churn	11-6-6-1	8000	84.26%	2000	82.95%
Diabetes	8-12-8-1	768	78.39%	-	-
Iris	4-10-10-3	120	98.33%	30	96.67%
Overlay	11-12-10-3	-	-	-	-

Table 5.2: Case study neural networks configurations.

quency of **770MHz**, which is close to theoretical maximum, 775MHz, of the device's DSP blocks [121]. The resource utilisation of each module is summarised in Table 5.3. It is important to note that the design of the stall mechanism results in an insignificant area overhead, while the total utilisation is very small, meaning this architecture could be scaled up significantly on this device.

Module	LUTs	LUTRAM	FFs	DSPs
Overlay	796	225	2552	25
Stall Mechanism	24	0	16	0
Total	819	225	2568	25
Available	230400	101760	460800	1728

Table 5.3: Resource utilisation on the Zynq Ultrascale+ ZU7EV.

5.5 Results and Discussion

From the simulations, the number of clock cycles for each network to process the first dataset entry has been extracted, labelled latency, along with the clock cycles required to process a following entry when the pipeline is saturated, labelled interval. The number of stall cycles is also provided to quantify the stalling overhead. In each case the maximum operating frequency of 770MHz is taken into consideration, showing how that translates to actual runtime in Table 5.4. Compared to other FPGA implementations in the literature, the authors in [73] implement a neural network for gas classification on a Xilinx Zynq XC7Z010T using Vivado HLS v2016.1. The architecture uses fixed point arithmetic and operates at 100MHz, as well as using the more expensive Sigmoid activation function. Parallelism is exploited with pragmas for loop unrolling and pipelining, and they report a latency of 540ns for their simpler 12-3-1 network topology, which is about 10× slower compared to the proposed overlay for a 8-12-8-1 network that results in a 48.026ns latency.

	Clock Cycles			Time (ns)		
Dataset	**Latency**	**Interval**	**Stall**	**Latency**	**Interval**	**Stall**
Customer Churn	32	11	0	41.536	14.278	0
Diabetes	37	12	4	48.026	15.576	5.192
Iris	35	10	6	45.43	12.98	7.788

Table 5.4: Theoretical timing results for the overlay.

To provide a reference for comparison, all three neural networks were processed in software on the ARM Cortex-A53, operating at 1.2GHz bare-metal, as found in the same Ultrascale+ device. The neural networks were also processed on a desktop PC running Ubuntu Linux 18.04 on an Intel Core i7-6700 CPU, at 3.40GHz. Fixed point representation was also used for the software, implemented in C. From the execution time measured and the theoretical timings of the overlay, the inference throughput has been calculated for each network in Table 5.5.

Neural Network	**Inferences/sec.**		
	ARM-A53 @1.2 GHz	**Core i7-6700 @3.40GHz**	**Overlay @770MHz**
Customer Churn	0.151×10^6	3.618×10^6	70.04×10^6
Diabetes	0.089×10^6	2.201×10^6	64.20×10^6
Iris	0.099×10^6	1.29×10^6	77.04×10^6

Table 5.5: Inferences per second on the different architectures.

The proposed overlay offers a significant performance improvement, compared to the embedded ARM core, able to process the networks in this chapter's case study at a significantly greater rate. The proposed overlay is at least 19× faster than the desktop class Intel Core i7-6700.

5.6 Summary

A lightweight streaming neural network overlay, has been presented in this chapter. The overlay was optimised for the high performance DSP blocks in modern FPGAs and exploited their programmability. Moreover, it reduced dependency to the backend toolflow and enhanced flexibility and programmability of FPGAs in the neural network domain. The implemented overlay architecture maintained low resource utilisation and operated at near the theoretical maximum of the platform. It also offered significant performance improvement compared embedded and desktop CPUs, offering 19× the performance of an Intel Core i7-6700, while being about 10× faster compared to previous work on FPGAs that used HLS. The high operating frequency, that also contributed in minimising the impact of leakage current, coupled with the minimal resource utilisation and significant performance improvements, rendered the proposed overlay ideal for processing at the edge.

Chapter 6

Lightweight Streaming LSTM Neural Network Overlay for FPGA

6.1 Introduction

The increasing popularity of Neural Networks has driven significant efforts to accelerate computation of different network topologies on a heterogeneous spectrum of computing platforms, from powerful servers to less capable devices at the edge. NNs are typically trained on highly parallel GPU platforms due to the high computational workloads that suit offline centralised implementation. Inference scales well on more constrained devices since various optimisations can be applied [51, 74, 120, 122]. Hence, there has been ample research on architectures for NN inference acceleration on a variety of platforms. A number of silicon vendors have also augmented processors with specialised neural processing units that offer the required parallelism, enabling significant acceleration of these workloads [61, 62].

LSTMs combined with fully connected layers are an ideal combination for processing time-series data in such lightweight applications, especially in wakeword or event detection, and hence the focus of this chapter is on these. Lightweight LSTM NNs have found applications in healthcare [123], weather prediction [49], and network security [124], among other applications. While fully connected layers are the most regular form of NNs, LSTMs include data feedback from previous timestep results and disrupt the regular flow of data, which in turn breaks up back to back matrix multiplications. Although this can be somewhat alleviated with batch processing or by executing multiple NNs simultaneously, both methods increase the volume of intermediate results to be cached, and may not fit all application data rate requirements. In addition, LSTMs use a wider variety of activation functions compared to traditional fully connected layers, requiring a mix of them in a single layer, including the more complex tanh and sigmoid functions.

In this chapter, an overlay architecture that is able process fully connected and LSTM layers flexibly is proposed, while operating at high frequency. Low level computations are abstracted to building blocks that can easily be replicated to reflect the structure of a model, while tailoring the datapath to their complex dataflow pattern. The proposed architecture is self contained and flexibly reconfigurable to implement different models and adapt to weight updates. The proposed approach is tailored to operate within edge SoC environments and can be used to accommodate lightweight to moderate NN workloads. It operates in streaming mode, with computations carefully mapped to DSP blocks, each mimicking the operation of a neuron, leaving LUTs for weight storage and other functionality. This architecture caches very few intermediate results as they are consumed by subsequent processing units in a streaming manner. Finally, the overlay can be tailored to a specific set of models or support models that fit the size constraints without hardware reconfiguration.

The work in this chapter has been submitted for publication to:

- L. Ioannou and S. A. Fahmy. Streaming Overlay Architecture for Lightweight LSTM Computation on FPGA SoCs. *Submitted to: ACM Trans. Reconfigurable Technol. Syst.* [5].

6.2 LSTM Background

LSTM operation, along with supporting equations, are discussed in more detail in Section 2.3.3. Compared to feedforward NNs, LSTMs have feedback connections from previous outputs (C_{t-1} and H_{t-1}). These dependencies restrict their performance while making routing, and dataflow in general, in custom architectures more complex. Nonetheless, common computing patterns in LSTMs and fully connected layers exist in equations 2.4 to 2.7. A challenge in LSTMs however is the fact that the dimensions of W and U are not necessarily the same. The former depends on the number of input features and the number of units while the latter depends solely on the number of units of the LSTM. This translates to unbalanced latencies, when the two are computed separately. These two matrices can be concatenated into one, creating a single larger matrix with the same dimensions across all gates. This not only balances the compute latency within each gate but also makes the compute pattern of each gate the same as the the multiply accumulate operations in the fully connected layers. Equations 2.4 to 2.7 can be mapped to a single neuron in a fully connected layer, thus an LSTM unit occupies the equivalent of 4 neurons in a fully connected layer.

6.3 Related Work

Previous related work that targets LSTMs in the embedded domain is presented in [50–52, 63, 76], demonstrating the benefits of custom computing architectures in energy efficiency and performance, compared to software programmable computing platforms. Specifically, the work in [52] uses quantised (6–16 bit) LSTM models for speech recognition enabling their design to keep weights and intermediate results on chip and avoid energy consuming external memory accesses. The authors implement a matrix-vector multiplication unit that partially unrolls parallelism within an LSTM layer and is time-multiplexed for a complete layer computation. Their proposed implementation operates at 100MHz on a Xilinx Zynq XC7Z045 FPGA and shows a distinct advantage in terms of energy efficiency compared to a high-end NVIDIA GeForce Titan X GPU.

The work in [50] presents a bi-directional LSTM for optical character recognition that uses 5-bit weights and fits in the on-chip memory of a Xilinx Zynq XC7Z045 device. The authors implement a single LSTM cell and unroll the computations of each gate in it, time multiplexing the instance according to the dimensions of each LSTM layer. In addition, the authors take advantage of the fact that in bi-directional LSTMs, two inputs are processed at a time and overlap their computations in order to alleviate the idle cycles between dependent LSTM iterations. The effectiveness of this approach is evaluated by implementing various designs, starting with a design that uses a single instance of their proposed architecture, exploring its scalability by instantiating 6 such computing blocks. For single input inference, the single instantiation design offers 152 GOPs throughput at 166 MHz or 130 GOPs at 142 MHz. The design that incorporates 6 instances, operates at 142MHz and obtains 308 GOPs for single input inference whereas for offline processing, in which batch processing with 6 images is used, 693 GOPs is achieved. The results show that the proposed approach does not scale well for single image inference, offering only a 2.4× scaling when instantiating 6 instances of the design. The proposed architecture scales better with batch processing, offering 5.3× the baseline throughput, however this is still not linear. The authors further expanded their work in [51] exploring extreme quantization methods using 1–8 bits. Their exploration yields a design that operates at 266MHz,on a Xilinx Zynq UltraScale+ XCZU7EV, and offers throughput that ranges from 661 to 4201 GOPs for the different precisions used.

A unified LSTM accelerator flow is presented in [76], that takes as input a trained model and the target FPGA device specification and generates an accelerator design accordingly. The proposed flow generates an accelerator for each model and the accelerator is time multiplexed for each LSTM layer within a network. The generated accelerator does not support the activation function computations, which are offloaded to software on the ARM core. The LSTM gate results are therefore transferred to the off-chip memory, processed by the ARM core and then transferred back to the accelerator, resulting in frequent external memory transfers that are energy demanding and add a performance overhead. The overall SoC design uses the ARM core for coordinating the accelerator throughout LSTM computation, in addition to the activation function computation, thus not offering a self-contained accelerator solution. The authors implemented various versions of their accelerator, using 16 bit fixed point, 32 bit floating point and their equivalent pruned versions, on a Xilinx Zynq XC7Z020 FPGA operating at 150MHz. The pruned equivalents reduced the inference time by 32% and 42% for the fixed and floating point implementations respectively. Compared to the work in [54], the authors obtained about 10× improved inference time for the fixed point implementation whereas the floating point implementation offers negligible acceleration. Both implementations however demonstrate better power efficiency, being 11.7% and 0.32× more power efficient. The authors extend their evaluation by exploring the scalability of their floating point architecture by implementing a larger LSTM layer on a Xilinx Virtex VX485T, FPGA, obtaining 10.7 GFLOPs.

The mapping of large LSTM layers on Xilinx Virtex VX690T and Zynq 7Z045 FPGAs is explored in [63]. The authors aimed at optimising the matrix-vector multiplications and their dependencies in LSTMs with weight matrix partitioning and an optimised batch processing strategy. Their proposed approach is tailored for batch processing and uses 16-bit fixed point representation while operating at 125 MHz and 142 MHz on the Virtex 7 and Zynq devices respectively. The authors obtained 356 GOPs on the Virtex 7 and 221 GOPs on the Zynq, demonstrating improved performance and energy efficiency compared to an Intel Xeon E5-2665 CPU, Nvidia TITAN X Pascal GPU, and other related previous work on FPGAs.

Other related work that targets the same LSTM models as the ones used in this chapter, therefore better suited for direct comparisons, is described in [53–55, 72]. The work in [54, 55] presents three different LSTM co-processors on an FPGA that balance memory bandwidth and internal storage utilisation to optimize performance per unit power. The first streams all the necessary data from off-chip memory, the second stores all data on chip and the third is a more balanced design. The authors test their co-processors on a character level network comprising 2 LSTM layers, each with 128 units. The NN model used, includes a fully connected layer at the end that uses 65 neurons for the final classification, which has not been included in the architecture. All co-processors use Q8.8 fixed point representation and operate at 142MHz on a Xilinx Zynq-7000 FPGA. The three architectures are compared in terms of resource utilisation and memory bandwidth, and shown to provide orders of magnitude better performance per unit power compared to embedded processors, with the design that stores all data on-chip being the most efficient in terms of performance per unit power.

A stochastic computing based LSTM implementation is presented in [72], focusing on reducing the hardware cost and power consumption of fundamental arithmetic components within an LSTM. The authors evaluate their approach on an LSTM layer with 16 hidden units, trained on the MNIST dataset. As with other previous work, the fully connected layer comprising 10 neurons was not included in the architecture. The authors implement their designs on a Xilinx Zynq-7000 FPGA, operating at 100MHz, and make comparisons between the baseline and their two proposed designs in terms of power consumption, classification accuracy, and runtime. They show a tradeoff between runtime and resource utilisation and power, demonstrating their ability to scale to a suitable specification.

The authors in [53] propose a high throughput and energy efficient LSTM architecture utilising an approximate multiplier. This results in a multiplier-less implementation and effectively reduces power consumption and resource utilisation, at the cost of multiple and variable clock cycles due to its data dependent nature. As a result, performance is less predictable. Hierarchical pipelining is used to improve performance by overlapping these computations. The proposed approach applies range-based linear quantization to a language model LSTM, with the same configurations as the model in [54, 55], on a Xilinx Zynq XC7Z030 FPGA. The implemented design uses 8-bit fixed point precision and operates at 100MHz.

Additional related work can also be found in [16, 48, 117, 118], in which the authors have focused on accelerating LSTM computation on more capable FPGAs with PCIe interconnect in servers. The approaches used in these works are not suitable for constrained edge devices, where fully unrolling computations cannot be achieved.

The majority of previous related work focus solely on the LSTM computation, not including the implementation of fully connected layers and thus do not provide a complete edge solution. Moreover, all reported operating frequencies are well below the devices' theoretical maximum, which results not only in lower performance but also in lower energy efficiency due to leakage currents [16]. Meanwhile, generic NN accelerator architectures cannot implement LSTMs without modification, or suffer a significant performance and energy overhead due to the dependencies on previous outputs necessitating transfers to off-chip memory. This calls for more programmable custom computing architectures for LSTMs. The work in [3] described a streaming overlay for fully connected layers utilising the DSP blocks of a Zynq Ultrascale+ ZU7EV FPGA. That design was shown to achieve close to the theoretical maximum frequency while using minimal resources, but supported only feed-forward networks with the ReLU activation function, and was not shown to scale. A streaming overlay architecture is proposed, that supports the computation of LSTM and Fully Connected layers, offering a complete edge solution, while supporting the more complex Sigmoid and Tanh activation functions through approximations. The overlay heavily exploits programmable DSP block capabilities and is carefully designed to maintain short critical paths and relatively moderate routing complexity in order to achieve high operating frequency. At the same time, the overlay concept offers a more programmable solution, compared to fixed accelerators, allowing model parameters to be updated. The proposed architecture also operates in streaming mode, which is more responsive compared to batch processing and is therefore more suitable for devices at the edge.

6.4 Proposed LSTM Architecture

This section outlines the various design choices and operation of the main building blocks of the proposed streaming architecture that can be configured to implement LSTM or fully connected layers. The architecture uses DSP blocks for the neural network computations while also supporting other widely used settings in these layers (e.g. the option to return sequences in LSTM layers). Systolic arrays are widely used for NN inference, as they very efficient for matrix-matrix multiplications, however this is not a requirement for their operation. This helps reduce the overheads of loading weights. In streaming processing, as targeted by this work, systolic arrays would be less efficiently utilised due to the lack of batching and shared weights. Additionally pipeline parallelism would be harder to achieve due to the dependencies inherent in LSTMs. Hence, an alternative approach is adopted, to implement the multiply-accumulate operations as outlined in the following sections.

6.4.1 Proposed Neuron Architecture

The proposed architecture takes the neural network computation and does away with the matrix representation, instead opting for neuron-based parallelism where each neuron is implemented as a computational unit that processes its output in a serial manner. Each neuron is mapped to a single DSP block, supported by the required control logic and memory to enable it to fully implement the neuron's function. It operates in one of three modes: configuration, control, and compute. Initially, configuration takes place, in which all weights, biases and activation functions are set for each neuron. Once configuration is complete, compute and control operations run concurrently

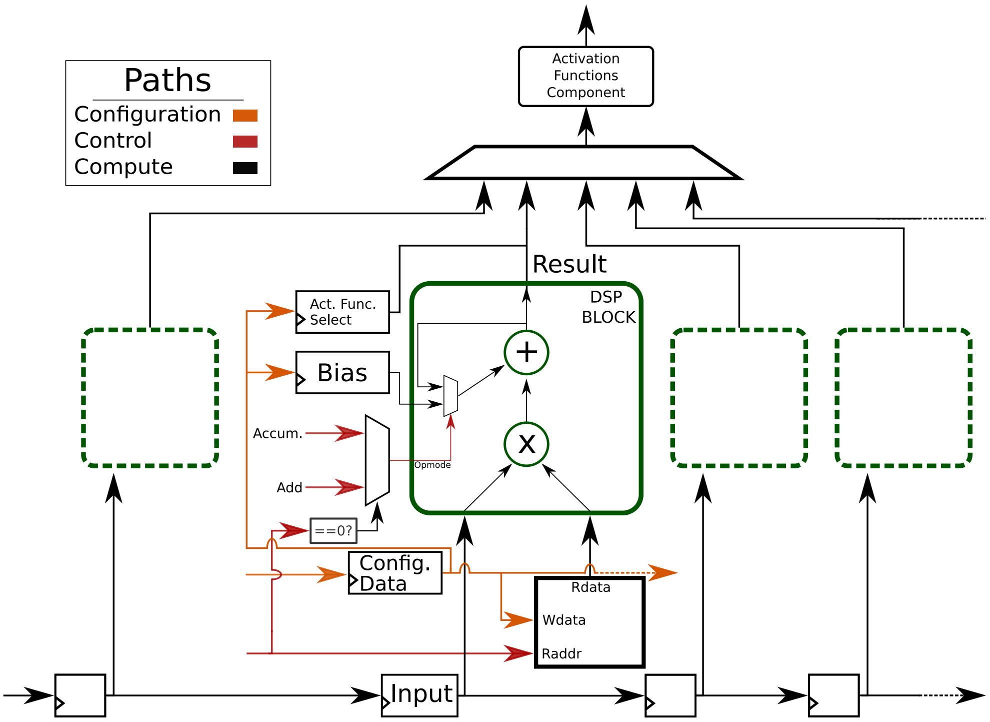


Figure 6.1: Neuron Architecture showing the configuration, control and compute paths.

and input data starts to flow in. Outputs from the previous layer stream serially, one for each neuron from that layer at a time, and are multiplied in each neuron in the current layer by the corresponding weight stored in the weight memory. An address counter manages weight memory addressing and DSP block opmode selection. Each DSP block operates in one of two different opmodes, the first input-weight product is added to the configured bias, while subsequent products are accumulated with this sum. This is enabled by the dynamic DSP block control in modern Xilinx FPGAs [104]. This results in not having to reset the accumulation register before a new neuron computation and saves a clock cycle compared to adding the bias after the completion of multiply-accumulate. The serial dataflow of the overlay, coupled with the more minimal use of resources and fanouts, allows for a more scalable architecture in which each self-contained neuron can be replicated as many times as needed to form a layer, and each layer in turn to form a lightweight neural network on chip.

6.4.2 Neural Network Multiply-Accumulate

The *Neural Network Multiply-Accumulate architecture*, shown in Figure 6.2, consists of a series of DSP blocks, each calculating the multiply and accumulate operation. The inputs flow in each layer serially, multiplied by their corresponding weights and accumulated in each DSP block.

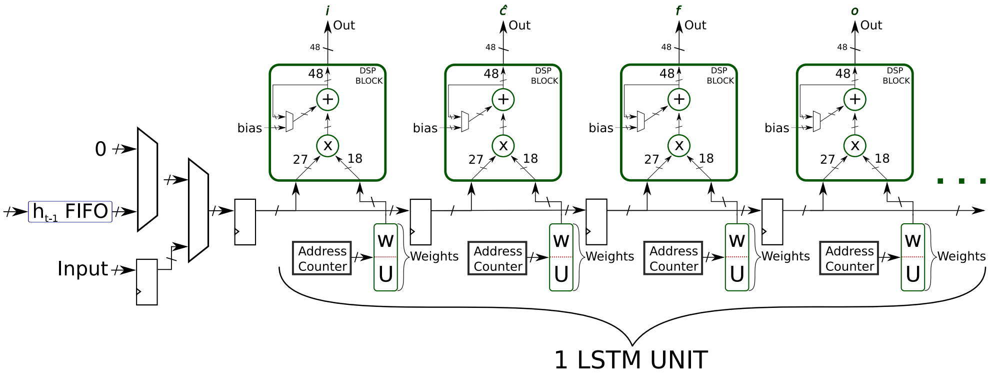


Figure 6.2: Neural Network Multiply-AccumulateArchitecture.

The wordlength of inputs, weights, and biases are defined to fit the DSP48E2 primitive in Xilinx UltraScale+ devices. Inputs are 27 bits, weights 18 bits, and biases are 16 bits. All wordlengths are signed and use 11 fractional bits. The accumulation register within the DSP block is 48 bits and uses 22 fractional bits. Unrolling parallel operations at each neuron results in a naturally balanced workload between DSP blocks, while being more resource efficient by using both multipliers and adders within the DSP blocks. This unrolling scheme enables the overlay to map all the neurons of lightweight to moderate NNs on chip, and by accumulating all the intermediate results of each neuron within a single register, it reduces on chip memory requirements.

Moreover, this arrangement enables the serial flow of input data from neuron to neuron, which results in relatively low fanout, while also passing all input data to each neuron just once, avoiding additional storage and operational overhead to cache and re-flow previous input data. The use of DSP blocks coupled with the more compact dataflow result in a short critical path and more manageable routing which in turn enables a high frequency of operation. The *Neural Network MAC*'s operation deviates from the more mainstream acceleration methods on larger devices used for the matrix-vector operations (e.g. systolic arrays) to better suit the constraints of edge devices.

When the *Neural Network MAC architecture* is configured as a fully connected layer, the data flows serially to it from the input source. If it is configured as an LSTM layer, however, the new input data flows in at first, while the previous output, H_{t-1}, stored in a FIFO, follows. Each of equations 2.4 to 2.7 is mapped to a DSP block, with 1 LSTM unit occupying 4 DSP blocks. The input selection has been implemented with multiplexers and corresponding control logic. A 2-bit register sets the desired activation function for each neuron. The top level architecture supports the ReLU, approximated versions of Sigmoid and Tanh, and a passthrough datapath in case none is selected.

6.4.3 Activation Function Approximation

Various activations functions are used in neural networks for non-linearities, with ReLU, Sigmoid, and Tanh being the most widely used. Although activation functions may not be the most computationally complex part of neural network architectures, the use of exponents in Sigmoid and Tanh functions make them difficult to implement in embedded architectures.

Although ReLU is suitable for hardware implementation, various NN applications call for the use of Sigmoid or Tanh functions. For example, the forget gate in an LSTM layer uses the Sigmoid function, the output of which determines the percentage of information to be kept from the previous layer. This has led to the exploration of alternative ways to implement these functions more efficiently, especially in fault-tolerant, approximate computing applications. The majority of previous neural network implementations map the activation functions in look-up-table memories, one for each function. With this approach, accuracy depends on the granularity of the look-up-table, with error being inversely proportional to the size of the table. This approach can use significant area in lightweight unrolled implementations, in which multiple tables are required. Furthermore, in architectures where multiple activation functions need to be supported, separate look-up-tables are required, of which only a subset are used at any one time. Other previous work has focused on piece-wise approximations of these functions [125], while others have approximated the active region of these functions linearly with cut-off regions [53]. Another example of the latter is the hard sigmoid activation function in Tensorflow [56]. More complex activation function architectures have also been presented in [126], where implementations in half and full precision floating point have been explored.

An activation function approximation using piecewise linear approximation is proposed, while also considering how some coefficients can be modified to be more efficiently implemented in hardware. Moreover, since only one activation function is used at a time in each unit, common expressions are merged between the different activation functions in hardware. As a result, the logic required is minimized, contributing not only to reduced area but also improved performance. The proposed activation function architecture can be configured to any one of the most popular activation functions at runtime, without re-implementation or re-loading of a lookup table, while maintaining low area utilisation and high performance.

```
def custom_sigmoid_hw(x):
  point_twenty_five = _constant_to_tensor(0.25, x.dtype.base_dtype)
  point_five = _constant_to_tensor(0.5, x.dtype.base_dtype)
  x = math_ops.mul(x, point_twenty_five)
  x = math_ops.add(x, point_five)
  x = clip_ops.clip_by_value(x, 0., 1.)
  return x

def custom_tanh_hw(x):
  point_seventy_five = _constant_to_tensor(0.75, x.dtype.base_dtype)
  x = math_ops.mul(x, point_seventy_five)
  x = clip_ops.clip_by_value(x, -1., 1.)
  return x
```

Listing 6.1: Approximation functions for Tensorflow.

Approximations Applied in Software

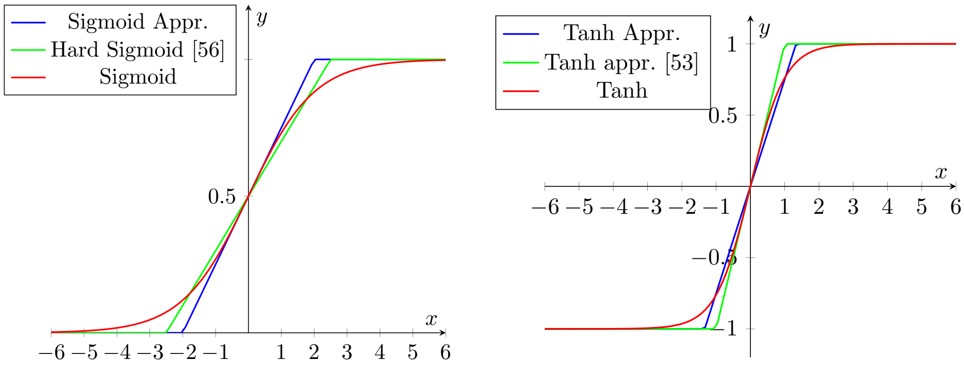
The most relevant previous work to ours in approximating the activation functions is summarised in Table 6.1, where *Sigmoid* and *Tanh* functions are bounded to 0,1 and -1,1 respectively. The approximated coefficients used in this work are modified slightly from those referenced to more efficiently suit the fixed point representation and hardware. Although the *Tanh* approximation used in [53] is simpler to implement, especially in hardware, it deviates more from the true function. All aforementioned approximations are presented graphically against the *Sigmoid* and *Tanh* in Figs. 6.3a and 6.3b.

Act. Func.	**Equation**	**Work**	**MAE**
Hard sigmoid	y=0.2x+0.5	[56]	0.019
Approx. Sigmoid	y=0.25x+0.5	[53], This work	0.033
Approx. Tanh	y=x	[53]	0.088
Approx. Tanh	y=0.75x	This work	0.063

Table 6.1: Approximated functions equations.

The main difference in the proposed approach is that the approximations can be applied during training, in addition to post-training. Similarly to how the hard sigmoid is defined in Tensorflow, by changing the coefficients accordingly, the approximations can be defined as shown in Listing 6.1 and be used to train a model in floating point. The error introduced by the approximations is therefore taken into consideration during training and is alleviated, leaving more margin for error in fixed point representation.

To demonstrate the effectiveness of this approach, comparisons with previous work on approximated activation functions are made in Table 6.1. The table also shows the Mean Absolute Error (MAE) between the baseline functions and their approximations, as calculated on 40 data points between -2



(a) Sigmoid functions.

(b) Tanh functions.

Figure 6.3: Activation functions and their approximations.

and 2 with a step of 0.1. As expected, the Hard Sigmoid approximation in Tensorflow generates less error compared to the approximation used in the proposed design, however the *0.25x* has a far more straight forward fixed point representation and multiplication. Meanwhile, the *Tanh* approximation used in [53], although simpler to implement, generates more error compared to that used in this work. Moreover, since common computations exist between the two approximations, the additional complexity introduced in the *Tanh* approximation is mitigated.

Trained with	**Sigmoid/tanh**		**Approx.**
Inf. using	Sig./Tanh	Approx.	Approx.
Train	0.3141	0.4358	0.3182
Validation	0.3370	0.4283	0.3302
Test	0.3575	0.5569	0.3751

Table 6.2: Approximated functions loss-Weather forecast.

The benefits of using the approximations are further explored during training, by training a three layer LSTM for temperature forecasting, similar to that in [127]. The weather time series dataset was used from the Max Planck Institute for Biogeochemistry to train a network with two LSTM layers, with 64 and 32 units respectively, and a fully connected layer for the output layer comprising 1 neuron. The RMSprop optimiser was used while measuring the loss with mean absolute error. The model is trained to receive the last 720 measurements that span over the last 5 days, and predict the temperature in 12 hours. Initially, a baseline model is trained for 10 epochs with Tensorflow v2.2 using the default activation functions while a variation of this model is trained using the approximated activation functions. Subsequently, inference on the two models is ran while also changing the activation functions of the baseline model to the approximations. The losses obtained from these three models are

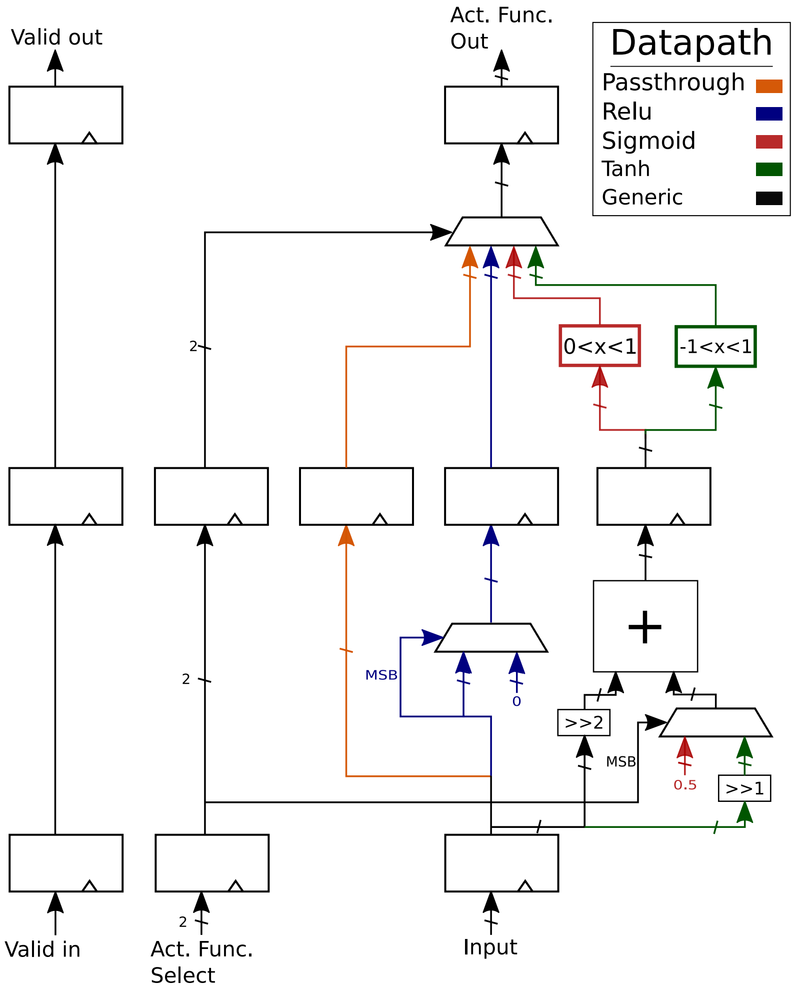


Figure 6.4: Activation functions architecture, showing the various datapaths, logic blocks and pipeline stages.

summarised in Table 6.2, showing that by using the approximated functions during training, loss is comparable to the original function implementations.

Activation Functions in Hardware

A hardware architecture is shown in Figure 6.4 that supports the most widely used activation functions, ReLU, approximated Sigmoid, and Tanh, while also providing a passthrough path in case none is needed. A key feature is that common computations between Tanh and Sigmoid approximations are merged while all the multiplications are replaced by shifts and adds, avoiding the use of computationally expensive multiplications since the coefficients are fixed.

A parametrized architecture has been created in Verilog HDL, and implemented using Xilinx Vivado 2018.2 on a XCZU7EV Ultrascale+ device. The parametrised architecture was used implement designs with various wordlengths in order to make comparisons in terms of resource utilisation. The results in Table 6.3 show that the proposed activation function architecture uses very few resources and these are able to operate at the device's maximum frequency.

Bits	Fraction Bits	LUTs	Registers
16	8	36	86
27	12	74	141
32	16	90	166
48	24	132	246

Table 6.3: Resource utilisation of the activation functions architecture.

6.4.4 LSTM Addon

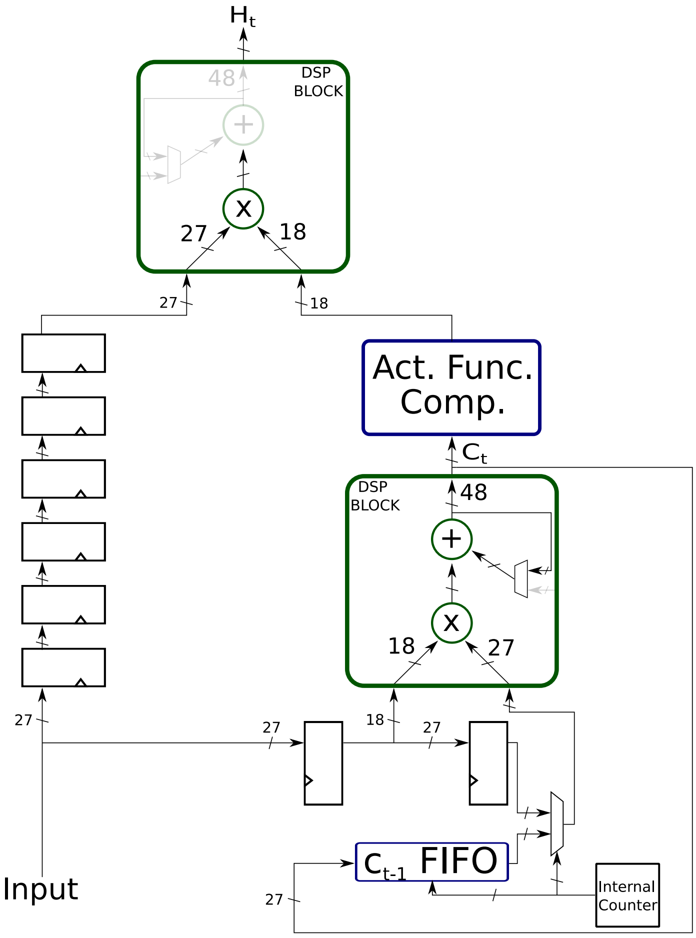


Figure 6.5: LSTM addon compute architecture, showing the various logic elements and delay registers.

The *LSTM Addon* computes equations 2.8 and 2.9 of an LSTM layer. The results of i_t, $\tilde{C}_t$, f_t and o_t flow in serially, in this particular order. This data flow pattern repeats for each LSTM unit in the *Neural Network MAC*. The i_t, $\tilde{C}_t$ and f_t are used by the datapath on the right in Figure 6.5, while

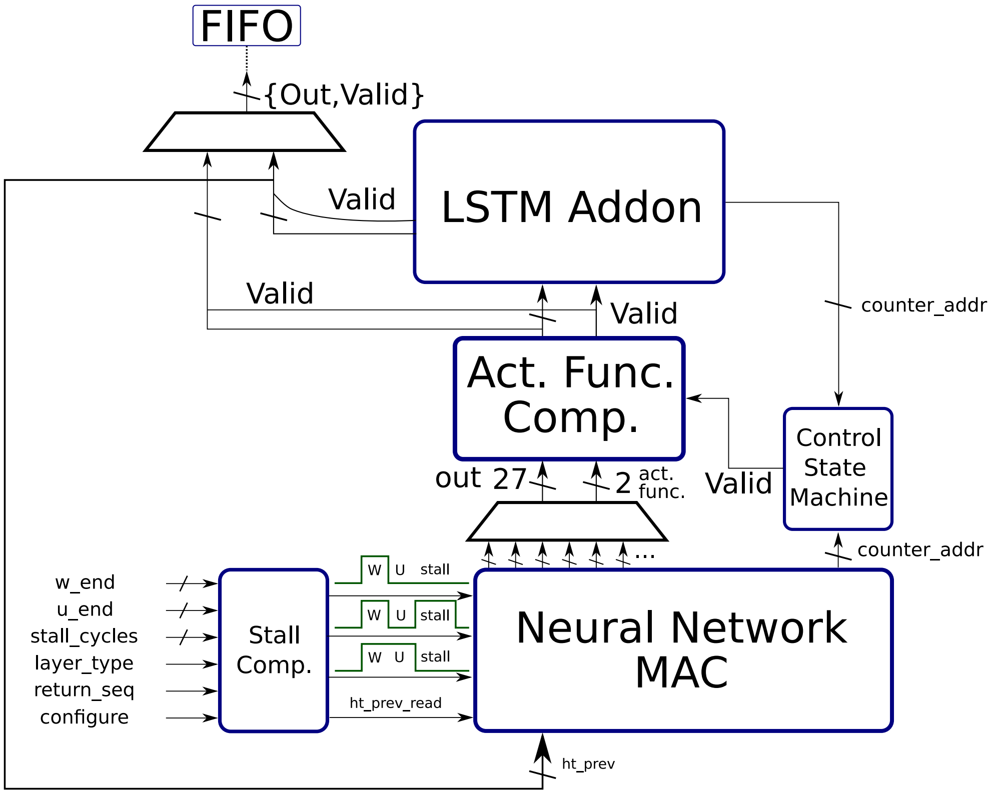


Figure 6.6: Top level layer architecture.

o_t passes through delay registers and is used only by the DSP block at the output. Initially i_t is multiplied by $\tilde{C}_t$, stored in the accumulation register of the DSP block, then the product of f_t and C_{t-1} that is read from the FIFO, is calculated and added to accumulation register. This completes computation of equation 2.8 and the result then fans-out to the C_{t-1} *FIFO*, where it is stored for the following timestep, and to the *Act. Func. Comp.*, where the activation function used in equation 2.9 is applied. The DSP block at the output uses only the multiplier and completes the computation in equation 2.9. An internal counter is used to synchronize all the operations and to reset the accumulation register of the DSP block between runs.

6.4.5 Top Level Layer and Network Architecture

Figure 6.6 shows the top level layer architecture that includes all these functional blocks, along with control logic to synchronise and configure the dataflow for a single layer. A complete NN is formed by stacking multiple of these according to the network structure as shown in Figure 6.7, which shows the arrangement, interconnect, and interaction of the FIFOs with various building blocks. Each FIFO stores only the data required from the previous LSTM iteration, since the stored data is consumed by various compute blocks in the subsequent iteration, alleviating the need to store redundant data from more than 1 iteration as happens in many architectures that time multiplex their compute units. The largest network the proposed approach can fully support is roughly estimated based on the number of layers, multiplied by the neurons in fully connected or number of units ×4 in LSTM layers, plus 2. This yields the number of required DSP blocks which should be less than what is available on the target device. Although, the proposed approach ideally targets lightweight LSTM networks that can be fully unrolled at the neuron level, where its efficiency is maximised, it is also versatile enough to be implemented as a single layer-time multiplexed implementation or even folding parallel compute units, trading off performance and resource utilisation to potentially adapt to larger networks.

The main control blocks in the top level layer architecture are the *Stall Component* and the *Control State Machine*. The *Stall Component* is configured with the number of weights, stall cycles needed, the type of layers, the number of iterations and whether to return the sequences in LSTMs. It generates the control signals required in the *Neural Network MAC*, for example, to enable the address counters or which input source to choose from. Meanwhile, the *Control State Machine* synchronizes the flow between the *Neural Network MAC* and the other blocks. The control unit used in this overlay is much more complex compared to the one in the previous chapter, thus implemented as a state machine, rather than in dedicated control logic.

A multiplexer after the *Neural Network MAC* selects which DSP block fires at each time step. Another at the output selects the appropriate datapath depending on whether it is an LSTM or fully connected layer. The *Neural Network MAC*'s serial operation, coupled with the ability to select the datapath according to the layer's configuration, through the use of multiplexers, forms an architecture that is able to adjust its latency and throughput for different neural network configurations. Although large scale multiplexers and decoders within the proposed architecture may cause delay, they can be pipelined according to the device's LUTs capabilities for high throughput at the cost of latency. Moreover, a FIFO is placed after each output multiplexer to gather the data required for the following layer.

The top level design that implements the whole NN is parametrised with the neural network configuration, i.e. number of layers, number of neurons in each layer, wordlengths, etc. This allows for a more flexible overlay that can also be ported to other devices that employ different DSP blocks. A specific overlay configuration can be used alongside a processor in an FPGA SoC. The processor can then configure the overlay at runtime with a specific network configuration. More importantly, weights and biases can also be set at runtime, enabling the overlay not only to adapt to weight updates and finetuning after deployment, but also to compute other LSTMs that fit within the bounds of the specified architecture. Underutilising the overlay does however incur a

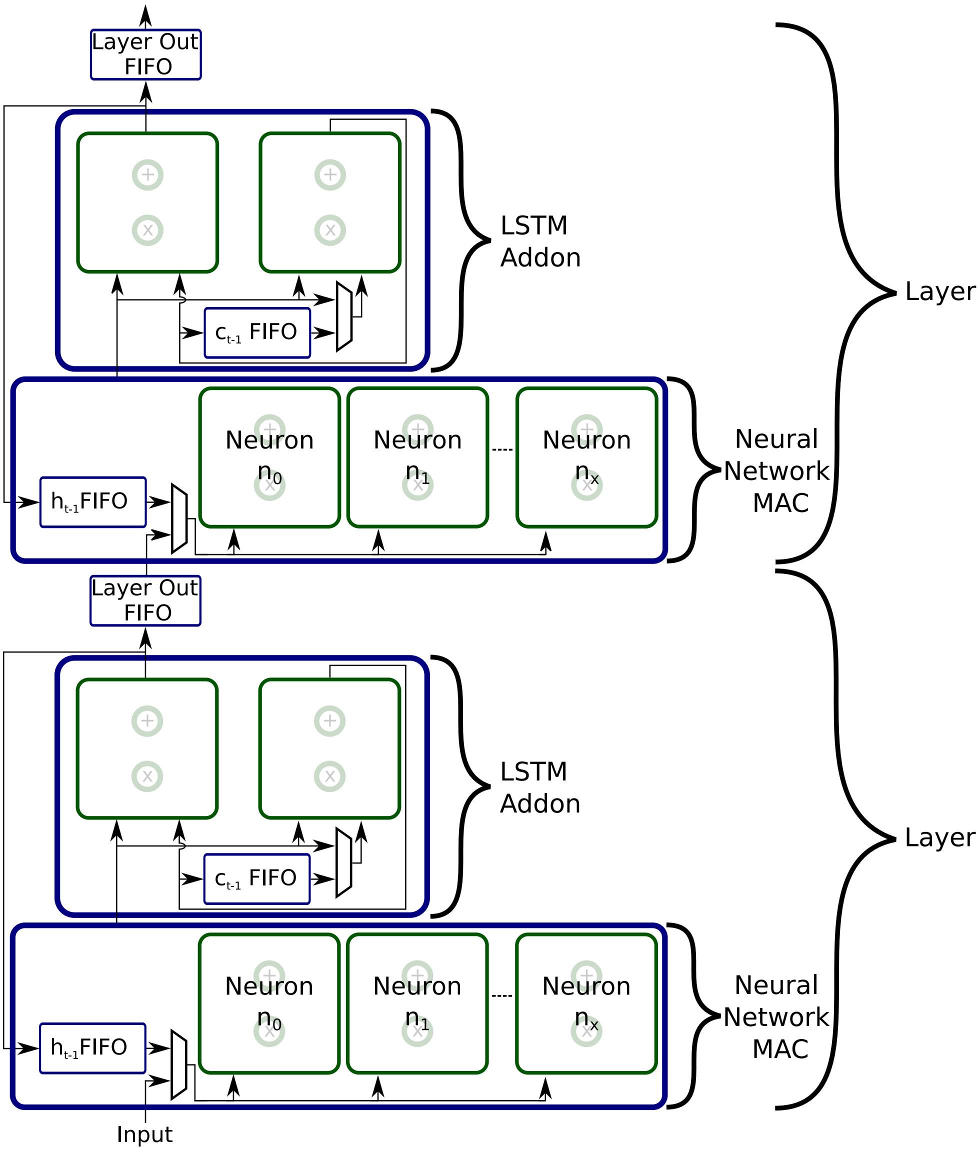


Figure 6.7: Top level Neural Network architecture.

performance overhead since data must still flow through unused layers.

6.5 Evaluation

6.5.1 Models for Evaluation

Two LSTM models are used to evaluate the proposed architecture. These models have been used previously in [53–55, 72], with which comparisons are made. While the exact code and framework were unable to be used, the models have been recreated to the best extent possible in Tensorflow v2.2 [56], according to the information provided in these references. The first model consists of a single LSTM layer with 16 units and a fully connected layer of 10 neurons. This network is trained as a classifier on the MNIST handwritten digit dataset, with the 10 output neurons corresponding to digits 0 to 9. The inputs to the LSTM model are 28×28 images with all pixel values normalised to a range from 0 to 1, this translates to 28 pixels being sent at a time for 28 iterations.

The second model is a character level LSTM trained on a part of Shakespeare's writing and comprises two LSTM layers, each with 128 units, and a fully connected layer with 65 neurons. The input to the LSTM is a vector with 65 one hot encoded values, each one representing a unique character that has been found in the text. A sequence of 50 of these vectors is passed to the LSTM model which generates the scores of the predicted 51st character at the output. An overlay architecture for each model has been created for direct comparison with previous work, although the MNIST model could be run on the larger character overlay.

6.5.2 Activation Function Impact

The proposed approximated activation functions are initially evaluated on how they impact the accuracy of these two models, as discussed for a different model in Section 6.4.3.

Trained with	**Sigmoid/tanh**		**Approx.**
Inf. using	Sig./Tanh	Approx.	Approx.
Train	0.0762/98.0%	0.984/80.7%	0.067/98.1%
Validation	0.108/97.3%	0.944/81.7%	0.109/97.4%
Test	0.106/97.5%	0.955/80.7%	0.117/97.4%

Table 6.4: Approximated functions loss/accuracy-MNIST.

The results are presented in Table 6.4 and 6.5. While there is an increase in loss, along with a decrease in accuracy for the MNIST network, when

Trained with	**Sigmoid/tanh**		**Approx.**
Inf. using	Sig./Tanh	Approx.	Approx.
Train	0.8733	2.9	1.29

Table 6.5: Approximated functions loss-Character level LSTM.

the activation functions are simply switched after training, it is shown to be alleviated by using the proposed activation functions during training. Although the loss in the character level LSTM can be considered high, this is believed to be due to its learning complexity. More specifically, even though a low loss would mean that a model can work more accurately in inference, in this case it would mean that the model has memorised the textbook, which is a very difficult task. Instead, the model is expected to learn the coarser text patterns, rather than the finer details, and generate similar text.

6.5.3 Compute Overlap

The dependence of LSTM layers on previous outputs usually means the next iteration in a layer cannot start until the previous iteration has completed, reducing the parallel processing efficiency and effective throughput. The dataflow used in the proposed LSTM architecture coupled with unrolling parallelism at each neuron, enables the overlay to overlap part of the computations between iterations. In addition, the serial computing in the LSTM architecture enables the propagation of any compute configurations at the initial layer to the following layers, e.g. stall cycles applied in the first layer affect when the next layer will initiate its computation and so on. The total latency of an LSTM layer in the proposed architecture is modelled in equation 6.1. Meanwhile, the *Neural Network MAC* can start the computation of the next iteration with part of the previously generated data. This is based on the principle that by the time that data is needed, it will have been generated. Thus the next iteration in an LSTM layer can be initiated as per equation 6.2, with the lowest margin in this equation being the $\#Units \times 4$.

$$LSTM\ latency = 17 + \#Inputs + \#Units + (\#Units \times 4) \tag{6.1}$$

$$Initiation\ Interval = LSTM\ Lat. - \#Inputs - \#Units + 1 \tag{6.2}$$

For the LSTM networks used, these values are shown in the first row in Table 6.6. The second row shows the compute overlap impact on the latency of a datapoint, which is a whole image for the MNIST LSTM and 50 iterations of characters for the character level LSTM. This is the clock cycle count for the input data to pass through all the iterations in all the layers of the network, including the fully connected layers, and to generate all the output results in simulation. Meanwhile, the third row shows the clock cycle count averaged for successive datapoints.

	Normal		**Overlapped**	
	MNIST	Char. LSTM	MNIST	Char. LSTM
II-1st layer	123	722	80	530
Clock Cycles	3503	37131	2342	27723

Table 6.6: Compute overlap when processing LSTMs.

The ability of the proposed architecture to overlap part of the computation within consecutive LSTM layer iterations, leads to a latency reduction of 33% for the MNIST-LSTM and **2**5% for the character level LSTM.

6.5.4 Weight Stationary Architecture

Previous work in neural networks has explored a wide spectrum of optimisations in an effort to reduce off-chip memory bandwidth, from computing parts of the neural network in batches and transferring weights accordingly, to extreme quantization of weights. In this architecture, the compute units, that typically consume most FPGA resources, are mapped to DSP blocks. This, coupled with the minimalistic approach in the design of other building blocks, results in releasing FPGA resources that can be used to store more weights on chip, maximising the device's storage capabilities. Table 6.7, shows the total weight and data sizes for a single classification for the two networks. The weights and biases in the MNIST LSTM amount to about 73% of total data and about 98% for the character level LSTM. Architectures that store these parameters in off-chip memory require high bandwidth to achieve high throughput, while the proposed approach reduces bandwidth requirements and potentially energy consumption.

	Number of coeff.		**Size in bits**	
	MNIST	Char. LSTM	MNIST	Char. LSTM
Weights (18 bits)	2976	238208	53568	4287744
Biases (16 bits)	74	1089	1184	17424
Inputs (27 bits)	784	3250	21168	87750
Coefficients to total data	-	-	**72.12%**	**98.00%**

Table 6.7: Weights to input size ratio.

	MNIST LSTM				Character LSTM			
	Baseline [72]	Appr. A [72]	Appr. B [72]	This work	[55]	Deepstore [54]	Appr. [53]	This work
FPGA	XC7Z020	XC7Z020	XC7Z020	XCZU7EV	XC7Z020	XC7Z045	XC7Z030	XCZU7EV
Precision	N/A	N/A	N/A	16-27 fixed	16 fixed	16 fixed	8 fixed	16-27 fixed
LUTs	7741 (14.55%)	9529 (17.91%)	6763 (12.71%)	4244(1.84%)	7201 (13.54%)	N/A	23036 (29.31%)	95263 (41.35s%)
Flip-Flops	2412 (2.27%)	8456 (7.95%)	5928 (5.57%)	9308(1.75%)	12960 (12.18%)	N/A	28481 (18.12%)	118261 (25.66%)
BRAM	3.58KB	0	0	0 (0%)	16 (11.43%)	N/A	180×36KB (67.92%)	518×18KB(83.01%)
DSP	1 (0.45%)	0 (0%)	0 (0%)	78(4.51%)	50 (22.73%)	N/A	0 (0%)	1095(63.37%)
Freq. (MHz)	N/A	100	100	640/160	142	142	100	420/105
DSP Max (MHz)	464	464	464	775	464	650	548-742	775
DSP Freq. %	N/A	21.6%	21.6%	82.6%	30.6%	21.8%	18.2-13.5%	54.1%
Power (w)	0.142[e]	0.072[e]	0.038[e]	0.679[e]	1.94	2.3	1.19[e]	8.531[e]
Latency (ms)	0.17[a,c]	18.58[a,c]	18.58[a,c]	0.0037[d]	0.932[a,c]	N/A	N/A	0.066[d]
T'put-LSTM (GOPs)	0.928	0.00847	0.00847	44.5	0.29	1.05	8.08(max)/2.26(avg)	363.7
T'put (class./s)	5882.4[b,c]	53.8[a,b]	53.8[a,b]	282186.9[d]	1073.0[a,b]	1969.0 [a,b]	N/A	15819.2[d]
T'put (GOPs)	N/A	N/A	N/A	44.6	N/A	N/A	N/A	363.9
Efficiency (GOPs/w)	6.54[a]	0.12[a]	0.22[a]	65.67[d]	0.15[a]	0.46[a]	6.79(max)/1.89(avg)[a]	42.7[d]

[a] LSTM layers only.

[b] Estimated from the reported average runtime.

[c] Reported as average runtime.

[d] Complete network (i.e. including FC layers).

[e] Vivado power estimator

Table 6.8: Resource utilisation and performance comparisons with same models.

	Single Instance [50] (Streaming)	6 Instances [50] (Streaming)	6 Instances [50] (Batch)	[51]	[63]	[63]	[76]
FPGA	XC7Z045	XC7Z045	XC7Z045	XCZU7EV	VX690T	XC7Z045	XC7Z020
Precision	5 fixed	5 fixed	5 fixed	1-8 fixed	16 fixed	16 fixed	16 fixed
LUTs	32815 (15%)	161574 (74%)	190036 (87%)	N/A	204000 (47.0%)	166000 (75.8%)	51604 (97%)
Flip-Flops	14532 (3%)	51213 (12%)	78516 (18%)	N/A	222000 (25.6%)	150000 (34.4%)	69160 (65%)
BRAM	83 (15%)	339 (62%)	498 (91%)	N/A	1070 (72.8%)	517.5 (94.9%)	179.2 (64%)
DSP	33 (4%)	195 (22%)	198 (22%)	N/A	2060 (57.0%)	900 (100%)	180.4 (82%)
Freq. (MHz)	166	142	142	266	125	142	150
DSP Max (MHz)	548-742	548-742	548-742	775	548-741	650	464
DSP Freq.%	30.3-22.4%	25.9-19.3%	25.9-19.3%	34.3%	22.8-16.9%	21.8	32.3%
T'put (GOPs)	152.16[b]	308.05[b]	693.12[b]	746-4201[b]	356	221	4.25
Power (W)	1.7 [a,b]	6.97[a,b]	12.46[a,b]	N/A	26.5	10.6	2.29
Efficiency (GOPs/w)	89.52[a,b]	44.22[a,b]	55.62[a,b]	N/A	13.48	20.84	1.86

[a] Calculated from the authors' work.

[b] Complete network (i.e. including FC layers).

Table 6.9: Resource utilisation and performance comparisons with different models.

6.5.5 Performance, Resource Utilisation, and Comparisons

Two versions of the proposed overlay architecture have been implemented, one for each NN, on a ZU7EV FPGA as found on the Xilinx Zynq Ultrascale+ ZCU104 board. Both overlays have been implemented in Verilog HDL using Xilinx Vivado 2018.2. Moreover, both have been integrated in an SoC implementation with the ARM Cortex A53 on the device and functionally verified, baremetal, on part of the datasets using Xilinx SDK. To enable the integration of the high operating frequency overlay in the SoC and overcome the lower operating frequency of required IPs, the proposed overlay employs a dual clock configuration. A high frequency clock is used for the overlay's compute mode, whereas a slower clock that operates at a quarter of the fast clock frequency is used to configure the overlay and to transfer the input and output data. To match the rate of the slow clock input data with the fast clock compute, 4 inputs are transferred at a time at the slow clock rate to a dual clock fifo, subsequently each input slot is extracted at the fast clock rate.

All results presented in this section are post-place and route for the overlay module only, extracted from the hierarchical results of the implemented SoC. Tables 6.8 and 6.9 compare the attributes of the proposed approach to previous work. To enable more objective comparisons with work targeting different FPGA devices, additional attributes are derived to encapsulate the overall efficiency. In addition, the theoretical maximum frequency of the DSP blocks is reported, for each of those devices, as found in the devices' datasheets [110, 121, 128, 129], and the frequencies achieved. In cases where the device's speed grade is not reported by the authors, a range of highest and lowest speed grades is used. Although the MNIST model can be computed within the larger overlay, an overly for each LSTM model is provided to enable objective comparisons with previous work, and to have a benchmark on how the proposed overlay scales. The MNIST overlay would consume 6.8% of the DSP blocks and about 1.24% of the implemented memories of the character level LSTM overlay.

In addition to the MNIST overlay reported in Table 6.8, which uses LUTRAMs for the neuron memories, the use of BRAMs in an identical architecture is also explored. Naturally this resulted in varying utilisation of memory elements on the FPGA, but importantly, this also resulted in a reduced frequency and higher power estimation for the BRAM based overlay. The BRAM based overlay is able to compute at 520MHz and configured at 130MHz, whereas the LUTRAM based overlay is able to compute at 640MHz and configured at 160 MHz. Meanwhile, the power estimation for the BRAM is higher, amounting to 0.845W compared to 0.679W the LUTRAM based overlay, which in turn results in poorer efficiency.

This suggests that the weight memories in lightweight and shallow neural networks are more performance and energy efficient when mapped to LUTRAMs, which can be partly due to the fact that only a small percentage of each BRAM bank is utilised. Meanwhile the simple routing of small FPGA fabric based memory offers better operating frequency. The configuration of the weights and biases takes place in streaming mode, using the slower clock, and is estimated to be around 0.02ms with negligible difference between the two overlays.

The proposed overlay for the character level LSTM model is implemented using a hybrid memory arrangement in the first two LSTM layers, e.g. one neuron uses LUTRAM memory, the other BRAM memory etc, while the output layer uses LUTRAM memories. A uniform memory overlay with either resources does not fit in the device when integrated with the SoC. Similarly to the MNIST overlay, the weights and biases are configured in streaming mode, using the slower clock, and this is estimated to take 2.29ms. Considering the BRAM based MNIST overlay as benchmark, we notice that the character level overlay scales well as the frequency is reduced by 19.2% while utilising 14× to 37× more resources.

Table 6.8 summarises other relevant previous work that implement the exact same LSTM models as in this work. The proposed MNIST overlay is competitive in terms of resource utilisation with the work in [72] which focuses on approximate computing to yield multiplierless-low power implementations, while significantly outperforming in terms of latency, throughput, and efficiency. In addition, the proposed architecture achieves 82.6% the theoretical DSP block maximum frequency, compared to 21.6% achieved in [72]. Regarding the more complex character level LSTM overlay, although it utilises more resources due to the higher neuron and layer parallelism and higher precision, the proposed implementation is significantly better in terms of latency, throughput, and efficiency. Specifically, it is 22.6× more efficient compared to the average performance of the most competitive previous work in [53]. Meanwhile, the proposed architecture operates at 54.1% the DSP block theoretical maximum frequency of the target FPGA, compared to 30.6% of the most competitive previous work in [55].

The comparisons with previous work are extended in the embedded domain in Table 6.9. Although these implementations do not target the same models, they use various other approaches of interest on more modern FPGA devices. The implementations in [50, 51] target a Bidirectional-LSTM model for optical character recognition. The complete network consists of a single Bidirectional LSTM layer with a total of 200 nodes followed by a fully connected layer, both of which have been implemented in the compute architecture. Compared to the proposed character level LSTM, this model is less complex in terms of LSTM cells used, number of layers, and precision, using 68.8% to 81.5%

reduced precision. Nonetheless, the proposed overlay architecture obtains better throughout compared to the single instance and 6 instances that operate in streaming mode. Although the single instance in [50] is more efficient, its performance does not scale well when 6 instances are implemented on the device, resulting in reduced efficiency which is slightly better than the proposed overlay. Its corresponding batch processing implementation with 6 instances yields improved throughput, compared to the streaming operation, while increasing its efficiency. This shows that streaming processing is more challenging to optimise, since it doesn't scale linearly with the increase of compute resources, whereas batch processing scales better with the availability of more input data. The authors further expanded their work in [51] for a more systematic exploration of the tradeoffs of reduced precision, improving their obtained throughput significantly. The work in [63] aims at partitioning large LSTM layers and achieves the obtained throughput efficiency with a batch size of 64, which the proposed overlay outperforms. Lastly, the authors in [76] target a single LSTM layer only, with a similar configuration to the first layer of the character level LSTM, while some of the computations are offloaded to the ARM core, obtaining lower performance and efficiency with, however, a less capable device. Regarding the operating frequencies of previous work in Table 6.8, the most competitive one operates at 32.3% of the device's DSP block theoretical maximum, while the least competitive overlay in this chapter operates at 54.1%. This demonstrates the frequency gains of the proposed approach, irrespective of the different FPGAs used in previous work.

Porting to Zynq 7000 series

	MNIST LSTM	**Character LSTM**
FPGA	XC7Z020	XC7Z100
Precision	16-27 fixed	16-27 fixed
LUTs	4120 (7.74%)	93920 (33.86%)
Flip-Flops	8972 (8.43%)	113838 (20.52%)
BRAM	0 (0%)	259 (34.30%)
DSP	78 (35.45%)	1095 (54.21%)
Freq. (MHz)	208	312
DSP Max (MHz)	464	650
DSP Freq.%	44.83%	48%

Table 6.10: Resource utilization and frequency on Zynq 7000 series.

The evaluation methodology is supplemented by porting the proposed overlays to Zynq 7000 devices for direct comparisons with previous work in Table 6.8. The supplementary implementation results are shown in Table 6.10,where it is demonstrated that the proposed overlays are effective on these devices as well. Specifically we see that the MNIST overlay achieves 208 MHz, which amounts to 44.83% of the device's theoretical maximum. Compared to previous work using MNIST in Table 6.8, the proposed overlay operates twice as fast on the same device. Furthermore, the character level overlay achieves 312 MHz out of the device's 650 MHz. This translates to 48% of the theoretical maximum of the XC7Z100 device and is approximately 2.2× to 3× faster compared to relevant previous pieces of work in Table 6.8.

6.6 Summary

This chapter presented a streaming overlay architecture based on DSP blocks that was able to compute lightweight to moderate sized LSTM and fully connected layers, with all required weights stored on chip. The proposed approach was aimed at enhancing programmability and flexibility with the overlay concept. The implemented overlay has been designed in architecture-centric manner and has therefore obtained high performance by virtue of high operating frequency, while it consumed its input data serially for better resource efficiency. Its serial data flow, parallel neuron computation, and pipelined operation, coupled with optimisations in compute overlap and on chip weight storage resulted in high throughput operation. The low level operation of the architecture was abstracted to form an overlay which can be configured at the top level with the model configurations. Specifically, the overlay Verilog code has been heavily parametrised at the top level to effortlessly form the configuration of the overlay, e.g. number of layers, neurons per layer. Underlying repetition loops instantiated and connected the smaller building blocks accordingly to construct the final overlay architecture. Moreover, the implemented overlay, was able to be configured after deployment, at runtime, with the different model configurations, weights, and biases. The extracted results have shown that the standalone overlay architecture operated at much higher frequencies in an SoC design, alongside the ARM core, within which it has been implemented and functionally verified. The proposed overlay architecture was also shown to be competitive in terms of efficiency and outperformed other generic previous work in streaming processing, while using higher precision.

Chapter 7

Conclusions and Future Work

The increasing ubiquity of Neural Network applications on edge devices has created the need to more efficiently execute this class of algorithms and reduce runtime within the limits of more constrained hardware. FPGAs' ability to accommodate custom computing architectures tailored to their available resources, coupled with their evolution over the years to provide more advanced functionality (e.g. DSP blocks, Arm cores), have rendered them ideal for edge computing applications. With neural networks still evolving in their application for different computing domains, there is a need to consider flexibility and generality in the implementation of architectures to accelerate them. The bulk of existing work on mapping neural networks to FPGAs can be split into two classes. The first implements general matrix computing hardware that can support acceleration of a variety of neural network workloads. These architectures require the streaming of input data and weights, and in some cases, multiple transfer to and from off-chip memory in intermediate stages of the neural network. Since they implement a generic computational architecture, they offer flexibility, but are not as efficient as more recent ASIC implementations of similar "tensor" processor cores. The second approach involves applying a series of optimisations to network structure and numerical representation to significantly reduce the cost of computation for a particular network at the cost of losing flexibility and requiring recompilation of the hardware for any parameter change. Various toolflows have been proposed that automate the generation of hardware for the first approach, by optimising the mix of matrix computation sizes to optimise off-chip memory access, or for the second by applying the required optimisations. Optimising an architecture for a specific model is problematic in an evolving application setting, or where there is a need to adapt a model to data after initial training, e.g. through federated learning. Furthermore, much of the existing work has not considered the FPGA architecture in detail and thus results in sub-optimal mappings that do not offer the throughput that should be achievable. This thesis has attempted to address this important area of flexible yet architecture-oriented neural network implementation on modern FPGAs.

This thesis has applied the concept of overlay architectures in the neural network domain to effectively enable an abstracted level of programmability on an FPGA. The implemented overlay offers a more rapid deployment and adaptation of NNs on FPGAs, while the long backend toolflow compilation is only required initially and is amortised over the long term use of the overlay. Therefore, compilation of NN models to the implemented overlay is vendor free and lightweight, and can potentially take place on the edge devices themselves. The work in this thesis demonstrated that careful consideration of the underlying FPGA architecture can yield overlays that operate at high operating frequencies, while retaining flexibility. Specifically, the careful use of DSP blocks, that are abundant in modern FPGA architectures, coupled with the more targetted use of FPGA resources to construct the datapath, have generated architectures that operate nearer to the theoretical maximum frequency of the FPGA compared to previous designs, even with complex networks. Dataflow in this overlay has been tailored towards a streaming flow as suited to sensor processing at the edge, while parallelism has been exploited at the neuron level, without the complete unrolling typical of matrix based computation approaches. This also enables the network parameters to be retained within the overlay, reducing significantly the required off-chip memory accesses. Rapid deployment with predictable performance are offered as a result, while maintaining generality in the supported NN computations, instead of optimising around a specific set of network parameters. Finally, the proposed approach is extensible to other layer types, which has been demonstrated in Chapter 6 by extending to LSTM networks.

7.1 Summary of Contributions

The original contributions of this thesis are summarised as follows:

7.1.1 Intrusion Detection System at Line Rate Detection

Chapter 3 demonstrated the use of NNs for network intrusion detection, in which a custom compute architecture has enabled line rate detection. During the training of the NN, categorical features were mapped to a one-hot encoded representation, mitigating the possible bias introduced by ad-hoc numerical mapping. The obtained accuracy of the proposed model has been extensively evaluated against previous work, showing the benefits of this approach. An accelerator design has been implemented using HLS which exploits parallelism in the proposed NN to offer high performance. Moreover, the accelerator offered some degree of flexibility by enabling weight updates dynamically, by an external source (e.g. Arm core). The accelerator has been integrated in an SoC implementation and functionally verified in practice. Comparisons with software equivalents and previous work showed that the approach in this chapter has contributed to achieving high performance and detection rate, providing line rate detection (1Gbps and 10Gbps).

7.1.2 2D Spatial Convolution Filters

A systematic exploration on the various design choices for implementing 2-D spatial convolution filters, based around FPGA DSP blocks, was explored in Chapter 4. The selected architecture was extensively scaled on a wide range of filter sizes and to modern image resolutions, thus explored how its underlying compute units utilised the FPGA resources. All explored architectures implement coefficient memories, that can be configured at runtime to different filter values, instead of considering fixed kernel coefficients and optimising the architecture around them. HLS equivalents were also implemented to draw comparisons, quantifying the performance and resource utilisation benefits of the proposed approach. In addition, thorough comparisons with previous work has shown that building such designs around modern FPGA DSP blocks, in an architecture-centric manner, offers significant improvements compared to more generic methods and tools.

7.1.3 Streaming Overlay Architecture for NN Computation Based on FPGA DSP Blocks

An overlay architecture for NNs was presented in Chapter 5, built around the capabilities of FPGA DSP blocks. The overlay architecture abstracted the FPGA fabric to a coarser grained architecture that mimicked the structure of NNs. The proposed overlay was tailored for streaming NN dataflow and serial processing, which rendered it ideal for edge computing. Various comparisons were made that motivate the design choices and unrolling factors in the overlay building blocks. Each neuron was considered as a serial execution unit, and neuron-level-parallelism has been exploited while full unrolling of the computations was not required. The overlay architecture achieved near the theoretical maximum frequency of the FPGA DSP blocks, although it has been tested in an SoC design in practice at a baseline frequency due to limitations from various interfacing IP blocks. Comparisons with software equivalents and previous work in HLS showed that the proposed overlay performed better, while maintaining domain specific flexibility.

7.1.4 Streaming LSTM overlay architecture

While the proposed overlay in Chapter 5 achieved high frequency, and as a result, improved performance compared to previous work, it was not shown to scale. The work in Chapter 6 expanded the initial overlay to support a more complex class of NNs, specifically LSTMs, in addition to fully connected layers. LSTMs include dependencies on previous outputs along with more complex activation functions which made the datapath, control, and compute units more complex. The overlay in this chapter was shown to scale at a low frequency overhead, while the various obstacles hindering the SoC integration at a high operating frequency have been overcome.The proposed overlay is evaluated on various metrics throughout Chapter 6, while thorough comparisons with previous work highlight the advantages of the underlying architecture-centric architecture. This chapter showed that the neuron-centric parallelism of this overlay made extensions to complex layer types possible and that these do not heavily compromise performance.

7.2 Future Work

The results presented throughout this thesis demonstrate the potential of the proposed overlay for further future work. The future work can optimise the proposed work in various aspects, as described in the following subsections.

7.2.1 Overlay Generation Framework

Although the configuration of the overlay architecture is at a high level, e.g. setting the parameters at the top level module, it still requires human intervention to do so. Although, this seems a relevantly a straightforward task, the human interaction can be problematic to users that are not familiar with the backend interface and toolflow. Moreover, there's still a gap between the training of the models, or analysing trained ones, and extract their weights and overlay configuration to the FPGA tool for the bitstream generation. Adding to the overall narrow accessibility to only a certain group of familiar users. Thus, an automatic overlay generation framework, that would bridge the two operations, would make the process of creating an overlay more abstracted, faster and accessible to more users. Additionally, the functionality of the framework itself could be further expanded to generate overlay configurations according to high level user constraints, e.g. specific throughput, latency or resource utilisation. The latter, however, would require to accurately model the capabilities of the overlay architecture in order to generate the corresponding low level configurations. Lastly, a lighter version of the framework that would only operate on trained models would also be useful on devices at the edge. This will enable for rapid deployment of accelerated neural networks on FPGAs at the edge, in a context where compilation can take place on the resource constrained platforms themselves, independently of the vendor backend toolflow.

7.2.2 Reduced Precision

The work in this thesis maintains relatively high precision for the task at hand. In Chapter 2, 32-bit floating point arithmetic was used, the 2-D spatial filters in Chapter 3 used 8 bit, while the overlays in Chapters 4 and 5 were tailored to the DSP blocks' wordlength, that span over 16 to 48 bit arithmetic. Previous work on neural networks has shown that even extreme reduced precision to 1 bit can generate a network with tolerable accuracy overhead. Therefore, there is potential in exploring reduced precision overlays, at a wordlength that would maintain satisfactory accuracy in a wide range of neural networks. Consequently, the computations within the overlay could be more efficiently mapped to the available DSP blocks through Single Instruction Multiple Data (SIMD) or Very Long Instruction Word (VLIW) configurations.

7.2.3 Efficient mapping and scheduling of irregular neural network workloads

Although pruning is an effective optimisation, as described in Chapter 2, it is not straightforward to exploit its computational benefits due to its irregular structure. Therefore, compute architectures that operate in a more regular manner either do not skip these computations at all, or may require a certain degree of sparsity in the network to amortise any overheads that may occur. A worthwhile expansion to the work in this thesis would therefore be additional logic and datapath that would efficiently map these irregular patterns to take advantage of both, memory and workload reduction. The proposed overlay would be a very interesting approach in this context since its computations haven't been fully unrolled. Therefore, it maintains some degree of flexibility through its serial element, the programmability of which can form the basis of a mechanism that would skip the pruned computations.

7.2.4 Support more layer types

Based on Chapters 3, 5 and 6, another worthwhile direction to explore is the support of more layers and emerging topologies. To start with, CNN layer support would expand the application domain of the work in this thesis to computer vision. The used streaming dataflow would therefore be evaluated in domains like robotics, in which real time response is required, instead of throughput and batch processing. Moreover, efficient support for emerging NN topologies, for example transformer networks or the idea of skipping layers with residual connections, would also extend this work to emerging concepts in this domain. As a result, further exploration can be carried out on how the proposed overlay can implement these functionalities and what would be the overhead to the performance and resource utilisation.

7.2.5 Support deep networks

Although the main target of this thesis are rather small networks that act as an event detection, with further action to be taken in the cloud or on device, supporting very deep networks would make the work in this thesis more comprehensive. Although it would be very difficult to maintain all the benefits of the approach described in this thesis, due to overheads introduced, simple support without significant requirements would also add to the functionality of the overlay. Potentially, the support can be software based, to explore the partitioning of the deep network to an overlay configuration. The tool would then define how these partitions would be scheduled and execute them. In addition, an efficiency exploration could also take place in order to derive an efficient scheduling scheme with very small batch size, as a compromise between latency and configuration overhead.

7.3 Summary

This thesis has explored the careful consideration of the underlying FPGA architecture in order to yield neural network implementations that operate at high operating frequencies. It has shown that an architecture centric approach, coupled with domain specific optimisations, can generate high performance implementations while maintaining a degree of flexibility. Specifically, DSP block based implementations were demonstrated to achieve high operating frequency in complex and demanding workloads, in addition to their dynamic programmability which offers a degree of flexibility for reconfiguring network parameters. The latter has been achieved by tailoring the implemented designs for SoC integration, therefore using the software programmable ARM core to reconfigure the overlay. This thesis has also shown that the optimal use of DSP blocks does not require detailed low-level HDL design for each individual workload, but that an architecture centric overlay can scale and provide this high performance in a flexible architecture that can be adapted and tuned without loss of performance. The contributions of this thesis have been extensively evaluated and compared, across a variety of attributes, against software and HLS equivalents, in addition to previous work in the literature. These comparisons have helped shape the final conclusions of this thesis along with future work described in this chapter.

Bibliography

[1] L. Ioannou and S. A. Fahmy. Network Intrusion Detection Using Neural Networks on FPGA SoCs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 232–238, 2019.

[2] L. Ioannou and S. A. Fahmy. Neural Network Overlay Using FPGA DSP Blocks. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 252–253, 2019.

[3] L. Ioannou and S. A. Fahmy. Lightweight Programmable DSP Block Overlay for Streaming Neural Network Acceleration. In *Proceedings of the International Conference on Field-Programmable Technology (ICFPT)*, pages 355–358, 2019.

[4] L. Ioannou, A. Al-Dujaili, and S. A. Fahmy. High Throughput Spatial Convolution Filters on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(6):1392–1402, 2020.

[5] L. Ioannou and S. A. Fahmy. Streaming Overlay Architecture for Lightweight LSTM Computation on FPGA SoCs. *Submitted to: ACM Trans. Reconfigurable Technol. Syst.*

[6] Elike Hodo, Xavier J. A. Bellekens, Andrew Hamilton, Christos Tachtatzis, and Robert C. Atkinson. Shallow and deep networks intrusion detection system: A taxonomy and survey. *CoRR*, 2017.

[7] Oğuz Karan, Canan Bayraktar, Haluk Gümüşkaya, and Bekir Karlık. Diagnosing diabetes using neural networks on small mobile devices. *Expert Systems with Applications*, 39(1):54–60, 2012.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, 2017.

[9] Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Steven Bohez, Sam Leroux, and Pieter Simoens. DIANNE: Distributed artificial neural

networks for the internet of things. In *Proceedings of the 2nd Workshop on Middleware for Context-Aware Applications in the IoT*, pages 19–24, New York, NY, USA, 2015. ACM.

[10] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, 2014.

[11] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. Cambricon: An instruction set architecture for neural networks. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 393–405, 2016.

[12] S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3): 880–888, 2007.

[13] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, 2016.

[14] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 161–170, New York, NY, USA, 2015. ACM.

[15] Xilinx. *UG579 UltraScale Architecture DSP Slice-User Guide*, V1.28, March 2020.

[16] E. Wu, X. Zhang, D. Berman, and I. Cho. A high-throughput reconfigurable processing array for neural networks. In *Proc. Field Programmable Logic and Applications (FPL)*, 2017.

[17] N. Shone, T. N. Ngoc, V. D. Phai, and Q. Shi. A deep learning approach to network intrusion detection. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(1):41–50, 2018.

[18] E. Hodo, X. Bellekens, A. Hamilton, P. L. Dubouilh, E. Iorkyase, C. Tachtatzis, and R. Atkinson. Threat analysis of iot networks using artificial neural network intrusion detection system. In *Proc. International Symposium on Networks, Computers and Communications*, 2016.

[19] Mohamed Idhammad, Karim Afdel, and Mustapha Belouch. DoS detection method based on artificial neural networks. *International Journal of Advanced Computer Science and Applications*, 8(4), 2017.

[20] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho. Deep learning approach for network intrusion detection in software defined networking. In *Proc. International Conference on Wireless Networks and Mobile Communications*, pages 258–263, 2016.

[21] T Tang, SAR Zaidi, D McLernon, L Mhamdi, and M Ghogho. Deep recurrent neural network for intrusion detection in SDN-based networks. In *Proc. International Conference on Network Softwarization*, 2018.

[22] C. Yin, Y. Zhu, J. Fei, and X. He. A deep learning approach for intrusion detection using recurrent neural networks. *IEEE Access*, 5:21954–21961, 2017.

[23] Zhipeng Li, Zheng Qin, Kai Huang, Xiao Yang, and Shuxiong Ye. Intrusion detection using convolutional neural networks for representation learning. In *Proc. International Conference Neural Information Processing*, pages 858–866, 2017.

[24] P. Bougas, P. Kalivas, A. Tsirikos, and K.Z. Pekmestzi. Pipelined array-based FIR filter folding. *IEEE Transactions on Circuits and Systems I: Regular Papers,*, 52(1):108–118, 2005.

[25] Jongsun Park, K. Muhammad, and K. Roy. High-performance FIR filter design based on sharing multiplication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(2):244 –253, 2003.

[26] Baptiste Wicht. *Deep Learning feature Extraction for Image Processing.* PhD thesis, 2018.

[27] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of FPGA, GPU and CPU in image processing. In *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 126–131, 2009.

[28] Danny Crookes. Architectures for high performance image processing: The future. *Journal of Systems Architecture*, 45:739–748, 1999.

[29] F. Schwiegelshohn, L. Gierke, and M. Hübner. FPGA based traffic sign detection for automotive camera systems. In *International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2015.

[30] Shang Wang, Chen Zhang, Yuanchao Shu, and Yunxin Liu. Live video analytics with FPGA-based smart cameras. In *Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pages 9–14, 2019.

[31] S. A. Fahmy, P. Y. K. Cheung, and W. Luk. High-throughput one-dimensional median and weighted median filters on FPGA. In *IET Computers and Digital Techniques*, volume 3, pages 384–394, 2009.

[32] Anna Gabiger-Rose, Matthias Kube, Robert Weigel, and Richard Rose. An FPGA-based fully synchronized design of a bilateral filter for real-time image denoising. *IEEE Transactions on Industrial Electronics*, 61(8): 4093–4104, 2013.

[33] Shiann-Shiun Jeng, Hsing-Chen Lin, and Shu-Ming Chang. FPGA implementation of FIR filter using m-bit parallel distributed arithmetic. In *IEEE International Symposium on Circuits and Systems*, 2006.

[34] F. Javier Toledo-Moreo, J. Javier Martínez-Alvarez, Javier Garrigós-Guerrero, and J. Manuel Ferrández-Vicente. FPGA-based architecture for the real-time computation of 2-D convolution with large kernel size. *Journal of Systems Architecture*, 58(8):277–285, 2012.

[35] Gian Domenico Licciardo, Carmine Cappetta, and Luigi Di Benedetto. Design of a Convolutional Two-Dimensional Filter in FPGA for Image Processing Applications. *Computers*, 6(2), 2017.

[36] S. Mirzaei, A. Hosangadi, and R. Kastner. FPGA implementation of high speed FIR filters using add and shift method. In *International Conference on Computer Design*, pages 308–313, 2006.

[37] Bradley McDanel, Sai Qian Zhang, H. T. Kung, and Xin Dong. Full-stack optimization for accelerating CNNs using powers-of-two weights with FPGA validation. In *ACM International Conference on Supercomputing*, pages 449–460, 2019.

[38] Z. Ma, Y. Yang, Y. Liu, and A. A. Bharath. Recurrently decomposable 2-D convolvers for FPGA-based digital image processing. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 63(10):979–983, 2016.

[39] S. Shreejith, B. Anshuman, and S. A. Fahmy. Accelerated artificial neural networks on FPGA for fault detection in automotive systems. In *Proc. Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 37–42, 2016.

[40] B. Ingre and A. Yadav. Performance analysis of NSL-KDD dataset using ANN. In *Proc. International Conference on Signal Processing and Communication Engineering Systems*, pages 92–96, 2015.

[41] B. Subba, S. Biswas, and S. Karmakar. A neural network based system for intrusion detection and attack classification. In *Proc. National Conference on Communication*, 2016.

[42] Apple. Hey Siri: An On-device DNN-powered Voice Trigger for Apple's Personal Assistant, 2017. URL `https://machinelearning.apple.com/research/hey-siri`.

[43] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent advances in convolutional neural networks. *Pattern Recogn.*, 77(C):354–377, 2018.

[44] Maurice Peemen, Bart Mesman, and Henk Corporaal. Efficiency optimization of trainable feature extractors for a consumer platform. In *Proceedings of the 13th International Conference on Advanced Concepts for Intelligent Vision Systems*, ACIVS'11, pages 293–304, 2011.

[45] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015*.

[46] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[47] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions. *ACM Computing Surveys*, 51(3):56:1–56:39, 2018.

[48] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar, R. Krishnamurthy, S. Gribok, B. Pasca, M. Langhammer, D. Marr, and A. Dasu. Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 199–207, 2019.

[49] Xiangyun Qing and Yugang Niu. Hourly day-ahead solar irradiance prediction using weather forecasts by LSTM. *Energy*, 148:461–468, 2018.

[50] Vladimir Rybalkin, Norbert Wehn, Mohammad Reza Yousefi, and Didier Stricker. Hardware architecture of bidirectional long short-term memory neural network for optical character recognition. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1390–1395, 2017.

[51] Vladimir Rybalkin, Alessandro Pappalardo, Muhammad Mohsin Ghaffar, Giulio Gambardella, Norbert Wehn, and Michaela Blott. Finn-l: Library extensions and design trade-off analysis for variable precision lstm networks on fpgas. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 89–897, 2018.

[52] Minjae Lee, Kyuyeon Hwang, Jinhwan Park, Sungwook Choi, Sungho Shin, and Wonyong Sung. FPGA-Based Low-Power Speech Recognition with Recurrent Neural Networks. In *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 230–235, 2016.

[53] E. Azari and S. Vrudhula. An Energy-Efficient Reconfigurable LSTM Accelerator for Natural Language Processing. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4450–4459, 2019.

[54] A. X. M. Chang and E. Culurciello. Hardware accelerators for recurrent neural networks on FPGA. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.

[55] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. Recurrent Neural Networks Hardware Implementation on FPGA. *CoRR*, 2015. URL `http://arxiv.org/abs/1511.05552`.

[56] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`.

[57] François Chollet et al. Keras. `https://keras.io`, 2015.

[58] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, page 675–678, New York, NY, USA, 2014. Association for Computing Machinery.

[59] The Theano Development Team et al. Theano: A python framework for fast computation of mathematical expressions, 2016.

[60] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the network be the AI accelerator? In *Proc. Morning Workshop on In-Network Computing*, NetCompute '18, pages 20–25, New York, NY, USA, 2018. ACM.

[61] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner. Survey and Benchmarking of Machine Learning Accelerators. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2019.

[62] A. Skillman and T. Edsö. A Technical Overview of Cortex-M55 and Ethos-U55: Arm's Most Capable Processors for Endpoint AI. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–20, 2020.

[63] Zhiqiang Que, Yongxin Zhu, Hongxiang Fan, Jiuxi Meng, Xinyu Niu, and Wayne Luk. Mapping Large LSTMs to FPGAs with Weight Reuse. *J. Signal Process. Syst.*, 92(9):965–979, 2020.

[64] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, 2017.

[65] R. Kastner, J. Matai, and S. Neuendorffer. Parallel Programming for FPGAs. *ArXiv e-prints*, 2018.

[66] S. Yusuf, W. Luk, M. K. N. Szeto, and W. Osborne. Unite: Uniform hardware-based network intrusion detection engine. In *Reconfigurable Computing: Architectures and Applications*, pages 389–400, 2006.

[67] R. Proudfoot, K. Kent, E. Aubanel, and N. Chen. Flexible software-hardware network intrusion detection system. In *Proc. International Symposium on Rapid System Prototyping*, pages 182–188, 2008.

[68] Andre Luiz Pereira de Franca, Ricardo Pereira Jasinski, Volnei Antonio Pedroni, and Altair Olivo Santin. Moving network protection from software to hardware: An energy efficiency analysis. In *Proc. IEEE Computer Society Symposium on VLSI*, pages 456–461, 2014.

[69] Andre Luiz Pereira de Franca, Ricardo P. Jasinski, Paulo Cemin, Volnei A. Pedroni, and Altair Olivo Santin. The energy cost of network security: A hardware vs. software comparison. In *Proc. International Symposium on Circuits and Systems (ISCAS)*, pages 81–84, 2015.

[70] S. Shreejith and S. A. Fahmy. Security aware network controllers for next generation automotive embedded systems. In *Proc. Design Automation Conference (DAC)*, 2015.

[71] Xilinx. *7 Series DSP48E1 Slice User Guide (UG479)*, V1.10, March 2018.

[72] G. Maor, X. Zeng, Z. Wang, and Y. Hu. An FPGA Implementation of Stochastic Computing-Based LSTM. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 38–46, 2019.

[73] X. Zhai, A. A. S. Ali, A. Amira, and F. Bensaali. MLP neural network based gas classification system on Zynq SoC. *IEEE Access*, 4:8138–8146, 2016.

[74] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 1135–1143, Cambridge, MA, USA, 2015. MIT Press.

[75] Thomas Hartenstein, Daniel Maier, Biagio Cosenza, and Ben Juurlink. Weight Pruning for Deep Neural Networks on GPUs. *PARS-Mitteilungen*, 35(1), 2020.

[76] Xinyi Zhang, Weiwen Jiang, and Jingtong Hu. Achieving Full Parallelism in LSTM via a Unified Accelerator Design. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 469–477, 2020.

[77] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Des. Autom. Embed. Syst.*, 16(3):31–51, 2012.

[78] T'etrault Marc-Andr'e. Two FPGA Case Studies Comparing High Level Synthesis and Manual HDL for HEP applications. *arXiv: Instrumentation and Detectors*, 2018.

[79] Jain, Abhishek Kumar and Maskell, Douglas L. and Fahmy, Suhaib A. Are Coarse-Grained Overlays Ready for General Purpose Application Acceleration on FPGAs? In *2016 IEEE 14th Intl Conf on Pervasive Intelligence and Computing (PiCom)*, pages 586–593, 2016.

[80] Abhishek Kumar Jain, Douglas L. Maskell, and Suhaib A. Fahmy. Coarse grained fpga overlay for rapid just-in-time accelerator compilation. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1478–1490, 2022.

[81] J. Coole and G. Stitt. Adjustable-cost overlays for runtime compilation. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 21–24, 2015.

[82] Stylianos I. Venieris and Christos-Savvas Bouganis. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47, 2016.

[83] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 2018.

[84] Vinayak Gokhale, Aliasger Zaidy, Andre Xian Ming Chang, and Eugenio Culurciello. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.

[85] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 65–74, New York, NY, USA, 2017. Association for Computing Machinery.

[86] Erwei Wang et al. Deep neural network approximation for custom hardware: Where we've been, where we're going. *ACM Computing Surveys*, 52(2):40:1–40:39, 2019.

[87] R. Abdulhammed, M. Faezipour, and K. M. Elleithy. Network intrusion detection using hardware techniques: A review. In *Proc. IEEE Long Island Systems, Applications and Technology Conference*, 2016.

[88] A. Gharib, I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani. An evaluation framework for intrusion detection dataset. In *Proc. International Conference on Information Science and Security*, 2016.

[89] A. Das, D. Nguyen, J. Zambreno, G. Memik, and A. Choudhary. An FPGA-based network intrusion detection architecture. *IEEE Transactions on Information Forensics and Security*, 3(1):118–132, 2008.

[90] Kwangjo Kim, Muhamad Erza Aminanto, and Harry Chandra Tanuwidjaja. *Network Intrusion Detection using Deep Learning: A Feature Learning Approach*, pages 35–45. Springer Singapore, 2018.

[91] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A. Ghorbani. A detailed analysis of the KDD CUP 99 data set. In *Proc. IEEE International Conference on Computational Intelligence for Security and Defense Applications*, pages 53–58, 2009.

[92] Karan Bajaj and Amit Arora. Improving the intrusion detection using discriminative machine learning approach and improve the time complexity by data mining feature selection methods. *International Journal of Computer Applications*, 76(1):5–11, 2013.

[93] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105 (12):2295–2329, 2017.

[94] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [DL] A survey of FPGA-based neural network inference accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 12(1):2:1–2:26, 2019.

[95] Kizheppatt Vipin and Suhaib A Fahmy. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys*, 51(4):72, 2018.

[96] K. Vipin and S. A. Fahmy. ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq. *IEEE Embedded Systems Letters*, 6(3): 41–44, 2014.

[97] Jon Postel. Internet protocol. *RFC*, 791:1–51, 1981.

[98] Donald G. Bailey. *Design for Embedded Image Processing on FPGAs*. Singapore : John Wiley & Sons (Asia), 2011. ISBN 9780470828502.

[99] Abdullah Al-Dujaili and Suhaib A. Fahmy. High throughput 2d spatial image filters on fpgas, 2017.

[100] Donald G. Bailey and Anoop S. Ambikumar. Border handling for 2D transpose filter structures on an FPGA. *Journal of Imaging*, 4(12), 2018.

[101] Zhangwen Tang, Jie Zhang, and Hao Min. A high-speed, programmable, CSD coefficient FIR filter. *IEEE Transactions on Consumer Electronics*, 48(4):834–837, 2002.

[102] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell. The iDEA DSP block-based soft processor for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):19:1–19:23, 2014.

[103] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy. DeCO: a DSP block based FPGA accelerator overlay with low overhead interconnect. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, 2016.

[104] B. Ronak and S. A. Fahmy. Mapping for maximum performance on FPGA DSP blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(4):573–585, 2016.

[105] B. Ronak and S. A. Fahmy. Multipumping flexible DSP blocks for resource reduction on Xilinx FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(9):1471–1482, 2017.

[106] *HDL coding practices to accelerate design performance. Xilinx White Paper (WP231)*, 2006.

[107] D.G. Bailey. Image border management for FPGA based filters. In *IEEE International Symposium on Electronic Design, Test and Application (DELTA)*, pages 144 –149, 2011.

[108] A. Benkrid, K. Benkrid, and D. Crookes. A novel FIR filter architecture for efficient signal boundary handling on Xilinx Virtex FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 273–275, 2003.

[109] Chang Choo and Punam Verma. A real-time bit-serial rank filter implementation using Xilinx FPGA. In *Proc SPIE. 6811, Real-Time Image Processing*, 2008.

[110] *Virtex-7 T and XT FPGAs Data Sheet: DC and AC Switching Characteristics (DS183)*, V1.28, March 2019.

[111] Susana Ortega-Cisneros, Miguel A. Carrazco-Díaz, Adrian Pedroza de-la Crúz, Juan J. Raygoza-Panduro, Federico Sandoval-Ibarra, and Jorge Rivera-Domínguez. Real time hardware accelerator for image filtering. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pages 80–87. Springer International Publishing, 2014.

[112] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf. A massively parallel coprocessor for convolutional neural networks. In *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 53–60, 2009.

[113] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. CNP: An FPGA-based processor for convolutional networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 32–37, 2009.

[114] Francisco Fons, Mariano Fons, and Enrique Cantó. Run-time self-reconfigurable 2D convolver for adaptive image processing. *Microelectronics Journal*, 42(1):204–217, 2011.

[115] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 47–56, 2012.

[116] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and Tell: Lessons Learned from the 2015 MSCOCO Image Captioning Challenge. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):652–663, 2017.

[117] A. Samajdar, T. Garg, T. Krishna, and N. Kapre. Scaling the Cascades: Interconnect-Aware FPGA Implementation of Machine Learning Problems. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 342–349, 2019.

[118] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[119] J. Chen and X. Ran. Deep Learning With Edge Computing: A Review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[120] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems 29*, pages 4107–4115. Curran Associates, Inc., 2016.

[121] *Zynq Ultrascale+ MPSoC Data Sheet: DC and AC Swithcing Characteristics (DS925)*, V1.16, July 2019.

[122] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.*

[123] M. Milenkoski, K. Trivodaliev, S. Kalajdziski, M. Jovanov, and B. R. Stojkoska. Real time human activity recognition on smartphones using

LSTM networks. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1126–1131, 2018.

[124] J. Kim, J. Kim, H. L. Thi Thu, and H. Kim. Long Short Term Memory Recurrent Neural Network Classifier for Intrusion Detection. In *2016 International Conference on Platform Technology and Service (PlatCon)*, pages 1–5, 2016.

[125] Xie Zhen-zhen and Zhang Su-yu. A Non-linear Approximation of the Sigmoid Function Based FPGA. In Liangzhong Jiang, editor, *Proceedings of the 2011, International Conference on Informatics, Cybernetics, and Computer Engineering (ICCE2011) November 19–20, 2011, Melbourne, Australia*, pages 125–132, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[126] B. Pasca and M. Langhammer. Activation Function Architectures for FPGAs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 43–437, 2018.

[127] Francois Chollet. *Deep Learning with Python*. Manning, 2017. ISBN 9781617294433.

[128] *Zynq-7000 SoC (Z-7007S, Z-7012S, Z-7014S, Z-7010, Z-7015, and Z-7020) (DS187)*, V1.21, December 2020.

[129] *Zynq-7000 SoC (Z-7030, Z-7035, Z-7045, and Z-7100): DC and AC Switching Characteristics (DS191)*, V1.18.1, July 2018.