FPGA Architectures for Low Precision Machine Learning

Duncan J.M. Moss

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

Faculty of Engineering & Information Technologies

University of Sydney

May 2018

Abstract

Machine learning is fast becoming a cornerstone in many data analytic, image processing and scientific computing applications. Depending on the deployment scale, these tasks can either be performed on embedded devices, or larger cloud computing platforms. However, one key trend is an exponential increase in the required compute power as data is collected and processed at a previously unprecedented scale. In an effort to reduce the computational complexity there has been significant work on reduced precision representations. Unlike Central Processing Units, Graphical Processing Units and Applications Specific Integrated Circuits which have fixed datapaths, Field Programmable Gate Arrays (FPGA) are flexible and uniquely positioned to take advantage of reduced precision representations.

This thesis presents FPGA architectures for low precision machine learning algorithms, considering three distinct levels: the application, the framework and the operator. Firstly, a spectral anomaly detection application is presented, designed for low latency and real-time processing of radio signals. Two types of detector are explored, a neural network autoencoder and least squares bitmap detector. Secondly, a generalised matrix multiplication framework for the Intel HARPv2 is outlined. The framework was designed specifically for machine learning applications; containing runtime configurable optimisations for reduced precision deep learning. Finally, a new machine learning specific operator is presented. A bit-dependent multiplication algorithm designed to conditionally add only the relevant parts of the operands and arbitrarily skip over redundant computation.

Demonstrating optimisations on all three levels; the application, the framework and the operator, illustrates that FPGAs can achieve state-of-the-art performance in important machine learning workloads where high performance is critical; while simultaneously reducing implementation complexity.

Statement of Originality

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and all sources have been acknowledged, specifically:

- The research direction has been provided by Professor Philip H.W. Leong.
- The idea of using an autoencoder for radio-frequency anomaly detection was originally conceived in a discussion with Professor Philip H.W. Leong and Dr David Boland.
- The on-line formulation of the DFT for data sampled at irregularly spaced time intervals was originally provided by Mr Zhe Zhang.
- The matrix multiplication hardware template was inspired by a floating-point version created by Mr Srivatsan Krishnan (Intel Corporation), directed by Mr Suchit Subhaschandra (Intel Corporation).
- The matrix multiplication results for Heterogeneous Load Balancing and Dynamic Dot Product were obtained with help from Mr Srivatsan Krishnan and Dr Eriko Nurvitadhi (Intel Corporation) respectively.
- The idea of the two speed multiplication algorithm was originally conceived in a discussion with Professor Philip H.W. Leong and Dr David Boland.

Publications

The work presented in this thesis has be published in a number of journals and conferences:

Journal Publications:

- **Duncan J.M. Moss**, David Boland, and Philip H.W. Leong. "A two speed serial-parallel multiplier". *In IEEE Transactions on VLSI Systems*, 2018. *under review.*

Conference Publications:

- **Duncan J.M. Moss**, David Boland, Peyam Pourbeik, and Philip H.W. Leong. "Real-time FPGA-based Anomaly Detection for Radio Frequency Signals". *In Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018. *to appear.*
- **Duncan J.M. Moss**, Zhe Zhang, Nicholas J. Fraser, and Philip H.W. Leong. "An FPGA-based Spectral Anomaly Detection System (with errata)". *In Proc. International Conference on Field Programmable Technology (FPT)*, pages 175–182, 2014. doi: 10.1109/FPT.2014.7082772.
- **Duncan J.M. Moss**, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. "A Customizable Matrix Multiplication Framework for the Intel HARPv2 Platform". *In Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 107-116, 2018.
- **Duncan J.M. Moss**, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. "High Performance Binary Neural Networks on the Xeon+FPGA Platform". *In 2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2017. doi: 10.23919/FPL.2017.8056823.

- Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, **Duncan Moss**, Suchit Subhaschandra, and Guy Boudoukh. "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?". *In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17*, pages 5–14, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021740.

Other published work:

Journal Publications:

- Jia-Chen Hua, Farzad Noorian, **Duncan Moss**, Philip H.W. Leong, and Gemunu H. Gunaratne. "High-dimensional Time-series Prediction using Kernel-based Koopman Mode Regression". *Nonlinear Dynamics*, 2017. In press, accepted 22nd August 2017.

Conference Publications:

- Stephen Tridgell, **Duncan J.M. Moss**, Nicholas J. Fraser, and Philip H.W. Leong. "Braiding: A Scheme for Resolving Hazards in Kernel Adaptive Filters". *In Proc. International Conference on Field Programmable Technology (FPT)*, pages 136–143, 2015. (doi:10.1109/FPT.2015.7393140)
- Nicholas J. Fraser, **Duncan J.M. Moss**, JunKyu Lee, Stephen Tridgell, Craig T. Jin, and Philip H.W. Leong. "A Fully Pipelined Kernel Normalised Least Mean Squares Processor for Accelerated Parameter Optimisation". *In Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, 2015. (doi:10.1109/FPL.2015.7293952)
- Nicholas J. Fraser, **Duncan J.M. Moss**, Nicolas Epain, and Philip H.W. Leong. "Distributed Kernel Learning using Kernel Recursive Least Squares". *In Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5500–5504, 2015. (doi:10.1109/ICASSP.2015.7179023)
- Lei Li, Farzad Noorian, **Duncan J.M. Moss**, and Philip H.W. Leong. "Rolling Window Time Series Prediction using MapReduce". *In Proc. IEEE International Conference on Information Reuse and Integration (IRI)*, pages 757–764, 2014. (doi:10.1109/IRI.2014.7051965)

Acknowledgements

First, I want to express my sincere thanks and gratitude to Professor Philip H.W. Leong. His foresight, novel ideas and encouragement have inspired me to push further than I thought possible. Throughout my time as his student, his patient guidance and commitment have been invaluable and I am deeply indebted for all that he has done.

I must express my gratitude to my wife, Emily. She has been my foundation, always providing a loving, comforting and consistent environment in which I could solely focus on my studies. I am still amazed by her ability to proofread countless pages of drafts, I can not thank her enough.

I want to thank the many people who helped me along way in the University of Sydney's Computer Engineering Lab. Specifically, Mr. Nicholas Fraser, Dr. Farzad Noorian, Mr. Stephen Tridgell and Dr. David Boland, whom have been close collaborators.

I also want to express my gratitude to the people I have collaborated with at the Intel Corporation. Especially Mr. Suchit Subhaschandra whom was my supervisor and mentor throughout my time there, and Mr Srivatsan Krishnan who looked after and guided me.

Finally, I want to thank my family. Leonard and Jana Gillespie have housed, fed and supported me through my studies, for that, I am deeply grateful.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

AAL	Intel **A**ccelerator **A**bstraction **L**ayer
AMBA	**A**dvanced **M**icrocontroller **B**us **A**rchitecture
API	**A**pplication **P**rogramming **I**nterface
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BBS	Intel **B**lue **B**it**S**tream
BIN	**BIN**ary
BLAS	**B**asic **L**inear **A**lgebra **S**ubroutine
BNN	**B**inarised **N**eural **N**etwork
BRAM	**B**lock **R**andom **A**ccess **M**emory
CCI	**C**ache **C**oherent **I**nterface
CNN	**C**onvolutional **N**eural **N**etwork
CONV	**CONV**olution Layer
CPU	**C**entral **P**rocessing **U**nit
DFT	**D**iscrete **F**ourier **T**ransform
DSP	**D**igital **S**ignal **P**rocessing
FC	**F**ully **C**onnected Layer
FF	**F**lip-**F**lop
FFT	**F**ast **F**ourier **T**ransform
FM	**F**requency **M**odulation
FP	**F**loating-**P**oint
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FXD	**F**ixed-**P**oint
GEMM	**G**eneralised **M**atrix to **M**atrix **M**ultiplication
GPU	**G**raphical **P**rocessing **U**nit
HARPv2	**H**eterogeneous **A**ccelerator **P**latform **v**ersion **2**
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh-**L**evel **S**ynthesis
HW	**H**ard**W**are
INT	**INT**eger
IPS	**I**mages **P**er **S**econd

LE	**L**ogic **E**lements
LSB	**L**east-**S**ignificant **B**it
LUT	**L**ook**U**p **T**able
MAC	**M**ultiply-**AC**cumulate
MLP	**M**ulti**L**ayer **P**erceptron
MSB	**M**ost-**S**ignificant **B**it
NN	**N**eural **N**etwork
PCI	**P**eripheral **C**omponent **I**nterconnect
PCI-e	**P**eripheral **C**omponent **I**nterconnect **e**xpress
PE	**P**rocessing **E**lement
PR	**P**artial **R**econfiguration
QPI	**Q**uick **P**ath **I**nterconnect
ReLU	**Re**ctified **L**inear **U**nit
RF	**R**adio-**F**requency
RTL	**R**egister **T**ransfer **L**anguage
SM	**S**ign **M**agnitude
SoC	**S**ystem-**o**n-**C**hip
SW	**S**oft**W**are
TC	**T**rue-and-**C**omplement
TOPs	**T**era-**O**perations **P**er **s**econd
TPU	**T**ensor **P**rocessing **U**nit
TS	**T**wo **S**peed
USRP	**U**niversal **S**oftware **R**adio **P**eripheral

Chapter 1

Introduction

1.1 Motivation

Machine Learning is undergoing a renaissance. With deep learning at its forefront, computers are able to provide better-than-human level accuracy on image recognition, drive cars unassisted and perform in-depth analysis of personal health and fitness biometrics. Machine learning is changing the way we access, interpret and interface with the world [3, 4]. From social media to instrumentation; companies are beginning to adopt machine learning into their organisations and customer facing products as a method to increase revenue and improve customer experience. Modelling system behaviour, extracting customer interactions and facial recognition, to name a few, are important parts of frameworks and applications used in both industry and research. Whether it is an embedded device, such as a mobile phone, or a permanent cloud-based installation, machine learning is now deployed at every level and scale. As the benefits of machine learning become increasingly apparent, data is collected and processed at an unprecedented scale. Consequently, there is a growing concern that traditional computing resources will be insufficient to handle this exponential increase [5–7].

The computing resources required to efficiently develop and deploy machine learning algorithms is often overlooked [6]. For example, AlexNet, which is considered small by today's standards, requires on the order of 10^9 math operations per image [8]. As companies begin to see the potential of machine learning, there will be a dramatic increase in the computing resources required [9]. To meet customer needs, companies are turning to large cloud providers such as Amazon, Google or Microsoft to provide the required compute and infrastructure. Algorithms are computationally intensive, especially during development, and the Central Processing Unit (CPU) is insufficient to handle large scale workloads [7]. In response, cloud providers are offering accelerators

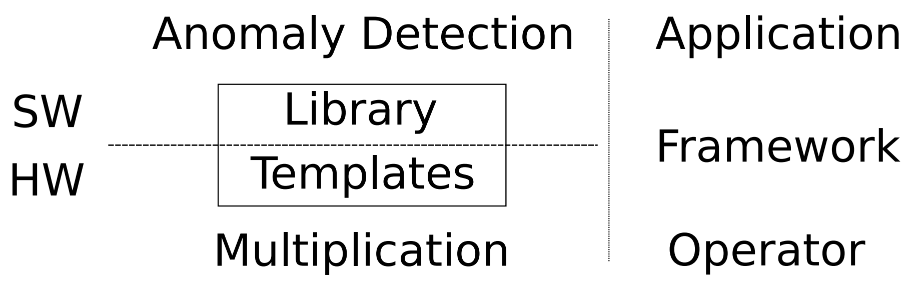


FIGURE 1.1: Thesis Overview

such as Graphical Processing Units (GPUs), and more recently Field Programming Gate Arrays (FPGAs) and dedicated Application Specific Integrated Circuits (ASICs) as alternatives [10]. As a recent example, Google now offers their own reduced precision ASIC called the Tensor Processing Unit (TPU), illustrating their need for high efficiency compute for deep learning and machine learning [9].

In an effort to reduce the computational complexity and overall processing requirements of machine learning algorithms, there has been significant work focusing on reduced precision representations [11–17]. Historically, single precision floating-point has been the widely accepted gold standard due to its large dynamic range. However, in digital systems, floating-point representation requires significant amounts of additional circuitry to support its implementation, sacrificing performance.

1.2 Aims and Contributions

Reduced precision representations offer two key optimisations:

1. Lower computation complexity leading to reductions in power, area and delay.
2. A smaller memory footprint as fewer bits are required for parameters.

The FPGAs' reconfigurable architecture allows for fast and relatively cheap changes to datapath precision compared to other devices. As a result, FPGAs are uniquely positioned to take advantage of reduced precision representations, since the datapaths of CPUs, GPUs and ASICs are fixed at deployment. Lower precision representations can be optimised for the FPGAs' look-up table based architecture and the reconfigurable fabric makes adapting to algorithmic requirements fast and simple. This makes FPGAs the perfect fit for lower precision machine learning.

This thesis presents FPGA architectures for low precision machine learning at three distinct levels: the application, the framework and the operator. Precision optimisations at the application level involves leveraging the reconfigurable architecture of FPGAs to take advantage of arbitrary precision fixed-point datapaths. Depending on the application's operating requirements, different precisions can be selected to optimise for area, latency, accuracy and power. As illustrated in Figure 1.1, to support different precisions at the application level, a customisable framework that contains a software library and hardware template is presented. The framework is designed to provide a consistent and reusable environment in which reduced precision applications can be constructed. With multiplication as the fundamental operator used in machine learning applications and frameworks, this thesis offers an optimised multiplication algorithm and implementation designed for seamless integration which provides improved performance for these algorithms.

Working from top to bottom in Figure 1.1, a reduced precision radio-frequency (RF) anomaly detection application is presented. The anomaly detector is given as an example of a machine learning application where reduced precision representations allow for larger model sizes, resulting in improved detection performance over floating-point. The detector is a neural network (NN) based autoencoder, taking either raw Quadrature (IQ) windows, or complex-valued frequencies. It is evaluated for a range of different precisions and analysed in terms of area, latency, throughput and detector accuracy. Another reduced precision detector based on a N-dimensional symbolic bitmap technique is presented and compared against the neural network detector. In contrast to the neural network detector, the hardware implementation is area and compute efficient requiring nine additions and multiplications, performed in $O(1)$ time.

Second, a generalised matrix multiplication framework with variable precision, designed for accelerating machine learning applications is explored. The framework is designed to abstract the complexity of implementing custom reduced precision accelerators on FPGAs and is easily integrated into existing platforms. It contains a software library and hardware template, providing a consistent and reusable interface to the FPGA. The library contains a software application programming interface (API) for easy integration into existing deep leaning and machine learning applications. The hardware template contains key runtime configurable optimisations such as a wide range of precisions, blocking, fusing of operations and a customisable interleaving scheme; offering substantial improvements for small matrix sizes and neural networks. The framework is evaluated for three state-of-the-art neural networks (AlexNet [8], VGGnet [18] and ResNet [4]), in addition to an in-depth comparison to the latest Pascal NVIDIA GPU and other related works in the area.

Finally, a new operator is presented; a novel bit-dependent multiplier algorithm for accelerating machine learning applications. The multiplier implementation is designed for easy integration into existing FPGA accelerator frameworks. By splitting the computational datapath into two separate sub-circuits, the multiplier quickly deduces whether an accumulation is required, otherwise it skips the computation. The multiplier is evaluated for multiple popular neural networks and compared against standard multiplication techniques.

In summary, the contributions of this thesis are:

- A single-chip RF anomaly detector, utilising a neural network, which is fully pipelined, producing an output every cycle. It leverages a heterogeneous architecture which supports updating the weights while inference proceeds, enabling simultaneous learning and inference with the former conducted on a processor or graphics processing unit [19, 20].
- A runtime configurable heterogeneous GEMM implementation which supports arbitrary matrix sizes and offers a wide range of precision, blocking, fusing of operations, buffering schemes and load balancing. It includes a dynamic dot product, enabling mixed precision training and binary inference [21, 22].
- A two speed multiplication algorithm where the datapath is divided into two subcircuits, each operating with a different critical path. This multiplier takes advantage of particular bit-patterns to perform less work; this results in reduced latency, increased throughput and superior area-time performance compared to conventional multipliers [23].

1.3 Thesis Structure

An introduction to the concepts of computer arithmetic, machine learning and deep learning is presented in Chapter 2. The chapter begins with an overview of computer arithmetic, focusing on reduced precision representations and their computation. Next, the core ideas of machine learning are presented along with their mathematical descriptions. Finally, neural networks, convolutional neural networks and their various computational methods are explored.

In Chapter 3, the anomaly detection application is explored and the key concepts of anomaly detection are discussed in detail. The neural network anomaly detector for real-time processing of radio-frequency signals is described. The implementation details of the autoencoder and fast Fourier transform algorithm are discussed. Furthermore, an on-line algorithm for computing the discrete Fourier transform, sampled at irregular time intervals is introduced along with an on-line bitmap anomaly detection technique. The neural network detector and bitmap detector are evaluated and their results presented. Finally, the resulting hardware implementations and design concerns are explored.

Chapter 4 covers the generalised matrix multiplication framework, designed for deep learning applications such as those presented in Chapter 3. Beginning with an overview of the system, a summary of the software API and various tuneable parameters in the hardware template are described. This is followed by an in-depth look at the hardware template and key runtime configurable optimisations such as fusing of operations, customisable interleaving schemes and dynamic dot product. Next, a discussion outlining the implementation details of mixed precision computation are presented and evaluated against a state-of-the-art GPU, illustrating that the FPGA remains competitive in terms of performance per watt and outperforms the GPU on more exotic precision types. Finally, the performance benefits of interleaving are explored and the framework is evaluated using three binarised neural networks (AlexNet [8], VGGnet [18] and ResNet [4]). A discussion on the dynamic dot product module and a comparison to previous work ends the chapter.

With the importance of multiplication in deep learning established, Chapter 5 presents the Two Speed multiplier. It outlines an overview of the radix-4 Booth algorithm [24] and the necessary changes needed to support the two speed optimisation. This is followed by an illustrative example describing the control flow and a model for estimating the multiplier's performance. The chapter concludes with a comparison to three standard methods, evaluated in terms of area, time and power.

The final chapter presents a summary of the work in this thesis and the direction of future work.

Chapter 2

Background

2.1 Introduction

This chapter presents the concepts and terminology used in computer arithmetic and architecture, reduced precision representations, multiplier implementations and neural networks relevant to this thesis. It begins by presenting background information on computer arithmetic and reduced precision representations; focusing on fixed-point representations and different multiplier types. This is followed by a description of machine learning and neural networks. The chapter concludes with an examination of neural network computational techniques and the convolution layer.

2.2 Field Programmable Gate Arrays

A Field Programmable Gate Array is a configurable two dimensional grid of flexible logic blocks [25]. These logic blocks contain look-up tables which can be configured to express any logic function of their given inputs, and memory for storing bits, called registers. The inputs and outputs of these logic blocks are connected together via programmable interconnections, implemented as programmable switch matrices. This technology has a distinct advantage over fixed function processors such as CPUs and GPUs as its flexibility offers different approaches to implement a given algorithm. Higher degrees of parallelism can be achieved through replication of computing units, pipelining and reducing instruction execution overhead. Moreover, the use of FPGAs offers the potential to tightly integrate machine learning with lower level data acquisition and/or networking hardware, reducing buffer sizes and further optimising latency.

In addition to logic cells, FPGAs contain two other common blocks: block memories (BRAM) and digital signal processing (DSP) blocks. BRAMs are dedicated one or two port memories which vary in size depending on the vendor and device. Each port on the BRAM can be configured as either an input or output. DSP blocks are dedicated math blocks, usually containing an adder and multiplier. Recently FPGAs have started to add DSP blocks that include native single precision floating-point support. [26] These blocks are fully pipelined and perform a multiply-accumulate every cycle with some result latency. These two hardened blocks, along with the logic cells form the majority of the FPGAs area and are arranged in columns.

An FPGA design is created by a two stage process named 'synthesis' and 'place and route'. These processes take a hardware description and determine the best configuration of the flexible compute blocks. *Synthesis* takes files written in a hardware description language (HDL) and translates them to a list of wire interconnects and logic functions; this is called the *net list*. The *net list* is used in the *map* stage to create a physical representation of the *net list* which is used in the *place and route* stage. The *place and route* stage determines the exact configuration and layout of the device's logic blocks to meet the desired timing and area constraints. This process creates a hard-wired configuration to implement the desired circuit. The exact layout is then stored in a file called the bitstream. Historically, users have been required to express the desired functionality using an HDL, however this is often verbose and overly complicated. To overcome this, there has been significant work in high-level synthesis (HLS) [27, 28]. HLS abstracts away many of the hardware specific details in an HDL and allows the user to express their desired functionally in a more succinct language.

FPGAs can be used as either standalone devices or heterogeneous accelerators [29]. The work in this thesis focuses on heterogeneous accelerators for machine learning applications. Heterogeneous CPU and FPGA systems are typically connected via [30–32]:

- Advanced Mircocontroller Bus Architecture (AMBA) Interconnect, or
- The Peripheral Component Interconnect (PCI), PCI Express (PCI-e), Ethernet or Serial Communication ports (JTAG, COMM, etc.).

The AMBA interconnect is designed for System-on-Chip (SoC) devices running an ARM processor. While an in-depth understanding of the differences between ARM processors and x86 processors is not required for this thesis, it is sufficient to say that in general, x86 processors include more complex circuitry; resulting in higher performance and provide additional functionality compared to ARM processors. As a result, x86 processors require more power and have been more suited to desktop and server based machines, whereas ARM processors are used in embedded and low power applications such as mobile phones. As such, heterogeneous CPU and FPGA systems based on the AMBA interconnect are designed for low power embedded applications. One of the main interfaces defined in AMBA is the Advanced eXtensible Interface (AXI). AXI targets devices with high clock frequencies, thus making it an appropriate choice for high performance FPGAs.

Connecting an FPGA over PCI, PCI-e, Ethernet or other serial ports is a method for communicating with the FPGA on a discrete board. These discrete FPGAs have dedicated memory located on their circuit boards that interface with the CPU via various ports and pins. A typical application that involves offloading work to the FPGA is performed in three steps: (1) the CPU transfers data to the FPGA dedicated memory, (2) the FPGA will access that memory and perform its compute and (3) the result will be returned to the dedicated memory, ready for the CPU to transfer it back. For heterogeneous accelerators, exchange of data between the CPU and FPGA is often a major performance bottleneck.

While there are other heterogeneous processor platforms that contend with ARM and x86, the prevalence of these two are motivating a shift towards the interconnect standardisation [33]. Ease of use and low level software-hardware abstraction are valued highly since they allow hardware blocks to operate on many different platforms without significant redesign or reimplementation. Whilst other heterogeneous platforms may be available, these two cover the majority of hardware designs.

2.3 Digital Arithmetic

Digital Arithmetic is the study of number representations, algorithms for operations on numbers, implementations of arithmetic units and their use in general-purpose and application-specific systems [34]. The two common cases of digital numbers are:

- fixed-point (FXD) numbers: represented by an integer $I = \{-N, \dots, N\}$ and a rational number $x = a/2^f$, where $a \in I$ and f is the fractional length, represented as a positive integer
- floating-point (FP) numbers: $x \times b^E$, x is a rational number, b the integer base, and E the integer exponent.

This section introduces the basic number systems for the fixed-point representation, which is used in the following chapters. The floating-point representation described is only used for evaluation later in the thesis.

2.3.1 Fixed-Point Number Representation

Performing operations on fixed-point numbers requires a specific representation [34]. A number is represented by an ordered n-tuple, called a digit-vector, with each element of the n-tuple a digit and the number of digits n denoting the precision of the representation. First, a representation of non-negative integers is presented, followed by the representation of signed integers and finally an extension to fractional fixed-point numbers.

The integer x is represented by the zero-origin, leftward-increasing indexed digit-vector:

$$X = (X_{n-1}, X_{n-2}, \ldots, X_1, X_0) \tag{2.1}$$

A number system which represents x requires: the number of digits n, a set of allowable values for the digits, and a rule of interpretation that corresponds to a mapping between the set of digit-vector values and the set of integers. D_i denotes the set of values X_i, for example $D_i = \{0, 1, 2, \ldots, 9\}$ is the digit set for the conventional decimal number system. A set of integers, each represented by a digit-vector with n digits, is a finite set with at most $K = \prod_{i=0}^{n-1} |D_i|$ different elements, where $|D_i|$ denotes the cardinality of set D_i.

If the representation is a one-to-one mapping between each digit-vector and a different integer, then the number system is considered non-redundant. It follows that in a redundant number system, multiple digit-vectors map to the same integer. Generally, number systems are weight systems, their mapping represented by:

$$x = \sum_{i=0}^{n-1} X_i W_i \tag{2.2}$$

where $W = (W_{n-1}, W_{n-2}, \ldots, W_1, W_0)$ is the weight-vector.

Radix number systems are an example of a weight number system, where the weight-vector is related to a radix-vector $R = (R_{n-1}, R_{n-2}, \ldots, R_1, R_0)$ as follows:

$$W_0 = 1; \quad W_i = \prod_{j=0}^{i-1} R_j \tag{2.3}$$

In a fixed radix system all elements of the radix-vector R have the same value r, hence the weight-vector is:

$$W = (r^{n-1}, r^{n-2}, \ldots, r^2, r, 1) \tag{2.4}$$

and substituting Equation 2.4 into Equation 2.2

$$x = \sum_{i=0}^{n-1} X_i \cdot r^i \tag{2.5}$$

A canonical system is a set of values for $D_i = \{0, 1, 2, \ldots, R_i - 1\}$, with $|D_i| = R_i$. A non-canonical system is a set of digit values that are not canonical, i,e $|D_i| \neq R_i$. In the case of $|D_i| > R$, it allows for more than one representation of a value, resulting in a redundant system. A number system with fixed positive radix r and a canonical set of digit values is called a radix-r conventional number system. For the commonly used radix-2, the corresponding weight-vector is $W = (\ldots, 16, 8, 4, 2, 1)$.

2.3.1.1 Signed Integers

The non-negative integer representation is now extended to a signed representation to facilitate the negative integers. There are two options for signed systems, Sign-and-Magnitude (SM) and True-and-Complement (TC).

A signed integer x is represented in the SM system by a pair (x_s, x_m) where x_s is the sign and x_m is the magnitude, represented as a positive integer. Generally, the two values of the sign $(+, -)$ are represented by a binary variable, where 0 corresponds to $+$ and 1 to $-$. The magnitude can be represented by any system, in the case of a radix-r system:

$$0 \leq x_m \leq r^n - 1 \tag{2.6}$$

where n is the number of digits in the magnitude representation.

The TC system makes no distinction between the representation of the sign and magnitude. An additional mapping, defined below, is required to make this representation work. A signed integer x can be represented in the TC system by a positive integer x_R such that:

$$x_R = x \mod C \tag{2.7}$$

where C is the complementation constant, a positive integer. For $max|x| < C$, Equation 2.7 is equivalent to:

$$x_R = \begin{cases} x & \text{if } x \geq 0. \\ C - |x| = C + x & \text{if } x < 0. \end{cases} \tag{2.8}$$

Following this, in order to have an unambiguous representation it is required that $max(x) < C/2$, such that the region for $x > 0$ and $x < 0$ do not overlap. The converse mapping is

$$x = \begin{cases} x_R & \text{if } x_R < C/2. \\ x_R - C & \text{if } x_R > C/2. \end{cases} \tag{2.9}$$

When $x_R = C/2 = -x$ the representation is asymmetrical. The positive integers are called the true forms and the negative integers the complement forms.

The positive integer x_R can be represented in any system. For a digit-vector of n digits the range is:

$$0 \leq x_R \leq r^n - 1 \tag{2.10}$$

Usually the complementation constant is $C = r^n$ or $C = r^n - 1$. In the case of radix-2, this results in the two's complement system $C = 2^n$ and the one's complement system $C = 2^n - 1$. For the two's complement system $x_R = 2^{n-1}$ can represent either $x = 2^{n-1}$ or $x = -2^{n-1}$, resulting an an asymmetric representation. Historically $x = 2^{n-1}$ is chosen to simplify the sign detection. The range of signed integers in the two's complement system is:

$$-2^{n-1} \leq x \leq 2^{n-1} - 1 \tag{2.11}$$

For the one's complement system $C = r^n - 1$, $x_R = C$ is represented with n digits hence there are two representations of $x = 0$: $x_R = 0$ and $x_R = 2^n - 1$. The range of signed integers in the ones's complement system is:

$$-(2^{n-1} - 1) \leq x \leq 2^{n-1} - 1 \tag{2.12}$$

Sign detection in the SM and TC systems is:

$$sign(x) = \begin{cases} 0 & \text{if } x \geq 0. \\ 1 & \text{if } x < 0. \end{cases} \tag{2.13}$$

For the SM case, detection is trivial as there is a dedicated signed bit x_s. In the TC system, since $|x| \leq C/2$, the sign is determined by:

$$sign(x) = \begin{cases} 0 & \text{if } x_R < C/2. \\ 1 & \text{if } x \geq C/2. \end{cases} \tag{2.14}$$

In the case of two's complement and ones's complement systems the most-significant bit is used:

$$sign(x) = X_{n-1} \tag{2.15}$$

For the two's complement case, mapping from the digit-vector to the value is performed by:

$$x = -X_{n-1}2^{n-1} + \sum_{0}^{n-2} X_i 2^i \tag{2.16}$$

2.3.1.2 Fixed-Point Fraction Representation

This last part further extends the signed integer representation to account for fractional values. A fixed-point representation of a number $x = x_{INT} + x_{FR}$ consists of integer and fraction components represented by m and f digits, respectively, where $n = m + f$. Equation 2.17 shows the specific notation.

$$X = (X_{(m-1)} \dots X_1 X_0 . X_{-1} \dots X_{-f}) \tag{2.17}$$

For a two's complement system x_{INT} is calculated by:

$$x_{INT} = -X_{m-1}2^{m-1} + \sum_{0}^{m-2} X_i 2^i \tag{2.18}$$

and x_{FR} is calculated by:

$$x_{FR} = \sum_{-f}^{0} X_i 2^i \tag{2.19}$$

It follows that a two's complement radix-2 fixed-point number can be calculated by:

$$x = x_{INT} + x_{FR} = -X_{m-1}2^{m-1} + \sum_{-f}^{m-2} X_i 2^i \tag{2.20}$$

For example, $-1.75 \leq x \leq 1.50$ is represented by $X = (X_1 X_0 . X_{-1} X_{-2})$.

2.3.2 Multiplication

Multiplication is arguably the most important primitive in machine learning and digital signal processing applications. Sze et. al [35] illustrated that the majority of hardware optimisations are aimed at improving multiplication, since multiplier circuitry grows with complexity n^2, as opposed to adders which grow with complexity n. Hence, careful construction of the compute unit, with a focus on multiplication, leads to the largest performance impact. Utilising dedicated FPGA multiplication hardware or optimisation via reduced precision representations are important aspects of design that need to be carefully considered.

This section presents an algorithm for the multiplication of positive integers. This is followed by its extension for signed integers. Let x and y be the multiplicand and the multiplier, represented by n digit-vectors X and Y in a radix-r conventional number system. The multiplication operation produces $p = x \times y$, where p is represented by the $2n$ digit-vector P. Multiplication is described as:

$$p = x \sum_{i=0}^{n-1} Y_i r^i = \sum_{i=0}^{n-1} r^i x Y_i \tag{2.21}$$

Equation 2.21 describes a process that first computes the n terms xr^iY_i and then performs the summation. Computation of the ith term involves a i-position left shift of X and the multiplication of a single radix-r digit Y_i. This single radix-r digit multiplication is a scaling factor of the ith digit in the digit-vector set. In the case of radix-2, this is either 0 or 1. Performing the computation in this manner lends itself to a combinational or parallel multiplication unit.

Instead, the multiplication can be expressed recursively:

$$\begin{aligned} p[0] &= 0 \\ p[j+1] &= r^{-1}(p[j] + r^n x Y_j) \quad \text{for } j = 0, 1, \ldots, n-1 \\ p &= p[n] \end{aligned} \tag{2.22}$$

Expanding this recurrence results in product $p[n] = x \times y$ in n steps. Each time step j consists of a multiplication of x by a radix-r digit, xY_j, similar to Equation 2.21. This is followed by n digit left shift, and accumulated with the result from the previous time step $p[j]$. The recurrence is finished with a one digit right shift. The recurrence is expressed in this manner to ensure that the multiplication can proceed from the least-significant digit of the multiplier y, while retaining the same position with respect to the multiplicand x. An example of a radix-2 multiplication is illustrated in Figure 2.1

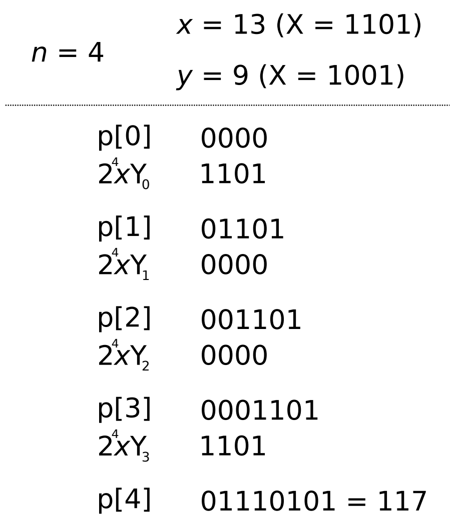


FIGURE 2.1: Unsigned Multiplication $p = x \times y$, where x is the multiplicand, y is the multiplier and X and Y are their respective $n = 4$ digit-vectors in the radix-2 conventional number system.

Extending the multiplication, Equation 2.21, to the two's complement system is trivial and by examining the multiplier y:

$$y = -Y_{n-1}2^{n-1} + \sum_{0}^{n-2} Y_i 2^i \tag{2.23}$$

and substituting it into Equation 2.21, the new multiplication expression is given by:

$$p = \sum_{i=0}^{n-2} xY_i r^i - xY_{n-1}2^{n-1} \tag{2.24}$$

Figure 2.2 illustrates the same example as Figure 2.1, however with the new two's complement system and updated Equation 2.24.

As per Section 2.3.1.1, the negation of x $(-x)$ is performed by flipping all of the bits $(bf(1101) = 0010)$ then adding a single bit in the least-significant position $(0010 + 1 = 0011)$.

2.3.3 Floating-Point Number Representation

This section introduces floating-point number representation [36]. While not explicitly used in this thesis for reduced precision optimisations, most of the work is compared against this representation. A floating-point number x consists of two components, the significand M_x^* and the exponent E_x. These components are combined with a constant

$n = 4$	x = -3 (X = 1101)
	y = -7 (X = 1001)
p[0]	0000
$2^4 x Y_0$	1101
p[1]	11101
$2^4 x Y_1$	0000
p[2]	111101
$2^4 x Y_2$	0000
p[3]	1111101
$-2^4 x Y_3$	0011
p[4]	00010101 = 21

FIGURE 2.2: Signed two's complement Multiplication $p = x \times y$, where x is the multiplicand, y is the multiplier and X and Y are their respective $n = 4$ digit-vectors in the radix-2 conventional number system.

b, called the base:

$$x = M_x^* \times b^{E_x} \tag{2.25}$$

The sign of the number x is determined by the sign of the significand M_x^*. The significand M_x^* can be represented using any representation system, SM or TC, however it is common practice to use SM. It follows that a floating-point number x can be represented by a tuple (S_x, M_x, E_x).

$$x = (-1)^{S_x} \times M_x \times b^{E_x} \tag{2.26}$$

In this case $M_x^* = (-1)^{S_x} \times M_x$, where S_x is the sign such that $S_x \in \{0, 1\}$ and M_x denotes the magnitude of the significand.

The floating-point representation is redundant; a floating-point number can have several representations. This redundancy is not helpful and to improve accuracy, a *normalised* representation is defined so that the most-significant digit of the significand always differs from zero. This reduces the range of floating-point numbers so that the smallest representable number is:

$$x = r^{m-f-1} \times b^{E_x} \tag{2.27}$$

To avoid this reduction in range, a special set of unnormalised numbers called *denormalised numbers* are allowed. These *denormalised numbers* are the set of values that cannot be represented in normalised form.

The main advantage of a floating-point representation is the increased dynamic range; which is defined as the ratio between the largest and smallest number that can be represented in the number system. The drawback of the floating-point representation is that it is less precise. As mentioned above, the precision of a representation corresponds to the number of digits in the significand, since the n digits in the representation is divided between the significand and the exponent, a floating-point number of n digits has a smaller precision than its n digit fixed-point counterpart.

In addition to smaller precision, floating-point numbers have more complex circuit implementations, requiring larger area and greater delay. Addition, subtraction, multiplication and division of a floating point number involves three stages: shifting, adding and rounding. During addition and subtraction, the exponent of the two operands need to match, this is achieved by shifting one of the significands. After matching the exponents, integer addition is performed on the significands as normal and the result is rounded and normalised. Multiplication and division on the other hand involve adding/subtracting the exponents, integer multiplation/division on the significands and is finalised by rounding and normalisation. Each basic operation requires three separate stages, often with complex rules regarding rounding and normalisation. In contrast to the integer and fixed-point operations, which involved a simple adder or multiplier circuit, the three stage complexity of the floating-point representation significantly increases the hardware utilisation cost. To support floating-point operations efficiently in FPGA, the vendors are including IEEE 754 compliant DSP [26].

2.4 Machine Learning

Machine Learning aims to model a process based on its history [37]. The core objective is for a learner to generalise from past information. Given some large set of N inputs $\{x_1, x_2, \ldots, x_N\}$, known as the training set, and their target $\{y_1, y_2, \ldots, y_N\}$, the goal of the learner is to find some function $f(x)$, such that the predictions $\{\bar{y_1}, \bar{y_2}, \ldots, \bar{y_N}\}$, generated by Equation 2.28, match the targets.

$$\bar{y_i} = f(x_i) \tag{2.28}$$

where i is one of the elements in the dataset. Note that for each input x, there is one target y. The function $f(x)$ is determined during the training phase, also known as the learning phase. Once the precise form of the function $f(x)$ is known, new inputs, not originally from the training set can then be identified. Identifying these new inputs is performed during the inference phase.

A simple regression problem, fitting a polynomial curve, is considered a machine learning problem. The formulation of the function $f(x)$ is now extended to include the coefficients of the polynomial, $\boldsymbol{w}$, to $f(x, \boldsymbol{w})$ hence the curve can now be expressed as:

$$f(x, w) = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j \tag{2.29}$$

where M is the order of the polynomial and x^j is x raised to the power of j.

The coefficients will be determined during the training phase by minimising the error function $E(\boldsymbol{w})$. An error function $E(\boldsymbol{w})$ measures the difference between the function $f(x, \boldsymbol{w})$, for any given value of $\boldsymbol{w}$, and the target y_i of any particular input x_i. In this case, a typical error function is given by the mean squared error between the predictions $\bar{y_i}$, calculated using $f(x_i, \boldsymbol{w})$, and the target y_i. Hence Equation 2.30 is minimised to find the values of the coefficients, such that $\{y_1, y_2, \ldots, y_N\} = \{\bar{y_1}, \bar{y_2}, \ldots, \bar{y_N}\}$.

$$E(\boldsymbol{w}) = \frac{1}{2n} \sum_{i=0}^{N} (f(x_i, \boldsymbol{w}) - y_i)^2 \tag{2.30}$$

In this simple example, the error function is a quadratic function of the coefficients $\boldsymbol{w}$, hence the minimisation of the error function has a unique solution; the derivative of the error function with respect to the coefficients which is found in closed form. For other machine learning algorithms such as neural networks, finding the precise form $f(x)$ is a more complex problem, requiring dedicated training algorithms.

2.5 Deep Learning

Deep Learning is the study and application of many-layered neural networks, trained on large datasets [37, 38]. Neural networks are a class of machine learning algorithms that are described as a connected graph of basic compute nodes called neurons. It operates in the same fashion as the polynomial model, Equation 2.29, This section begins by introducing the fundamental unit in a neural network, the neuron and is extended to a single layer network, which is a one dimensional collection of neurons. The single layer network is then expanded upon to a multilayered description. A method for training neural networks is described and a discussion on other neural network layer types and computation methods concludes the section.

2.5.1 The Neuron

A neuron is the fundamental compute unit in a neural network and operates in a similar fashion to the polynomial model, Equation 2.29. Given a single M dimensional input $\boldsymbol{x} = \{x_M, x_{M-1}, \dots, x_2, x_1\}$, a vector of coefficient $\boldsymbol{w} = \{w_M, w_{M-1}, \dots, w_2, w_1\}$ called weights, an activation function $\sigma(z)$ and a bias b, where M is the input dimensionality. The output of a neuron a is defined by:

$$\begin{aligned} a &= \sigma(x_M w_M + x_{M-1} w_{M-1} + \cdots + x_2 w_2 + x_1 w_1 + b) \\ &= \sigma(\sum_{i=1}^{M} x_i w_i + b) \\ &= \sigma(\boldsymbol{w}^T \boldsymbol{x} + b) = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b) \end{aligned} \tag{2.31}$$

In Equation 2.31, the corresponding inputs and weights are multiplied together and summed with the bias which is passed to the activation function. In vector form, this is expressed as $\sigma(\boldsymbol{w}^T \boldsymbol{x} + b)$ or $\sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$ denoting the dot product between the two vectors. Usually, $\sigma(z)$ denotes the sigmoid function, otherwise known as the logistic function and is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.32}$$

To understand the sigmoid function, consider the case when $z = \boldsymbol{w} \cdot \boldsymbol{x} + b \to +\infty$. Then $e^{-z} \approx 0$ and $\sigma(z) \approx 1$. In the case when $z = \boldsymbol{w} \cdot \boldsymbol{x} + b \to -\infty$, then $e^{-z} \approx \infty$ and $\sigma(z) \approx 0$. Figure 2.3 illustrates a three input neuron example, the activation function is implicit within the neuron itself. Finally, for convenience, it is common to represent the bias as the 0th weight $w_0 = b$ and set the 0th input to 1, resulting in $1 \times w_0 = b$. Hence Equation 2.31 can be further simplified to:

$$a = \sigma(\boldsymbol{w} \cdot \boldsymbol{x}) \tag{2.33}$$

where $\boldsymbol{w} = \{w_M, w_{M-1}, \dots, w_2, w_1, b\}$ and $\boldsymbol{x} = \{x_M, x_{M-1}, \dots, x_2, x_1, 1\}$.

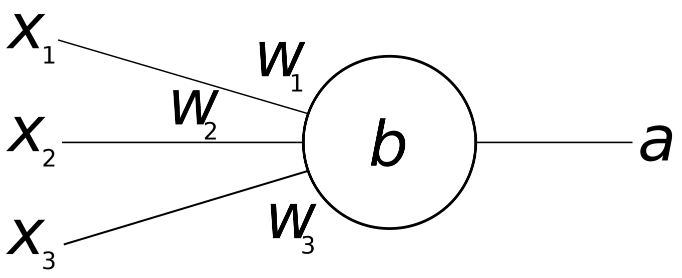


FIGURE 2.3: Single Neuron: The activation function $\sigma(z)$ is implicit within the neurons output.

2.5.2 Single Layered Network

A single layer neural network is a one dimensional vector of neurons. By extending the single neuron to a collection of neurons, the neural network can now act as non-linear function estimator. As shown in Figure 2.4, each neuron is connected to all inputs and produces it own output. Given the size of the layer D, The computation for a single layered network can be described as:

$$a_j = \sigma(\boldsymbol{w}_j \cdot \boldsymbol{x}) \quad \text{for } j = 1, 2, \ldots, D \tag{2.34}$$

where j denotes the jth neuron. The output of the single layer neural network is a vector of neuron output $\boldsymbol{a} = \{a_D, a_{D-1}, \ldots, a_2, a_1\}$. Since each neuron has own set of

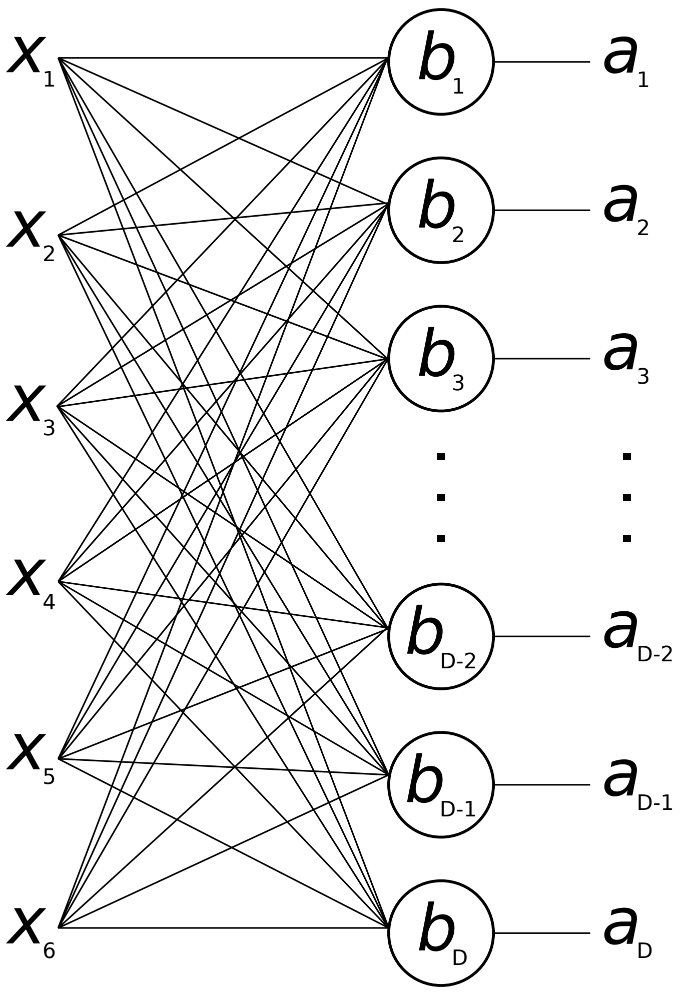


FIGURE 2.4: Single Layer Network: The weights $\boldsymbol{w}$ have been excluded to avoid clutter.

weights, the weights for the network W can be described as a matrix:

$$W = \begin{bmatrix} w_{D,M} & w_{D,M-1} & \dots & w_{D,1} & b_D \\ w_{D-1,M} & w_{D-1,M-1} & \dots & w_{D-1,1} & b_{D-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{2,M} & w_{2,M-1} & \dots & w_{1,1} & b_2 \\ w_{1,M} & w_{1,M-1} & \dots & w_{1,1} & b_1 \end{bmatrix} \tag{2.35}$$

Given that $z_j = \boldsymbol{w_j} \cdot \boldsymbol{x}$ for the jth neuron, the computation for all z's, $\boldsymbol{z} = \{z_D, \dots, z_1\}$ can be expressed as the matrix-vector multiplication $W\boldsymbol{x}^T$:

$$\boldsymbol{z} = \begin{bmatrix} z_D \\ \vdots \\ z_1 \end{bmatrix} = \begin{bmatrix} w_{D,M} & \dots & w_{D,1} & b_D \\ \vdots & \ddots & \vdots & \vdots \\ w_{1,M} & \dots & w_{1,1} & b_1 \end{bmatrix} \begin{bmatrix} x_M \\ \vdots \\ x_1 \\ 1 \end{bmatrix} \tag{2.36}$$

Expressed in vector form:

$$\boldsymbol{z} = W\boldsymbol{x}^T \tag{2.37}$$

Finally, if $\Phi(z)$ denotes an element-wise application of the activation function $\sigma(z)$, then computing the single layer network is:

$$\boldsymbol{a} = \Phi(W\boldsymbol{x}^T) \tag{2.38}$$

where $\boldsymbol{a}$ denotes the vector of outputs $\boldsymbol{a} = \{a_D, \dots, a_1\}$.

Up until this point, the description has only considered processing a single input from the training set at one time. As described in Section 2.4, machine learning often involves processing a large dataset of training examples. Processing multiple input vectors in a neural network is performed by sequentially evaluating Equation 2.38. This approach is very memory intensive. For large networks, the weights need to be loaded for every input vector. To alleviate this, multiple input vectors can be processed in batches by modifying Equation 2.36.

Given the dataset of input vectors $X = \{\boldsymbol{x}_B, \dots, \boldsymbol{x}_1\}$ where B denotes the batch size. The corresponding set of output vectors $Z = \{\boldsymbol{z}_B, \dots, \boldsymbol{z}_1\}$ is given by:

$$Z = \begin{bmatrix} z_{D,B} & \dots & z_{D,1} \\ \vdots & \ddots & \vdots \\ z_{1,B} & \dots & z_{1,1} \end{bmatrix} = \begin{bmatrix} w_{D,M} & \dots & w_{D,1} & b_D \\ \vdots & \ddots & \vdots & \vdots \\ w_{0,M} & \dots & w_{0,1} & b_1 \end{bmatrix} \begin{bmatrix} x_{M,B} & \dots & x_{D,1} \\ \vdots & \ddots & \vdots \\ x_{0,B} & \dots & x_{0,1} \\ 1 & \dots & 1 \end{bmatrix} \tag{2.39}$$

In matrix form:

$$\begin{aligned} Z &= WX \\ A &= \Phi(WX) \end{aligned} \tag{2.40}$$

where A is the output matrix after the element-wise activation function $\Phi(z)$.

In summary, computing a single layer network can be performed using a matrix-vector multiplication and the single input vector form $\boldsymbol{a} = \Phi(W\boldsymbol{x}^T)$. If multiple inputs are computed concurrently, a matrix-matrix multiplication can be used with the matrix form $A = \Phi(WX)$.

2.5.3 Multilayered Network

Building upon the single layer network, Figure 2.5 illustrates an example of a four layer network. The nomenclature for mutlilayer networks is as follows: first the *input layer* $\boldsymbol{x}$, followed by a series of *hidden layers*, also known as fully connected layers (FC), $\boldsymbol{a}^1, \boldsymbol{a}^2$ and $\boldsymbol{a}^3$ where the superscript denotes the layer number, and finally the *output layer* $\boldsymbol{a}^4$. Historically, this is known as a Multilayer Perceptron (MLP). Multilayer networks are the most commonly used today since they are able to represent more complex models than their single layer counterparts. Additionally, more complex non-linear combinations can be learned as subsequent layers are connected to the previous layer's interpretation of its input, i.e. learned information is not restricted to only the inputs; Using hidden layers allow more complex interactions to be learnt.

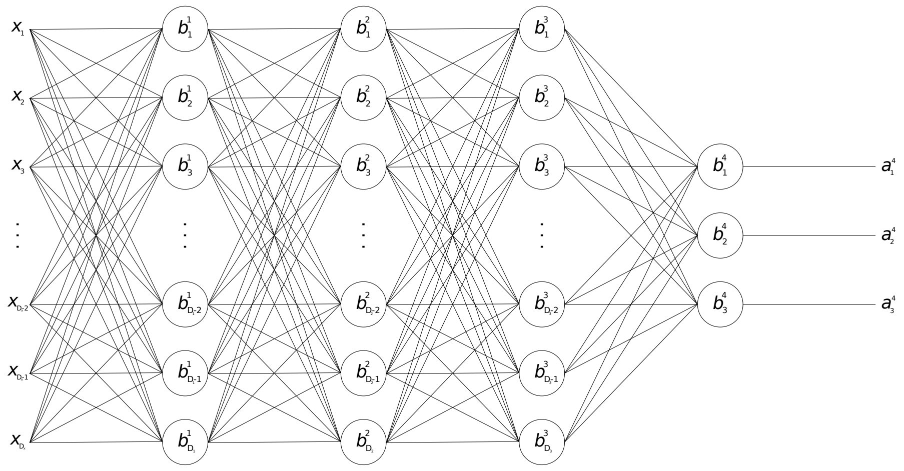


FIGURE 2.5: Multilayer Network: This is an example of a four layer network, the inputs and each layer have their own dimensionality, D_i, where i denotes the ith layer and the dimensionality of the input vector x is $D_x = M$.

Generalising Figure 2.5 requires a modification to the notation in Section 2.5.2. Specifically, the superscript $\boldsymbol{a}^i$ denotes that this is the output activations of the ith layer. Given this, the computation for a multi layer network of L layers each with their respective size $\{D_x, D_1, D_2, \ldots, D_L\}$ (D_x the dimensionality of the input vector x is $D_x = M$) can be expressed as:

$$\begin{aligned} a_j^1 &= \sigma(\boldsymbol{w}_j^1 \cdot \boldsymbol{x}) \quad \text{for } j = 1, \ldots, D_x \\ a_j^l &= \sigma(\boldsymbol{w}_j^l \cdot \boldsymbol{a}^{l-1}) \quad \text{for } j = 1, \ldots, D_l \text{ and } l = 2, \ldots, L \end{aligned} \tag{2.41}$$

where j denotes the jth neuron and l the lth layer.

Equation 2.41 describes a two step process. The first hidden layer is computed with the input layer, followed by a recurrence of the previous layers activations $\boldsymbol{a}^{l-1}$ to calculate the lth layers activations $\boldsymbol{a}^l$. The notation can be simplified to a matrix-based form for single x input vectors as follows:

$$\begin{aligned} \boldsymbol{a}^1 &= \Phi(W^1 \boldsymbol{x}) \\ \boldsymbol{a}^l &= \Phi(W^l \boldsymbol{a}^{l-1}) \quad \text{for } l = 2, \ldots, L \end{aligned} \tag{2.42}$$

where $\boldsymbol{a}^l$ denotes the lth layer activation vector and W^l the lth layers weight matrix. Finally, processing multiple input vectors in a batch is performed by:

$$\begin{aligned} A^1 &= \Phi(W^1 X) \\ A^l &= \Phi(W^l A^{l-1}) \quad \text{for } l = 2, \ldots, L \end{aligned} \tag{2.43}$$

where A^l is the l layers activation matrix for the set of inputs X.

2.5.4 Training

The objective of the training algorithm is to modify the weights such that an optimal network configuration is found. The goal is for the output of the final layer $\boldsymbol{a^L} = \{a_D^L, \ldots, a_0^L\}$ to match the target $\boldsymbol{y} = \{y_D, \ldots, y_0\}$ for the input training set. Similarly to the error function presented in Equation 2.30, the aim is to reduce the difference between the prediction $\boldsymbol{a^L}$ and the target $\boldsymbol{y}$. The error function can also be called the cost function. Given the mean squared error cost function for an individual input x is:

$$C = \frac{1}{2} \sum_{i=1}^{D} (a_i^L - y_j)^2 \tag{2.44}$$

The standard method to train neural networks is to use the backpropagation algorithm and gradient descent. Gradient descent is an iterative algorithm which updates the parameters of the model via a scaled gradient of the cost function, ∇C. It can be thought of as taking "steps" towards the optimal solution. For example, updating the weight w^l_{ij} involves

$$w^l_{ij} = w^l_{ij} - \nu \nabla C \tag{2.45}$$

where ν is the learning rate, a constant which controls how large a "step" is made.

The backpropagation algorithm is used to to calculate ∇C. Fundamentally, backpropagation is a measure of the rate of change of the cost function with respect to changes in the weights and biases. This involves computing the partial derivatives $\partial C/\partial w^l_{ji}$ and $\partial C/\partial b^l_j$, where j denotes the jth neuron, l the lth layer and i then ith weight in the neuron. To achieve this, the value δ^l_j, which is the error in the jth neuron in the lth layer, is calculated for the current input vector. Working from the final layer L, δ^L_j is calculated by:

$$\delta^L_j = \frac{\partial C}{\partial a^L_j} \sigma'(z^L_j) \tag{2.46}$$

This expression can be understood as the rate of change of the final layer activation $\frac{\partial C}{\partial a^L_j}$ and a measure of how fast the activation function σ' changes with respect to z^L_j, i.e. $\sigma'(z^L_j)$. In matrix-based form this is expressed as:

$$\delta^L = \nabla_a C \odot \sigma'(\boldsymbol{z}^L) \tag{2.47}$$

where $\nabla_a C$ is a vector of the partial derivatives of C with respect to a^L_j ($\frac{\partial C}{\partial a^L_j}$) and $\odot$ is the Hadamard product, the element-wise product of two vectors. Note that for the example cost function, Equation 2.44, $\nabla_a C$ can easily be calculated as the difference between the two vectors:

$$\nabla_a C = (\boldsymbol{a}^L - \boldsymbol{y}) \tag{2.48}$$

The error in any layer δ^l is now defined as:

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{2.49}$$

where $(W^{l+1})^T$ is the transpose of the weight matrix at the $(l+1)$th layer. Intuitively, this can be understood as a two step process. Firstly, $((W^{l+1})^T \delta^{l+1})$ moves the error from δ^{l+1} back through the network to the output of the lth layer. Secondly, $\odot\sigma'(z^l)$

moves the error through the activation function in layer l, resulting in the lth layers error δ^l.

With Equation 2.47 and Equation 2.49, the error in any layer can be calculated. Now, the two partial derivatives $\partial C/\partial w_{ji}^l$ and $\partial C/\partial b_j^l$ are calculated by:

$$\frac{\partial C}{\partial b_j^l} = \delta^l \tag{2.50}$$

and

$$\frac{\partial C}{\partial w_{ji}^l} = a_i^{l-1}\delta^l \tag{2.51}$$

The algorithm for training a neural network involves performing the feedforward step on an input vector $\boldsymbol{x}$ to compute $\boldsymbol{z}^l$ and $\boldsymbol{a}^l$ for each layer. Next, the output error is calculated $\delta^L = (\boldsymbol{a}^L - \boldsymbol{y}) \odot \sigma'(\boldsymbol{z}^L)$ and the error is backpropagated by calculating $\delta^l = ((W^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l)$ for each layer. The final step is to apply gradient descent to the weights and biases. Algorithm 2.1 illustrates the training algorithm.

Algorithm 2.1: Neural Network Training Algorithm

```
Data: $X$: Input Dataset
// For each Input Vector
1 for $k = 1$ to $N$ do
      // Compute $\boldsymbol{a}$ and $\boldsymbol{z}$ for each layer
2     $(\boldsymbol{a^l}, \boldsymbol{z^l}) = Feedforward(\boldsymbol{x}_k)$;
      // Compute $\boldsymbol{\delta^L}$
3     $\delta^L = (\boldsymbol{a}^L - \boldsymbol{y}) \odot \sigma'(\boldsymbol{z}^L)$;
      // Backpropagate the error to compute $\boldsymbol{\delta^l}$ for each layer
4     $\delta^l = ((W^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l)$;
// Perform Gradient Descent, updating the weights and biases for all
   input vectors
5 $w^l \to w^l - \frac{\nu}{N}\sum_{i=1}^{N} \boldsymbol{\delta}_i^l \cdot \boldsymbol{a}_i^{l-1}$;
6 $b^l \to b^l - \frac{\nu}{N}\sum_{i=1}^{N} \boldsymbol{\delta}_i^l$;
```

2.5.5 Neural Network Computation

So far this section has covered a formal description of neural networks, including the formulation of single to multi layered networks and a brief look at the training methodology. This section concerns itself with computing the neural network. Specifically, the computation required to evaluate Equation 2.42 and Equation 2.43, the equations describing the single input and multi input cases for multi layered neural network computation.

As noted above, Equation 2.42 and Equation 2.43 are expressed as matrix-vector and matrix-matrix multiplication problems. These matrix and vector operations are common amongst high performance and scientific computing algorithms, many requiring significant computational resources to complete in a timely manner. As a result, optimised Basic Linear Algebra Subroutine (BLAS) libraries have become the main workhorse for computing these algorithms [39]. As illustrated below, the routines offered in these libraries can be used to compute individual layers, both the single input case Equation 2.38 and batch case Equation 2.40.

BLAS functionally is categorised into sets of routines called "levels", each corresponding to the degree of complexity of the algorithms. The level 1 routines focus on vector-vector operations, the typical example 'axpy' computing:

$$\boldsymbol{y} \rightarrow \alpha \boldsymbol{x} + \boldsymbol{y} \tag{2.52}$$

where y and x are vectors and α a scalar constant. This first multiplies each element of x by the scalar α then computes the element-wise sum of αx and y. An important attribute of level 1 routines is that they typically take linear time, $O(n)$. $O()$ denotes Big-O notation which is an estimate of the time taken for running a particular routine.

Level 2 routines focus on matrix-vector operations, and typically take quadratic time $O(n^2)$. An example of a level 2 routine in the generalised matrix-vector multiplication (GEMV):

$$\boldsymbol{y} \rightarrow \alpha A \boldsymbol{x} + \beta \boldsymbol{y} \tag{2.53}$$

where A is a matrix, and β a scalar constant. Note that this is very similar to the single input case Equation 2.42. By substituting Equation 2.53 into Equation 2.38 and setting the appropriate constants, the GEMV can be used to compute a single layer.

$$\boldsymbol{a} = \Phi(1 \times W \boldsymbol{x} + 0 \times \mathbf{0}) \tag{2.54}$$

where $\alpha = 1$, $\beta = 0$, $A = W$ and $y = \mathbf{0}$ a vector of zeros since there is no need to perform accumulation.

Finally, the level 3 routines perform matrix-matrix operations. These take cubic time $O(n^3)$ and are the most costly of the three levels. General matrix multiplication (GEMM) is a key routine, performing a matrix-matrix multiplication [39]. The 'GEMM' routine is expressed as:

$$C \rightarrow \alpha AB + \beta C \tag{2.55}$$

where A, B and C are matrices, and α and β are scalar constants. Similar to the matrix-vector case, 'GEMM' can be substituted into Equation 2.40 to calculate a single layer for the batch input case.

$$A = \Phi(1 \times WX + 0 \times \emptyset) \tag{2.56}$$

where $\emptyset$ denotes a matrix of zeros, $\alpha = 1$ and $\beta = 0$.

2.5.6 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of neural networks with an additional layer, referred to as the convolution layer (CONV), designed for image-based problems. LeNet [40] was the first neural network to popularise convolutional neural networks for use in document recognition.

A convolution is performed by taking each filter and applying a dot product to sections of the input volume. After each application, the filter slides across the width and height by some stride; this creates a two dimensional activation map. This process is repeated for each filter, creating multiple two dimensional activation maps. These activation maps are stacked along the depth dimension, creating the output volume. Given some input I where the dimensions of the input are the height I_H, the width I_W and depth I_D. The dimensions of the output O are calculated by:

$$O_W = \frac{(I_W - F + 2P)}{S} + 1 \tag{2.57}$$

$$O_H = \frac{(I_H - F + 2P)}{S} + 1 \tag{2.58}$$

$$O_D = K \tag{2.59}$$

Where O_H is the output height, O_W the output width, O_D the output depth, K is the number of filters, F is the filter's height and width, S is the stride and P is the padding length. The standard method for computing the convolution layer is described in Algorithm 2.2.

The computation complexity (for inference) of the batched standard convolution is given by:

$$B \cdot I_D \cdot O_D \cdot O_H \cdot O_W \cdot F^2 \tag{2.60}$$

where B is the image batch size. While the different methods for computing the convolution layer are not necessary for understanding the contributions of this thesis, Appendix A contains a brief summary of the three most prominent techniques.

Algorithm 2.2: Standard Convolution

```
Data: $I$: Input Volume, $W$: Filters
Result: $O$: Output Volume
1  $O_W = \frac{(I_W - F + 2P)}{S} + 1$;
2  $O_H = \frac{(I_H - F + 2P)}{S} + 1$;
3  $O_D = K$;
4  for $i = 0$ to $I_D$ do
5      for $o = 0$ to $O_D$ do
6          for $j = 0$ to $O_W$ do
7              for $l = 0$ to $O_D$ do
8                  for $m = -F/2$ to $F/2$ do
9                      for $p = -F/2$ to $F/2$ do
10                         $j_I = Sj + m$;
11                         $l_I = Sl + p$;
                           // Negative Indices resolve to 0
12                         $O[o][j][l] += I[i][j_I][l_I] * W[i][o][m][p]$;
```

2.6 Summary

A brief background on FPGAs, digital arithmetic, machine learning and neural networks was presented is this chapter. It contained a discussion of FPGA designs, including the differences between heterogeneous architectures. The fundamentals of the fixed-point and floating-point number representations were outlined, and a description of the standard multiplication algorithm was presented. The core concepts of machine learning were introduced, using polynomial curve fitting as a motivating example. The chapter continued with a description of neural networks. It began with a mathematical description of the neuron and later expanded on these to a multilayer description. The chapter concluded by outlining the standard training algorithm, backpropagation, and described various computational techniques in addition to the convolution layer.

With this as a foundation, precision optimisations in the three distinct levels; the application, the framework and the operator, are now presented. The next chapter focuses on the application. A real-time radio-frequency anomaly detector is described, utilising a neural network autoencoder. The anomaly detector operates at a low precision, allowing the FPGA to implement more complex network topologies compared against traditional precision approaches. As a point of comparison, an on-line bitmap detection technique is presented and the resulting implementation evaluated against the autoencoder approach. Both detectors operate with high throughput and low latency, demonstrating superior performance to traditional standard techniques.

Chapter 3

Real-Time Radio-Frequency Anomaly Detection

3.1 Introduction

Anomaly detection, otherwise known as outlier detection, is the problem of identifying data which is not in accordance with expected behaviour. When processing data streams time is of the essence, as quickly making a determination is often times as important as making a decision. Processing high frequency signals in real-time is computational demanding due to the very high data rates involved. Real-time data mining and machine learning techniques have long been applied to fields ranging from forecasting financial markets to autonomous vehicles, adaptive processing and machine prognostics [41]. Specifically in the case of radio-frequency communication used in hostile environments, real-time anomaly detection is particularly important in identifying missing or tampered information. Although there has been considerable progress in addressing how to scale off-line systems to match the rapidly increasing quantities of data, real-time learning remains a challenge. New algorithms and computer architectures are needed as current implementations based on general-purpose computing are not able to process data with sufficiently low latency or high throughput. For any application, the premise of machine learning is the same: a computer system will learn to model future outcomes based on previously acquired data.

The following chapter presents two FPGA-based spectral anomaly detectors as an example of a reduced precision application. The first detector employs a bitmap technique to perform anomaly detection and uses a reformulation of the Discrete Fourier Transform (DFT) to compute the power spectrum in an on-line manner. When given a new input,

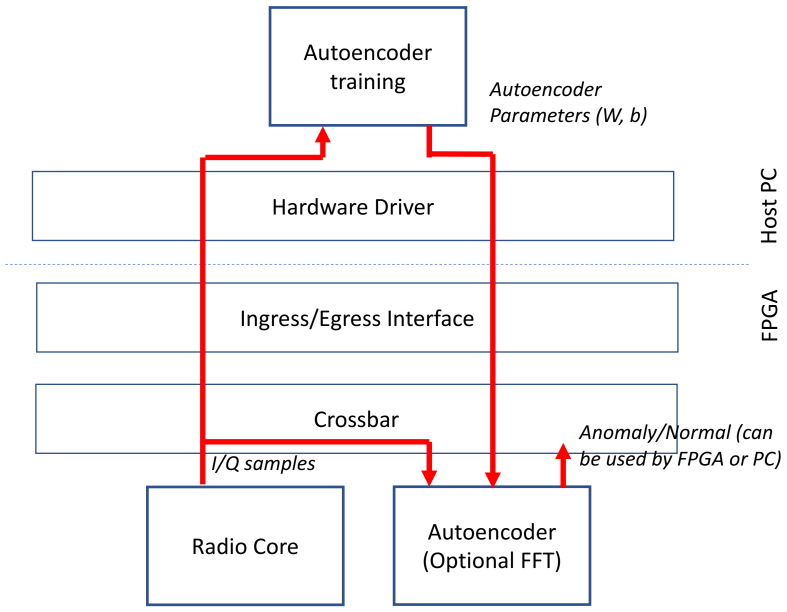


FIGURE 3.1: Block diagram of system implementation. The autoencoder module accepts the raw IQ data from the radio core.

the implementation is designed to update its state in time complexity $O(1)$. This is the asymptotic minimum complexity of any technique which considers all the input data.

As a point of comparison, a neural network anomaly detector is presented. There is a distinct lack of low latency neural network anomaly detectors in the area of radio-frequency signals. The detector in this chapter aims to address this by illustrating real-time anomaly detection through the utilisation of low precision neural network implementations. The detector, illustrated in Figure 3.1, is a neural network autoencoder designed for physical-layer radio-frequency signals. This thesis presents a fully pipelined implementation, utilising a reduced precision representation to ensure energy efficient and parallel computation. The implementation is integrated on the same chip as other processing hardware, able to achieve low latency and high throughput, producing an output every cycle. It is a heterogeneous architecture, supporting weight updates while inference is performed. This enables simultaneous training and inference with the former conducted on a processor or GPU.

The contributions of this chapter are:

- An on-line power spectrum computation technique for both regularly and irregularly-sampled time series data, reducing latency compared with the discrete Fourier transform and fast Fourier transform.
- An on-line formulation of the bitmap anomaly detection technique, resulting in an implementation that updates its state in time complexity $O(1)$.

- A single-chip implementation of a physical-layer RF neural network anomaly detector which is fully pipelined and can produce an output every cycle.
- The highest reported performance to date, supporting continuous 200 million samples per second (MS/s) complex inputs with latencies of 100 ns (time-domain) and 140 ns (frequency domain), at least 4 orders of magnitude lower than the processing time in GNU radio [42].
- A heterogeneous architecture which supports updating of weights while inference is performed; enabling simultaneous learning and inference with the former conducted on a processor or GPU.

3.2 Anomaly Detection

Expanding on the concepts of machine learning introduced in Chapter 2, anomaly detection is a type of machine learning aimed at identifying data samples that aren't in accordance with what is expected [41, 43–45]. Figure 3.2 is an example of a typical regression type anomaly [1]. The time period 0 to 1000 illustrates normal behaviour, whereas from time period 1000 to 2000 there is a deviation, hence signifying anomalous data. In this case the time series is tracking the position of a gun barrel being raised and lowered repeatedly; the anomaly occurs when the individual fumbles the gun, creating the position anomaly.

The method of anomaly detection used in this thesis, involves finding the difference between the current input vector $\boldsymbol{x}$ and the expected input vector $\bar{\boldsymbol{x}}$:

$$\bar{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{w}) \tag{3.1}$$

where $\bar{\boldsymbol{x}}$ is given by a function $f(\boldsymbol{x}, \boldsymbol{w})$ with coefficients $\boldsymbol{w}$:

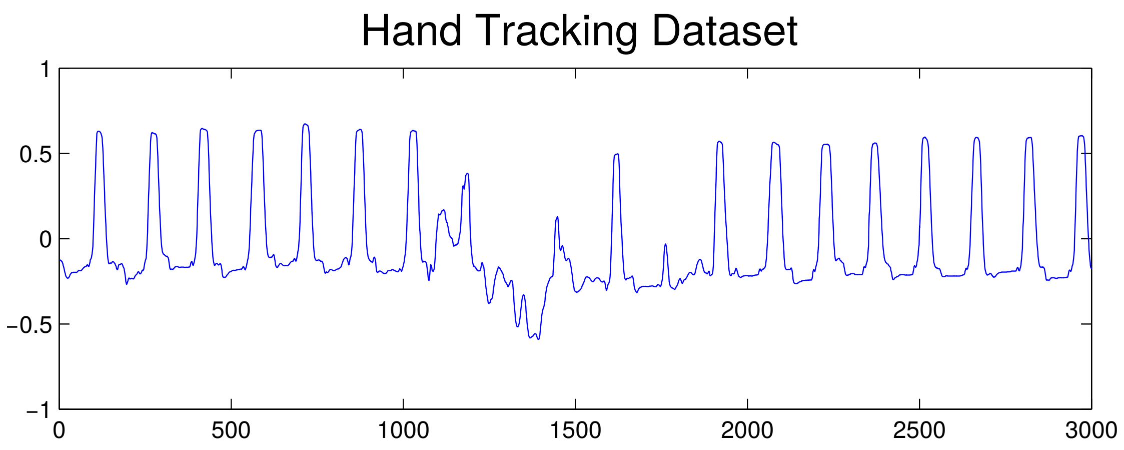


FIGURE 3.2: Anomaly Example: This is 2D hand tracking video [1]. The anomaly can clearly be seen from time period 1000 to 2000.

This technique of anomaly detection is classed as an *unsupervised learning* problem and does not need an expert to generate the targets $\boldsymbol{y}$ before training. The coefficients are learned on non-anomalous inputs, building a set of coefficients $\boldsymbol{w}$ that perform the identity mapping $\boldsymbol{x} \rightarrow \bar{\boldsymbol{x}}$. A difference function $d(\boldsymbol{x}, \bar{\boldsymbol{x}})$ determines the similarity or differences between the two vectors. A typical difference function for regression problems is the Euclidean distance between the two vectors:

$$d(\boldsymbol{x}, \bar{\boldsymbol{x}}) = \sqrt{\sum_{i=0}^{N} (x_i - \bar{x}_i)^2} \tag{3.2}$$

where N is the length of the input vector.

Using the difference function, data is determined to be anomalous if above a particular threshold l. In the case of a binary anomaly decision, the following is used:

$$anomalous = \begin{cases} 1, & \text{if } d(x, \bar{x}) > l \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

3.2.1 Discrete Fourier Transform

In some cases, It is advantageous to extract additional information to include in the input vector $\boldsymbol{x}$ [46]. One such method involves converting a signal into the frequency domain by sampling the signal and dividing it into its frequency components. Each of these components are expressed as a single sinusoidal oscillator with distinct frequencies, amplitudes and phases, and can be used as additional inputs in machine learning systems. This section provides a small background on the process of extracting the single sinusoidal oscillators.

The DFT, defined in Equation 3.5, is a complex-valued function of frequency.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j2\pi kn/N} \tag{3.4}$$

$$X_k = \sum_{n=0}^{N-1} x_n [cos(2\pi kn/N) - i * sin(2\pi kn/N)] \tag{3.5}$$

where x_n is an N-point time series ($n = \{0, 1, \ldots, N-1\}$) sampled at uniformly spaced points in time and k denotes the number of frequency bins, chosen such that $k = \{0, 1, \ldots, N-1\}$.

The computation complexity of computing Equation 3.5 is $O(N^2)$. A fast Fourier transform (FFT) is an algorithm that calculates the DFT using $O(N \log N)$ operations where

Algorithm 3.1: Radix-2 Decimation in Time FFT

```
1  Function $fft(x, N, s)$
       Data: $\{x_0, x_1, \ldots, x_N\}$, $N$, $s$
       Result: $\{X_0, X_1, \ldots, X_N\}$
2      if $N = 1$ then
3          $X_0 = x_0$;
4      else
5          $\{X_0, \ldots, X_{N/2-1}\} = fft(x, N/2, 2s)$;
6          $\{X_{N/2}, \ldots, X_N\} = fft(x + s, N/2, 2s)$;
7          for $k = 0$ to $N/2 - 1$ do
8              $t = X_k$;
9              $X_k = t + X_{k+N/2} * e^{-j2\pi k/N}$;
10             $X_{k+N/2} = t - X_{k+N/2} * e^{-j2\pi k/N}$;
```

N is a power of 2. The most commonly used FFT is the Cooley-Tukey FFT algorithm [47]. Algorithm 3.1 illustrates the recursive method for computing the radix-2 (decimation in time).

After performing the DFT, by either using an FFT algorithm or directly from the Equation 3.5, the resulting complex-valued frequencies can be used as additional inputs in an anomaly detection system.

3.2.2 Previous RF Anomaly Detector and FPGA-based Detectors

There has been considerable recent interest in utilising advances in neural network technology for RF anomaly detection applications. The most relevant to this thesis is by O'Shea et. al [48], who applied a number of anomaly detectors to RF signals. Starting with Frequency Modulation (FM), Global System for Mobile Communications (GSM), industrial, scientific, and medical radio (ISM), and long-term evolution (LTE) band data; Short-time Broadband Bursts, Brief Periods of Signal Non-Linear Compression, Pulsed QPSK Signals and Pulsed Chirp Events were introduced as anomalies. A comparison of a number of algorithms: a 3rd order Unscented Kalman Filter/Predictor, dense neural network, long short-term memory and Dilated Convolutional Neural Network was conducted and it was observed that in most cases, the neural network approaches outperform the Kalman-based approach. This work did not address the issue of real-time implementation. Latency in GNU radio/Universal Software Radio Peripheral platforms (USRP - they measured USRP1 and USRP2 whereas the device used in this thesis is a newer USRP3) was thoroughly analysed by Trong et. al [42]. They concluded that the time for processing at the host computer dominates the communication bus latency and measured values well in excess of 1 ms.

Other examples of anomaly detection in the reconfigurable computing literature is network intrusion detection. This requires operation at network line speeds, where the advantages of FPGAs are clear. Das et. al [49] proposed a system based on feature extraction and Principle Component Analysis (PCA) to identify anomalies. Their architecture could support data at over 20 Gbps. The PCA part of the system was performed in an off-line manner.

Carter et. al. [50] proposed an Exponentially Weighted Moving Average (EWMA) method where an anomaly is detected when the absolute value of the difference between the local mean and the input data sample exceeds the estimated standard deviation times a constant multiplier. This results in an effective and robust method that quickly adapts to distributional data shifts. It has update time complexity $O(1)$, making it suitable for a low latency, high throughput implementation. Unfortunately, it only considers the simple mean value and cannot track changing sequences of patterns in the data.

3.3 On-line Learning for Power Spectral Detection

This section focuses on the bitmap anomaly detection technique. An on-line formulation of this technique is presented and evaluated on the physical-layer radio-frequency signals. It provides high throughput and low latency anomaly detection at a reduced hardware resource cost. The bitmap detector uses the power spectrum calculated using a reformulation of the DFT for irregularly sampled time series. As with the frequency spectrum, the power spectrum is often used in applications including machine prognostics [51], astronomy [52] and computational finance [53]. In some applications, it is either impossible to sample at uniform intervals, e.g. astronomical observations [52], or data points are sampled at irregularly spaced intervals e.g. financial tick data [53].

3.3.1 Power Spectra of Irregularly Sampled Time Series

The standard tool for calculating power spectral density is the DFT. For an N-point time series $x_n, n = \{0, 1, \ldots, N-1\}$ sampled at uniformly spaced time points, the DFT is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j2\pi kn/N} \tag{3.6}$$

This section presents the reformulations of existing power spectra and anomaly detection algorithms to facilitate efficient FPGA-based spectral anomaly detection implementations. Though computing Equation 3.6 for frequency bins $k = \{0, 1, \ldots, N-1\}$ is most common, the frequency resolution can be arbitrarily chosen which is particularly useful for the purpose of estimating power spectra. Where the frequency domain resolution of M bins is uniformly distributed across the frequency range of $\omega \in (-\pi, \pi)$ is $\omega = 2\pi k/N($ in radians per sample).

As a function of ω, Equation 3.6 can be rewritten as:

$$X(\omega) = \sum_{n=0}^{N-1} x_n e^{-j\omega n} \tag{3.7}$$

The periodogram or normalised power for frequency ω is commonly computed from the squared magnitude of $X(\omega)$:

$$P(\omega) = |\frac{1}{N} X(\omega)|^2 \tag{3.8}$$

For the generalised univariate time series $x_n = x(t_n)$ with arbitrarily spaced but strictly increasing sampling times t_n, Equation 3.7 may be used by replacing the time index n in complex exponent by the samples observation time t_n:

$$X(\omega) = \sum_{n=0}^{N-1} x_n e^{-j\omega t_n} \tag{3.9}$$

where ω is in radians per unit time t_n. Now computing over uniformly spaced frequency bands $\omega = 2\pi j/M$, and $j = \{0, 1, \ldots, (M-1)\}$, the power based on Equation 3.9 is known as the *classical Fourier periodogram.*

To further reduce computational and bookkeeping costs, an exponentially decaying weight is applied, replacing the fixed-width sliding window.

$$X(\omega) = (1-\gamma) \sum_{n=0}^{N-1} \gamma^{N-1-n} x_n e^{-j\omega t_n} \tag{3.10}$$

where γ is the exponential weighting factor $\gamma \in (0, 1)$. Given a particular choice of sample half-life λ (in number of samples), γ is related to λ by $\gamma^\lambda = \frac{1}{2}$.

As the number of samples N increases towards infinity, the sum of exponential weights will converge to:

$$\lim_{N\to\infty} \textstyle\sum_{n=0}^{N} \gamma^{N-n} = \frac{log_e \gamma}{\gamma - 1} \int_0^\infty \gamma^n dn = \frac{1}{1-\gamma} \tag{3.11}$$

and hence the normalisation by its inverse $(1-\gamma)$.

In the case of computing Equation 3.10, it is desirable to have the power updated as soon as a new sample is available with the least amount of computation possible. The DFT at the time of sample number N with adjusted time reference is:

$$X(\omega, N) = (1-\gamma)\sum_{i=0}^{N} \gamma^{N-i} x_i e^{-j\omega(t_i - t_N)} \tag{3.12}$$

Using the latest sample time t_N as the zero time reference is intuitive for on-line data streaming and this phase shift has no effect on the magnitude of power.

It is quite simple to manipulate and express Equation 3.12 as a recursive function of its previous value at $N-1$:

$$\begin{aligned} X(\omega, N) &= (1-\gamma)\sum_{n=0}^{N-1} \gamma^{N-i} x_n e^{-j\omega(t_n - t_N)} + \\ &\quad (1-\gamma) x_N e^{-j\omega(t_N - t_N)} \\ &= X(\omega, N-1)\,\gamma\, e^{j\omega(t_N - t_{N-1})} + \\ &\quad (1-\gamma) x_N \end{aligned} \tag{3.13}$$

The phase shift is based on the time elapsed after the previous update, $\Delta t = t_N - t_{N-1}$ and does not depend on any absolute time reference, updates require $O(1)$ computation.

The power is then computed by:

$$P(\omega, N) = |X(\omega, N)|^2 \tag{3.14}$$

Note that while M evenly spaced frequencies is used in this work, the algorithm allows for the selection of an arbitrary number of frequencies. Non-uniform spacings may be advantageous in some applications.

Compared with the classical periodogram, Least-Squares Spectral Analysis (LSSA), otherwise known as the Lomb-Scargle periodogram, generally produces better and more accurate power spectra by including compensation terms, which reduce the effect of global aliasing due to non-uniform sampling [52, 54]. However, Scargle also stated that the Lomb-Scargle periodogram typically does not differ significantly from the classical periodogram [52]. Since the full Lomb-Scargle periodogram is approximately 3-4 times more expensive to compute, and its output is subsequently discretised into a small number of symbols, it is safe to assume that the classical periodogram will be a sufficient approximation.

3.3.2 Bitmap Anomaly Detection

The bitmap detector in this thesis is based on an algorithm proposed by Kumar et. al. [55]. The numerical input is first quantised to a discrete representation. While Kumar et. al. used Symbolic Aggregate approXimation (SAX) [2], which generates symbols that are approximately equiprobable, this requires two passes through the data. This implementation processes the data in a single pass by taking the most-significant b bits of the data. The number of symbols is determined by 2^b, in the case of $b = 2$, a 4 symbol alphabet is used, represented by the symbols a,b,c and d. The quantisation and signal string process is illustrated in Figure 3.3, resulting in *ccbabcd*.

Once the signal string has been extracted, bitmaps are then constructed from the time series of symbols in a window of size W. The frequency of all contiguous sequences of length-d symbols (i.e. all d-grams) is calculated, and used as entries in a bitmap represented by a d dimension array, of 2^b dimensional size. The number of symbol transitions is calculated directly by $W - (d - 1)$. For the symbol signal in Figure 3.3, $d = 2$ as represented by the length two symbol transitions, and $b = 2$, the bitmap size is a 4×4 array illustrated in Figure 3.4.

A bitmap contains the frequency of the symbol transition, each element in the bitmap is initialised to 0 and accumulated by 1 for each symbol transition. Transitions are order

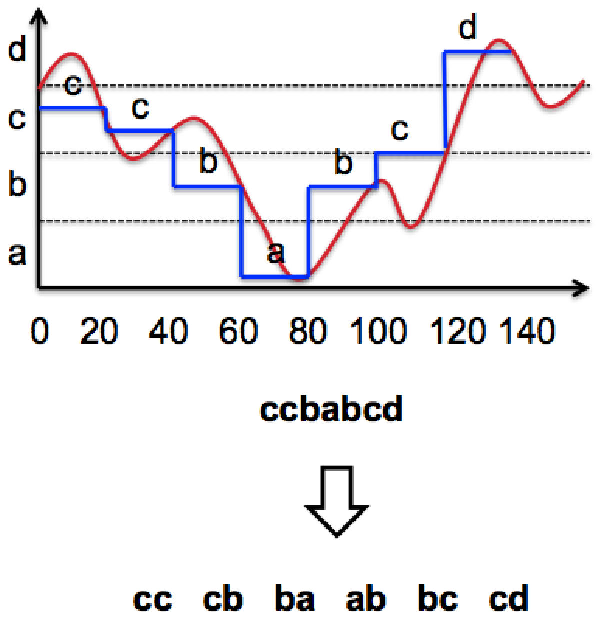
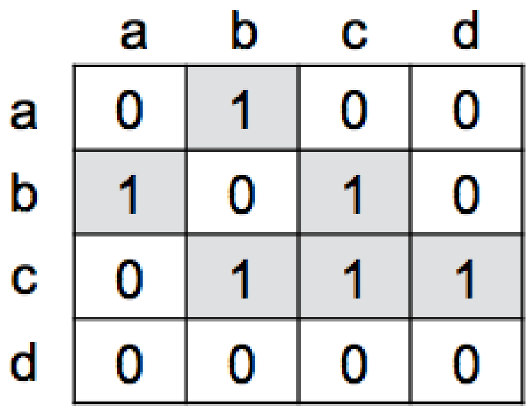


FIGURE 3.3: Signal Quantisation: In this example the signal is quantised based on the two most-significant bits of the input. This creates four separate bands which identify the signal's symbols when sampled at a particular point in time. For this example signal, the input is sampled every 20 time steps, resulting in the signal string *ccbabcd*. Finally, the symbol transitions are extracted from the signal string [2].



	a	b	c	d
a	0	1	0	0
b	1	0	1	0
c	0	1	1	1
d	0	0	0	0

FIGURE 3.4: Signal Bitmap: This contains the frequency count for $d = 2$ and the signal *ccbabcd*.

dependent with the first symbol denoting the first dimension of the bitmap, symbol two denoting the second dimension, and so on until dimension d. Hence, *cb* and *bc* increment two different locations in the bitmap. Figure 3.4 illustrates the resulting bitmap for all adjacent pairs in Figure 3.3's symbolic signal, the entries contain the frequency for $d = 2$.

Performing anomaly detection using the bitmap technique results in a comparison between two windows. Detector and reference bitmaps, B_D and B_R, are calculated for two different window sizes, W_D and W_R respectively. Typically, W_R is significantly larger than W_D as the aim of B_R, the reference bitmap, is to contain a large portion of non-anomalous signal. Comparing the two bitmaps involves taking a normalised sum of squared differences and arriving at a final score; the relative difference between the two. This is illustrated in Equation 3.15, Each bitmap can be unrolled and represented as a single dimension vector of length $2^b \times d$.

$$s = \sum_{i=0}^{2^b \times d} \left(\frac{B_R[i]}{W_R - (d-1)} - \frac{B_D[i]}{W_D - (d-1)}\right)^2. \tag{3.15}$$

Each frequency count is normalised by dividing through by the total number of symbol transitions in the bitmap, $W_R - (d-1)$ and $W_B - (d-1)$ for the reference and detector respectively, to allow for the two differently sized windows to be compared. Hence, A symbol transition, SYM, is calculated by concatenating the bits of the d subsequent quantised values to create the index, illustrated in Equation 3.16.

$$SYM = cat(Q^j_{i-d}, \ldots, Q^j_{i-1}, Q^j_i). \tag{3.16}$$

As new inputs are received, both the detector and reference windows are slid by one value, taking in the new input and removing the oldest value. This is often referred to as a sliding window, with the detector and reference overlapping on the most recent values. The reference window typically extends further back in time, amortising the effect of any anomalous data.

3.3.3 Algorithm & Implementation Details

This algorithm is a result of combining the power spectra computation and the bitmap anomaly detection techniques, described in Algorithm 3.2. Each channel of the power spectra requires two memories, M_R and M_B, that contains the symbols for its own reference and detector bitmaps respectively. For each input sample x, a vector of the previous DFT X_{i-1}, the time elapsed since the last input δt and the previous quantisation vectors $Q_{i-d-1}, \ldots, Q_{i-1}$ is required.

The anomaly detection is conducted over C channels, resulting in C scores s_j ($j = 0 \ldots (C-1)$). The consolidated anomaly score is simply

$$a = \frac{1}{M} \sum_j s_j \tag{3.17}$$

and an anomaly is detected if

$$anomaly = \begin{cases} \text{TRUE} & a > l \\ \text{FALSE} & otherwise \end{cases} \tag{3.18}$$

where l is a user-defined threshold.

This allows for detection of anomalies resulting from subtle changes in particular frequencies. While summation was used to aggregate anomaly scores over the channels, many alternative techniques are available. For example, the max function could result in more sensitive detection.

To optimise the algorithm for an efficient FPGA implementation, Step 4 can be simplified by replacing the *for loops* with another two memories containing B_R and B_D, only updating the bitmap with the changed value. Each bitmap undergoes two updates, the incoming symbol SYM and the outgoing symbols $M_R[W_R - (d-1)]$ and $M_D[W_D -$

Algorithm 3.2: On-line Channel Independent Spectra Anomaly Detection

```
// Called for each input sample, $x$, $i$ represents the increment in time.
Data: $x$, $\delta t$, $X_{i-1}$, $Q_{i-1} \dots Q_{i-d-1}$
Result: $S$, $X_i$, $Q_i$
1  for $j = 0$ to $C$ do
       // Step 1: Calculate the Power Spectra
2      $X_i^j = X_{i-1}^j \gamma e^{j\omega_j \Delta t} + (1-\gamma)x$;
3      $P = |X_i^j|^2$;
       // Step 2: Quantise and Symbol
4      $Q_i^j = sr(P, L - \log_2 - 1)$; // $sr$:right shift
5      $SYM = cat(Q_{i-d-1}^j, \dots, Q_{i-1}^j, Q_i^j)$; // $cat$:concatenation function
       // Step 3: Update Window Memories
6      $M_R^j[1 : W_R - (d-1)] = M_R^j[0 : W_R - (d-2)]$;
7      $M_D^j[1 : W_D - (d-1)] = M_D^j[0 : W_D - (d-2)]$;
8      $M_R^j[0] = SYM$;
9      $M_D^j[0] = SYM$;
       // Step 4: Create the Bitmaps
10     for $k = 0$ to $W_R - (d-1)$ do
11         $B_R[M_R^j[k]] += 1$;
12     for $k = 0$ to $W_D - (d-1)$ do
13         $B_D[M_D^j[k]] += 1$;
       // Step 5: Calculate the Score
14     $S^j = \sum_{k=0}^{2^b \times d} (\frac{B_R[k]}{W_R-(d-1)} - \frac{B_D[k]}{W_D-(d-1)})^2$;
```

$(d-1)]$ for the reference and detector respectively. Similarly, Step 5 can be optimised by updating a running score using only the changed elements of the bitmaps. Given the three changed symbol transitions, SYM, $M_R[W_R]$ and $M_D[W_D]$, the update is performed in three parts:

1. **Score Update:** removes the bitmap values associate with the three changing symbol transitions.
2. **Bitmap Update:** updates the bitmaps
3. **Score Update:** accumulates back the updated bitmap values associated with the 3 changing symbol transitions.

Finally, to support the new optimisations, Algorithm 3.2 needs to be reorganised so that updating the window memories is performed last. The new algorithm is shown in Algorithm 3.3 along with an illustration of the hardware blocks in Figure 3.5.

The implementation is parameterised according to Table 3.1, any of which can be modified at compile time. Since the M frequency components can be computed independently,

Algorithm 3.3: Optimised On-line Channel Independent Spectra Anomaly Detection

```
// Called for each input sample, $x$, $i$ represents the increment in time.
Data: $x$, $\delta t$, $X_{i-1}$, $Q_{i-1} \dots Q_{i-d-1}$, $S$
Result: $S$, $X_i$, $Q_i$
1  for $j = 0$ to $C$ do
       // Step 1: Calculate the Power Spectra
2      $X_i^j = X_{i-1}^j \gamma e^{j\omega_j \Delta t} + (1-\gamma)x$;
3      $P = |X_i^j|^2$;
       // Step 2: Quantise and Symbol
4      $Q_i^j = sr(P, L - \log_2 -1)$; // $sr$:right shift
5      $SYM = cat(Q_{i-d-1}^j, \dots, Q_{i-1}^j, Q_i^j)$; // $cat$:concatenation function
       // Step 3: Update Score Part 1
6      $S^j - = (\frac{B_R^j[SYM]}{W_R-(d-1)} - \frac{B_D^j[SYM]}{W_D-(d-1)})^2$;
7      $S^j - = (\frac{B_R^j[M_R^j[W_R]]}{W_R-(d-1)} - \frac{B_D^j[M_R^j[W_R]]}{W_D-(d-1)})^2$;
8      $S^j - = (\frac{B_R^j[M_D^j[W_D]]}{W_R-(d-1)} - \frac{B_D^j[M_D^j[W_D]]}{W_D-(d-1)})^2$;
       // Step 4: Update Bitmap
9      $B_R^j[SYM] + = 1$;
10     $B_D^j[SYM] + = 1$;
11     $B_R^j[M_R^j[W_R]] - = 1$;
12     $B_D^j[M_D^j[W_D]] - = 1$;
       // Step 5: Update Score Part 2
13     $S^j + = (\frac{B_R^j[SYM]}{W_R-(d-1)} - \frac{B_D^j[SYM]}{W_D-(d-1)})^2$;
14     $S^j + = (\frac{B_R^j[M_R^j[W_R]]}{W_R-(d-1)} - \frac{B_D^j[M_R^j[W_R]]}{W_D-(d-1)})^2$;
15     $S^j + = (\frac{B_R^j[M_D^j[W_D]]}{W_R-(d-1)} - \frac{B_D^j[M_D^j[W_D]]}{W_D-(d-1)})^2$;
       // Step 6: Update Window Memories
16     $M_R^j[1 : W_R - (d-1)] = M_R^j[0 : W_R - (d-2)]$;
17     $M_D^j[1 : W_D - (d-1)] = M_D^j[0 : W_D - (d-2)]$;
18     $M_R^j[0] = SYM$;
19     $M_D^j[0] = SYM$;
```

the computation complexity can range from $O(1)$ in which all channels are processed in parallel, to $O(M)$ when they are processed one channel at a time.

In the implementation, several further optimisations are applied:

1. Computation of the exponential $e^{j\omega_j \Delta t}$ is avoided by using a 512-entry look-up table.
2. All computation is done in fixed-point with wordlength L. In this case 16 bits was chosen to compare with the neural network anomaly detector.

Table 3.1: Default parameters for the bitmap anomaly detector implementation.

Parameter	Description	Default
L	Global wordlength	16
γ	Recursive DFT decay rate	0.995
W_D	Detector window length	9
W_R	Reference window length	33
M	Number of frequency channels	4-32
b	Number of symbols in alphabet	8
l	Anomaly threshold	0.5
d	Bitmap level	2

3. The score computation and bitmap updates are performed. This computation requires six addition/subtractions and six multiplications for each of the three changed indices.

4. Combinations of the L, b and d parameters are chosen so that the required number of bits, $L \times b^d$, is less than the size of a block RAM on the FPGA.

This implementation was created in Xilinx Vivado HLS 2013.4, targeted at a Xilinx Virtex 7 XC7VX690TFFG1930-3. It employs reduced precision at multiple stages; the on-line DFT, the quantisation and the bitmap update steps.

3.4 Bitmap Detection Evaluation

This section describes the resource utilisation, performance and accuracy of the bitmap detector. Firstly, identifying an anomaly in repeated hand motions in a 2D video [1]

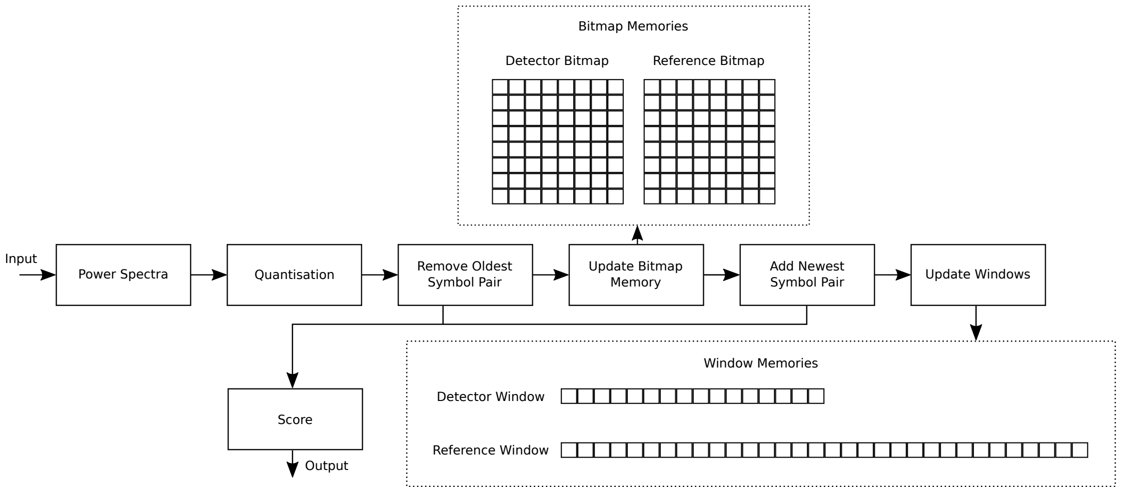


Figure 3.5: Overview of the Spectra Anomaly Detector: Each block represents a step from Algorithm 3.3.

is shown. Secondly, the detector is evaluated on the synthetic and real FM data signals and compared against the neural network detector. The anomalies were injected into the signals in the same manner as Section 3.6.1. Unless specified otherwise, the parameters were determined by a grid search over a range of values and the accuracy was determined by comparing detected anomalies with anomalies known a-priori. If two configurations produced very similar anomaly scores, the configuration which reduced the computational requirements was chosen.

3.4.1 Detector Accuracy Example

As an example of bitmap anomaly detection technique, Figure 3.6 shows a selected subset of the time series along with the aggregated anomaly score. The parameters for the 2D video hand tracking time series were $M = 16, \gamma = 0.99, W_R = 600, W_D = 100$, $d = 2$ and $b = 8$. The anomalous behaviour between 1000 and 2000 clearly causes an increased anomaly score and results in an easily identifiable anomaly. A more thorough treatment of the detectors performance in regards to RF signals is proved at the end of the chapter with a comparisons to the neural network detector.

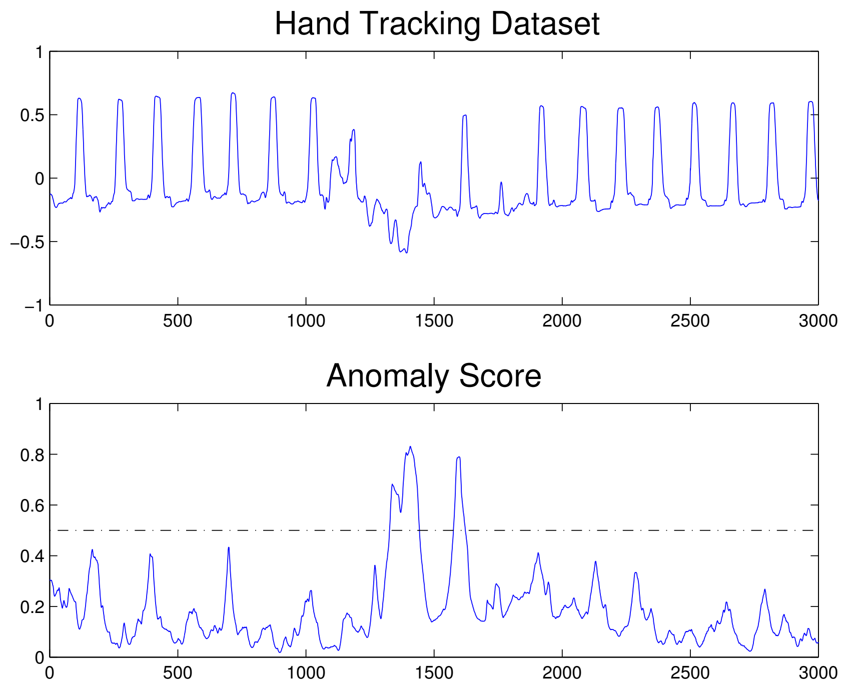


FIGURE 3.6: Hand tracking in a 2D video. The raw data is shown as well as an aggregated anomaly score.

Table 3.2: FPGA bitmap anomaly detector implementation compared with a C implementation, disregarding input/output. Times reported are to process a single data input.

M	Throughput	Latency	CPU Time	Throughput Speedup	Latency Reduction
1	40 ns	68 ns	34 ns	0.9×	0.5×
4	40 ns	68 ns	273 ns	7×	4×
8	40 ns	68 ns	544 ns	14×	8×
16	40 ns	68 ns	1085 ns	27×	16×
256	40 ns	68 ns	17969 ns	449×	264×

Table 3.3: Resource utilisation for $M = 1$. Numbers in parentheses indicate the available resources on the chosen FPGA.

LUTs	Flip Flops	BlockRAM	DSP
220 (433200)	541 (866400)	3 (2940)	14 (3600)

3.4.2 Resource Utilisation & Performance

The default parameters as summarised in Table 3.1 were used to create an $M = 1$ design. This was synthesised to obtain the FPGA resource utilisation summarised in Table 3.3. Vivado reported an initiation interval of 10 clock cycles, a latency of 17 cycles and a maximum frequency of 250 MHz. Projections for the performance of multiple channels are given in Table 3.2. Since multiple channels are processed and extracted from a single input vector in parallel, the throughput of the entire system is constant regardless of the number of channel processed.

The datapath for the design requires a total of six L bit fixed-point multipliers, nine subtracters, and seven adders, and is independent of the other parameters. The total memory in bits required for the design is $2MLb^d + L(W_R + W_N)$. The first term accounts for the reference and detection bitmaps, and the second for window buffers.

As can be seen from Table 3.3, DSP resources restrict the maximum number of parallel designs to approximately 256. However, designs can go beyond this value using look-up table (LUT) resources. The next limit is BRAMs which restrict M to approximately 1000.

The same C implementation used to synthesise the FPGA design was compiled using gcc version 4.6.3 with the -O3 compiler flag. Execution speed was tested on a 1.6 GHz Intel Core i5 Sandy Bridge processor with 4 GB of memory, 3 MB cache and using the Mac OSX Mavericks Operating System. CPU execution time was recorded by appropriately instrumenting the program using the high resolution Linux timer. Both FPGA and CPU results do not include input/output overheads and the effect of other peripheral devices has been taken into account. For the CPU benchmarks the data was generated and stored in the cache, for the FPGA benchmarks it was stored in the block memories. The low memory access time and higher clock frequency allowed the CPU to outperform the FPGA for the single channel case. However when multiple channels are considered the CPU requires each channel to be processed iteratively resulting in the significant speed demonstrated by the FPGA.

The Vivado Power Report estimates the $M = 1$, allowing for a direct comparison to the CPU, design running at 250 MHz to have a power consumption of 0.3 W, 99% of this being static power. In comparison, the same processor used for the speed tests draws 11.25 W, making the FPGA approximately 37.5$\times$ more power efficient. The CPU requires 34 ns which is similar to that of the FPGA, making the energy efficiency approximately 33.75$\times$ better. For larger values of M, energy efficiency is significantly higher.

3.5 Neural Network Frequency Domain Detection

As a point of comparison, a neural network anomaly detector designed for physical-layer radio-frequency signals is presented in this section. Over recent years, neural networks have achieved results surpassing all other approaches on difficult pattern recognition problems such as image analysis, speech recognition and machine translation. While previous work has demonstrated the utility of applying neural networks to RF applications, little has been published on their real-time implementation. Neural network algorithms are massively parallel in nature, and amenable to computation using low-precision; making them very suitable for efficient digital implementations.

3.5.1 Design Considerations

Performing anomaly detection on physical-layer radio-frequency signals presents the following challenges. Firstly, anomalies can appear in each frequency bin, however their effects might not be apparent until non-linear combinations of the respective frequencies are taken into account. Secondly, seasonality and the signal's history plays an important role in determining future predictions; the choice of window size often dictates the performance of the detector if long term effects need to be considered. Finally, the operating precision needs to be selected thoughtfully; without sufficient precision, small anomalous events can be lost either due to rounding or truncation in the number representation.

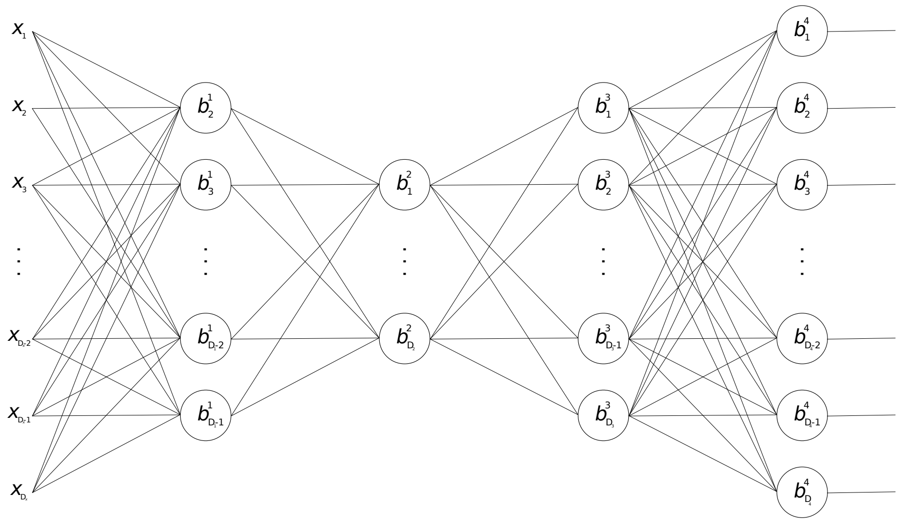


FIGURE 3.7: Simplified 4-layer autoencoder signal flow diagram.

Hence, the neural network detector was designed to address these issues in the following ways:

- All frequency channels are considered in the one model, allowing for non-linear combinations of the individual frequencies.
- The neural network is not restricted to a sliding window size; it is built from all historical data. This addresses limitations in prediction accuracy resulting from either an insufficient model size, shifts in signal pattern or historical anomalies not present in the model's current history.
- A range of different precision experiments and implementations were conducted and constructed to identify limitations in prediction accuracy and detector hardware performance.
- The weights are improved over time as an off-line training step is performed on a subset of the most recent input data. The weights used for inference are then updated using the new weights from this training step.

3.5.2 Autoencoder

An autoencoder is a multilayer neural network, constructed in two parts: an encoder and decoder. The encoder starts with the original inputs and passes it through successively smaller layers. The final layer of the encoder stage is overlapped with the first layer of the decoder. The decoder takes the encoded input back to its original size through another group of successive layers.

Autoencoders are generally symmetrical, an example autoencoder is illustrated in Figure 3.7 for a 4-layer network. The inputs are given by the vector $x \in \mathbb{R}^{D_x}$, and the outputs are $\hat{x} \in \mathbb{R}^{D_x}$, where D_x is the length of the input vector. It is trained on the identity mapping, striving to make the inputs and outputs match, enabling *unsupervised* training. Each layer implements the vector function $F^l : \mathbb{R}^{D_l} \to \mathbb{R}^{D_{l+1}}$ (D_l is the number of inputs of the layer):

$$a^l = \Phi(W^l a^{l-1} + b^l) \tag{3.19}$$

where L is the number of layers, $0 \le l \le L$ the layer number, $W^l \in \mathbb{R}^{D_{l+1} \times D_l}$ the weight matrix, $b^l \in \mathbb{R}^{\gtrdot}$ the bias and Φ is the activation function applied element-wise, which is the identity function for the 0'th and final layers and the element-wise rectified linear unit ($relu(x) = \max(0, x)$) otherwise.

Dimensionality reduction is performed since the dimensionality of the middle layers are fewer than the input/output. This is illustrated in Figure 3.7 with the first two layers acting as an encoder which compresses the input to 1-dimension. Similarly, the final two layers act as a decoder by taking the compressed representation and reproducing the input. The mean squared error is used as the cost function, however any type of neural network training scheme can be used to minimise the reconstruction error, the experiments described in this thesis use standard backpropagation to train the network [56].

3.5.3 System Architecture & Implementation

Referring back to Figure 3.1, The host computer is responsible for training; updating the neural network weights to ensure that the model remains relevant. The training routine operates in a batched manner, waiting for 1024 input samples before running the new batch along with the weights of the previous model. Initial and/or updated biases and weights thus computed are downloaded to the FPGA which implements the autoencoder, operating independently of the host computer. Training isn't performed on the FPGA because the anomaly detector implementations use the majority of the FPGAs resources, as demonstrated later in Table 3.4. The hardware autoencoder operates at the baseband sampling rate, maximising the probability of detecting transient anomalies, and ensuring the system is able to react as soon as possible. Training on the computer allows the anomaly detector to adapt to changing environmental conditions, while avoiding the problem of training on the FPGA, which would impact performance. While this implementation focuses on RF signals, it is relevant to the general problem of real-time neural network processing and can be generalised to other application domains and neural network architectures.

Figure 3.8 shows a block diagram of the fully pipelined spectral anomaly detection architecture. The design supports a 200 MS/s sampling rate, producing an input and

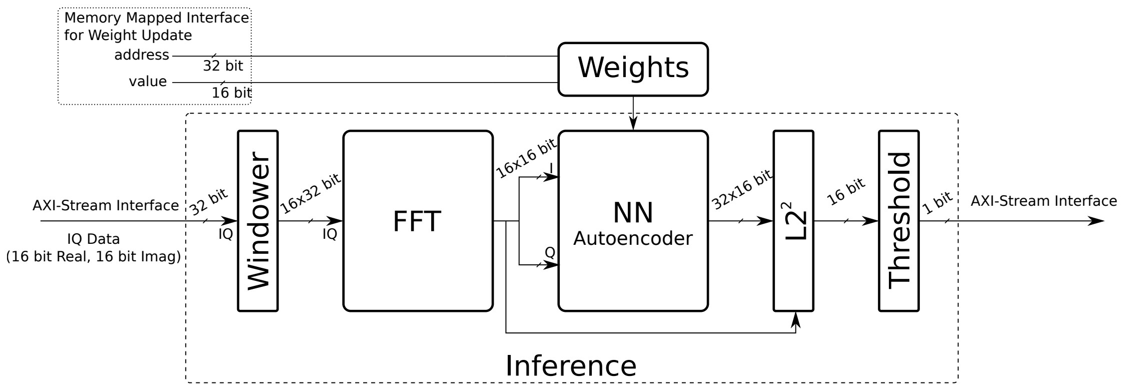


FIGURE 3.8: Block diagram of the anomaly detector FPGA implementation.

output every cycle. The architecture consists of five or four stages depending on the configuration. The first stage of this design is the Windower, which accepts complex I/Q samples from the source and uses a shift register to produce a sliding window of past inputs to the next stage. This is followed by an optional FFT stage, enabling anomaly detection in either the time or frequency domain. The FFT is implemented using a 4-stage radix-2 based algorithm derived from reference [57], and described in Section 3.2.1. It is important to note that for efficient hardware implementations, the twiddle factors, $e^{-j2\pi k/N}$, are calculated ahead of time with the values stored in an on-chip look-up table. The real and imaginary outputs of the FFT/Windower are concatenated to form the inputs of the autoencoder, which is trained to reconstruct an input window after performing dimensionality reduction. If anomalies occur, the autoencoder will be unable to reconstruct the input. Comparing a threshold to the squared L_2-norm $L2^2(x,\hat{x}) = \sum_{i=0}^{N}(x_i - \hat{x}_i)^2$ between the input and output of the autoencoder, yields a binary anomaly/normal determination.

3.5.4 Precision

All values are represented using 16 bit fixed-point numbers with the neural network weights employing truncated rounding and saturating arithmetic [58]. Accumulation is performed using 32 bit fixed-point to provide extra integer bits, avoiding potential overflow. The resulting values are converted back to 16-bit, with saturation performed if needed. This maps efficiently to the FPGAs digital signal processing (DSP) blocks, which includes dedicated 25x18 bit multiply-accumulate circuitry. The FFT block operates with complex inputs and outputs, while the autoencoder uses real values by stacking both real and imaginary parts of the FFT output into a single real valued vector. This relatively high precision for inference ensures that the fixed-point results will achieve similar accuracy to floating-point [59].

Table 3.4: Resource Utilisation for Varying Precisions

Precision	II	Latency (cycles)	BRAM	DSP	FF	LUT	MSE
Available	N/A	N/A	1590	1540	508400	254200	N/A
FP32	1	834	2322	7244	14943048	646378	2.29e-9
FXD32.10	1	70	2322	4168	1401256	123158	1.61e-8
FXD24.8	1	78	1752	2120	1375918	82603	3.18e-6
FXD20.8	1	77	1448	2120	1158282	71575	3.39e-5
FXD18.8	1	28	1284	1352	279778	60477	7.93e-4
FXD16.7	1	28	1192	1352	252426	57581	3.69e-4

Deciding upon the precision requires careful analysis of the hardware resource utilisation and the error introduced for the typical signal with anomalies present. Table 3.4 shows a breakdown of the hardware utilisation by changing the neural network's precision. In this case DSP stands for the digital signal processing blocks, FF denotes Flip-Flop and LUT the look-up table resources. For single precision floating-point the required resources far exceed what is available on the FPGA, making a fully floating-point implementation intractable. For fixed-point precision 32, 24 and 20 the resource utilisation in terms of DSP, FF is significantly above what is available on the FPGA. This is due to the DSP architecture only supporting up to 18 bit multiplications in a single DSP. Hence, extra DSPs are needed to handle the extra precision and extra FF to support routing the results at a high frequency. For fixed-point 18 bit and 16 bit, the required resources are within the FPGAs limits, even allowing for additional functionally to be implemented if needed. 16 bit fixed-point was chosen for its lower resource utilisation, with minimal impact to the prediction accuracy as illustrated in the Mean Squared Test Set Error (MSE) column. Given the threshold values, presented in Section 3.6.1, required to detect the anomalies in RF signals, the different between $1e-9$ and $1e-4$ make little difference in the detection accuracy.

3.5.5 Interface

To support on-line updates of the neural network models, the implementation allows the neural network weights and biases, as well as the threshold value, to be updated during operation via a memory mapped register interface. This allows the FPGA to continuously perform inference, with continuous training at a lower speed on a microprocessor. Periodic updates of the weights and biases are made to the FPGA. The entire implementation is generated from a Python description using standard module generation techniques to produce a synthesisable C output. Google's TensorFlow package is used on the host machine for training. The implementation is configurable so that arbitrary sized FFT and neural networks can be implemented. Their configuration is set a compile time by the developer based on their requirements.

3.6 Neural Network Detection Evaluation

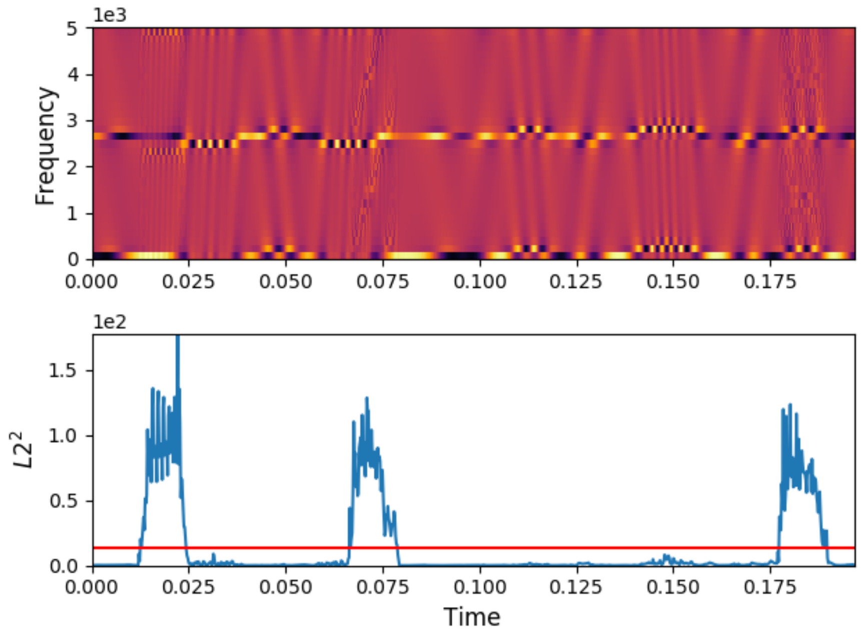
In this section, the performance of a 16-point complex FFT and a 32-input, 4-layer network (32,16,8,16,32) is studied. This network configuration was chosen since the network showed satisfactory accuracy in identifying anomalies and was small enough to fit on the FPGA in a fully pipelined manner. The radio platform was an Ettus X310 software defined radio, which supports DC-6 GHz operation with up to 160 MHz of baseband bandwidth; PCIe, dual 10 GigE, and dual 1 GigE interfaces; utilising a Xilinx Kintex-7 XC7K410TFFG900-2 FPGA. The design was synthesised from C to register transfer language (RTL) using the Xilinx Vivado HLS tool [58] and a bitstream generated using the Xilinx Vivado 2015.4 Design Suite. Verification of the hardware implementation was completed at three levels: C simulation, register transfer level simulation and testing on the Ettus X310 using the RFNoC software development package and interface.

3.6.1 Detector Accuracy

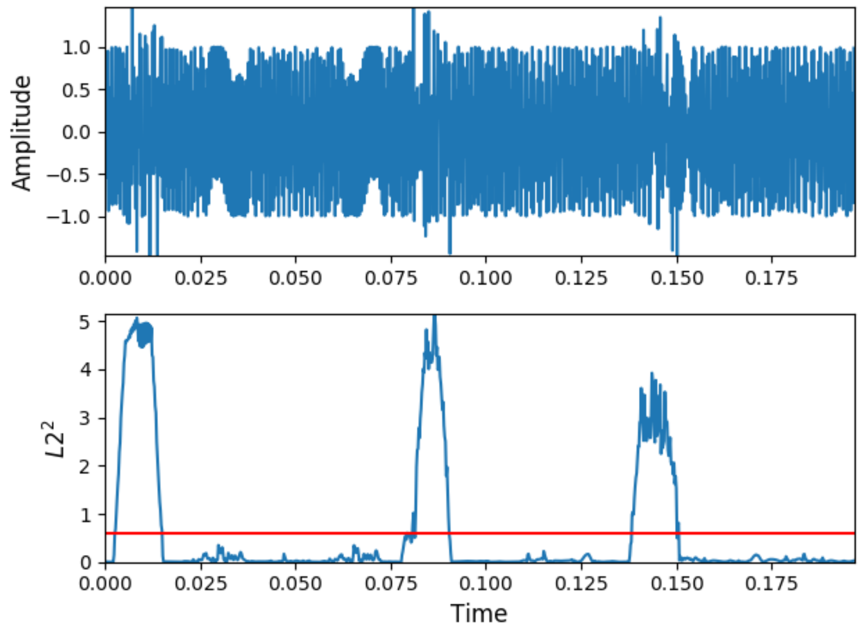
The detector was evaluated using a synthetic signal generated by modulating a sine wave with a randomly modulated tone and an FM signal was recorded using the Ettus X310 SRD. Three types of anomalies, similar to those in [48], were injected into each signal. The anomalies have a given amplitude, A, and exist over time window $t \in [t_s, t_e)$, where t_s and t_e are the start and end time of the anomalous period:

- Period of Gaussian Noise across the entire bandwidth: Modelled as $n(t) = Gaussian(-A, A)$.
- Pulsed Complex Sinusoid: $n(t) = A{*}exp(2\pi t F_n)$ where $F_n = Uniform(-F_s/2, F_s/2)/F_s$.
- Pulsed Chirp Event: $n(t) = A * exp(2\pi t F_n)$ where F_n ranges linearly from F_{c1} to F_{c2}. Both F_{c1} and F_{c2} are sampled from $Uniform(-F_s/2, F_s/2)/F_s$.

Figure 3.9a and Figure 3.9b respectively demonstrate successful detection of different anomalies in the frequency and time domain. Figure 3.9a shows the FFT windows (top panel) and the $(L_2)^2$ (bottom panel), whereas Figure 3.9b shows the signal in the time domain (top panel) and the $(L_2)^2$ (bottom panel) from the anomaly detector. In Figure 3.9a, the complex sinusoid was injected at time step 0.010 and its effect can clearly be show, with a correct identification of the anomaly in the $(L_2)^2$ panel. Similarly, at time step 0.075 and 0.150 the chirp event and Gaussian noise were injected respectively, both resulting in correct identifications of the anomalies.



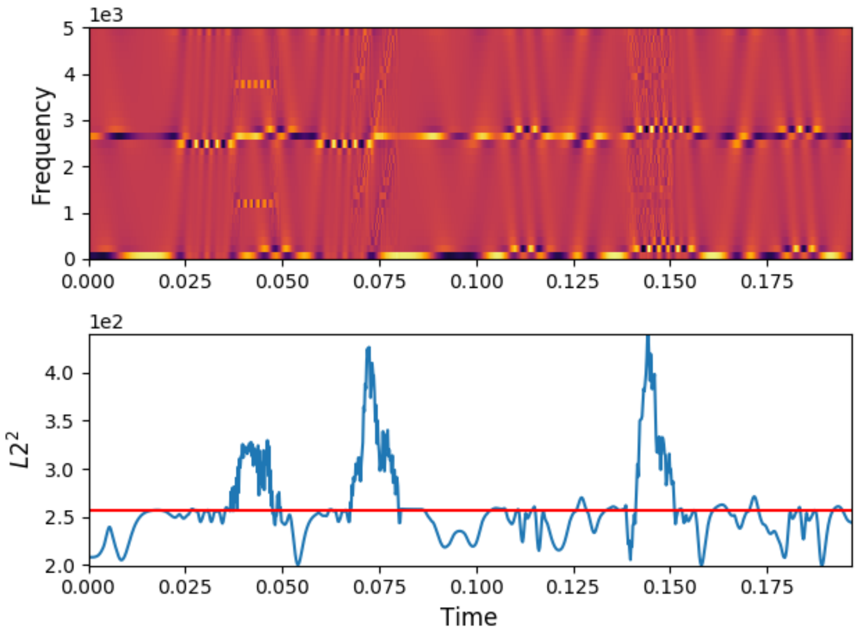
(A) Frequency domain anomaly detection example.



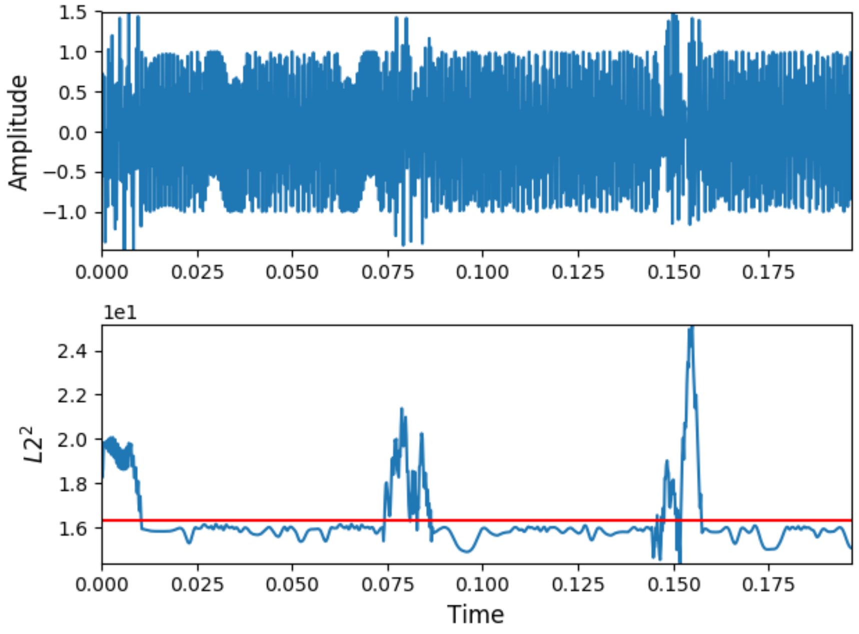
(B) Time domain anomaly detection example.

FIGURE 3.9: Anomaly detection example. The top panel displays the signal in the (a) frequency or (b) time domain. The 3 different types of anomaly can clearly be seen from left to right: pulsed sinusoid, chirp and Gaussian. The bottom panel shows the $(L_2)^2$ and anomaly threshold.

In Figure 3.9b, the same anomalies were injected in the same order at time steps 0.010, 0.080 and 0.150. For the synthetic signal, the neural network anomaly detector correctly identified the 3 types of noise with little difficulty. As illustrated in the figures, the threshold value used for all cases was 40% above the running average of the $(L_2)^2$.



(A) Frequency Domain Detection: Sample using the Ettus X310



(B) Time Domain Detection

FIGURE 3.10: FM Anomaly Detection: Sample using the Ettus X310

Figure 3.10a and Figure 3.10b shows the anomaly detector operation on real FM signals. The performance is similar to the synthetic case, resulting in correct identification of three anomalies. In a practical application, the choice of threshold should be guided by the value of the $(L_2)^2$, and an analysis over different signal-to-anomaly ratios (SAR), similar to Figure 3.11, should be performed.

Figure 3.11 shows the probability of correctly detecting an anomaly at different SARs. In this particular case, windows were identified as anomalous if the $(L_2)^2$ was 40% over its running average. The SAR was calculated using the power, $P = \frac{1}{N}\sum_{i=0}^{N}(x_i^2)$, and Equation 3.20.

$$SAR = 10\log_{10}(\frac{P_{Signal}}{P_{Anomaly}}) \tag{3.20}$$

As the power of the anomaly becomes proportionally smaller to the signal power, the probability of detection also decreases. As illustrated in Figure 3.11, detection in the frequency domain is more resilient to decreasing anomaly power, however its impact is marginal and may be overcome with different thresholding schemes.

3.6.2 Resource Utilisation & Performance

The results presented in Table 3.5 show the latency, initiation interval (II) and resource utilisation of the design. Importantly, all modules to perform inference (Windower, FFT, NN, $L2^2$ and Thres) have an II of 1, meaning it is fully pipelined, and a total latency of only 37 cycles. Since the implementation operates at 200 MHz, the throughput and latency are 200 (complex) MS/s and 185 ns respectively, as illustrated in Table 3.6.

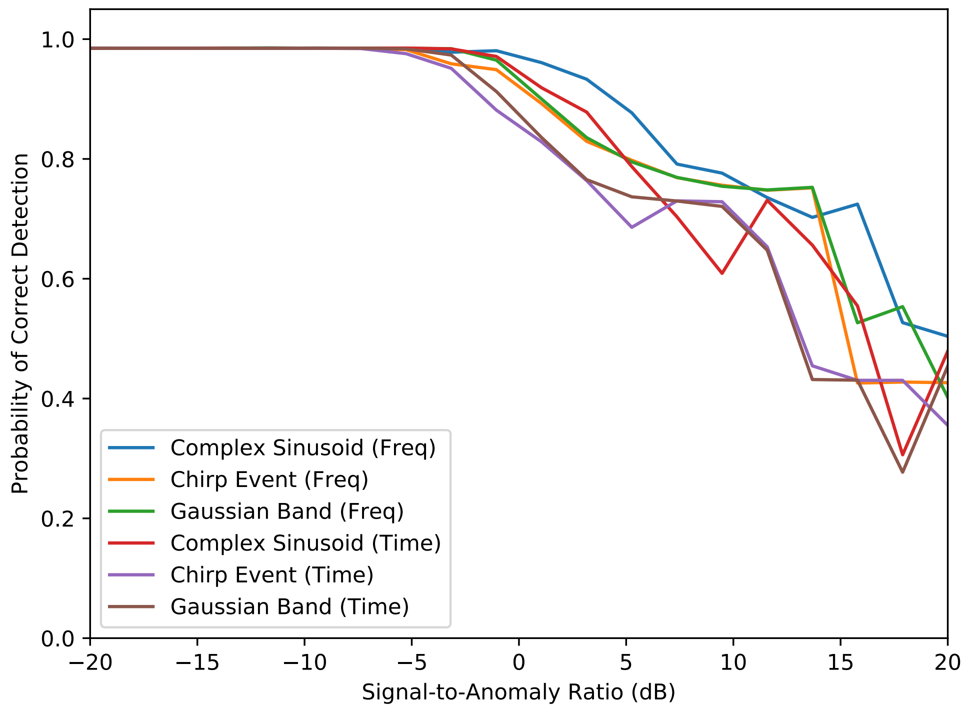


FIGURE 3.11: Probability of detection. Multiple signals with anomalies of varying amplitude were generated for the frequency and time domains. The figure displays the average probability of making a correct detection.

TABLE 3.5: Breakdown of autoencoder performance and resource utilisation

Module	II	Latency (cycles)	BRAM	DSP	FF	LUT
Windower	1	0	0	0	1511	996
FFT	1	8	0	40	4989	2577
NN	1	16	4	1280	199034	19273
$L2^2$	1	4	0	32	1482	873
Thres	1	0	0	0	3	21
Weight Update	258	257	0	0	22049	4609
224079 Inference (FFT+NN)	1	28	1196	1352	230132	29411
Inference (NN)	1	20	1196	1312	225143	26834
Total	N/A	N/A	1196	1352	252426	57581
Total Util.	N/A	N/A	75%	87%	49%	22%

Most BRAMs are contained within the top-level, 'Inference', module

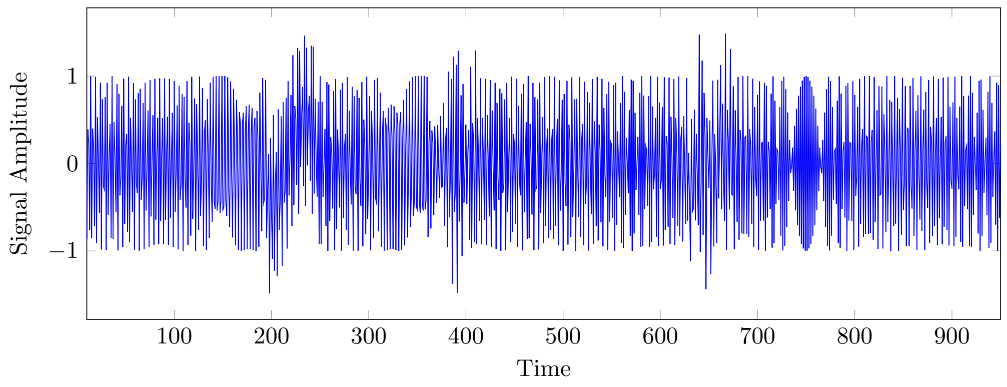
Since both the FFT and NN require multiply-accumulate operations, the DSP resources constrain the parallelism of the design. On-chip block random access memories (BRAMs) are the next resource constraint, since all weights and biases are stored on-chip. This could be addressed by using off-chip memory, at the cost of greatly increased latency; potentially becoming a bottleneck in the design.

3.7 Detector Comparison

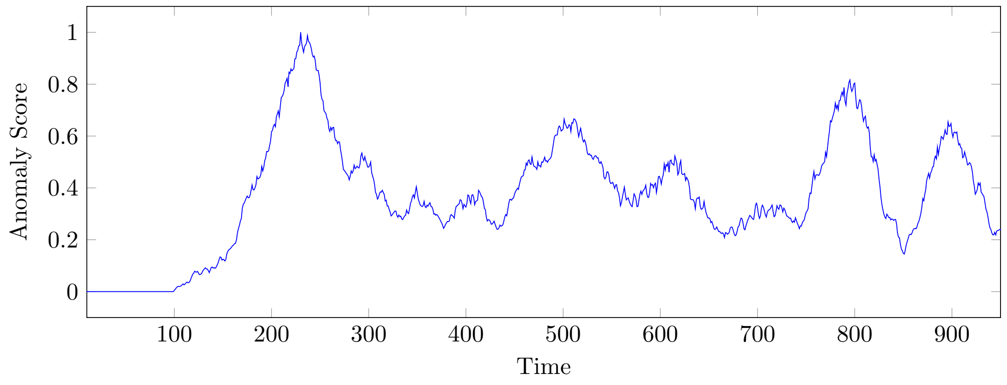
Figure 3.12 and Figure 3.13 illustrate the performance of the bitmap detector compared with the neural network detector for the synthetic and real FM data respectively. Unlike the 2D hand tracking example, the bitmap technique finds it significantly harder to correctly identify all three types. In Figure 3.12 the neural network detector performs well, correctly identifying the three anomalies. Compared with the bitmap detector, the first anomaly is detected, however the peak score is delayed. For the other two anomalies, their scores are relatively low and in both cases report a lower score than non-anomalous data.

TABLE 3.6: Raw anomaly detection performance.

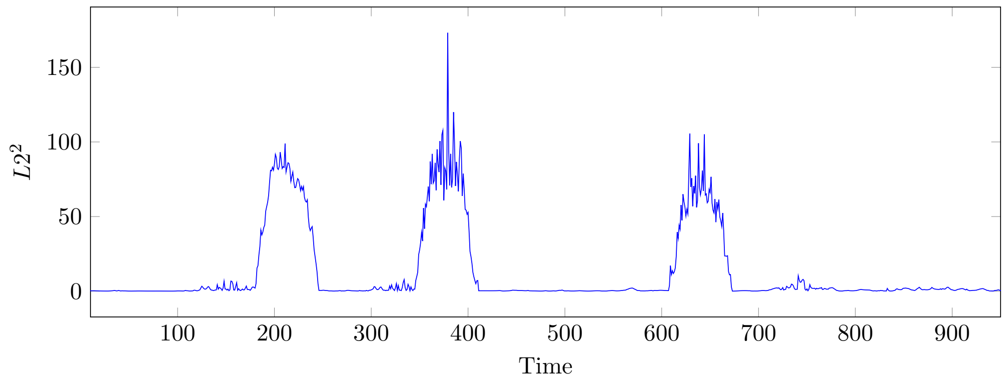
Operation	Throughput	Latency
Inference(FFT+NN)	5 ns	140 ns
Inference(NN)	5 ns	100 ns
Weight Update	1290 ns	1285 ns



(A) Synthetic Signal



(B) Bitmap Anomaly Detector Score

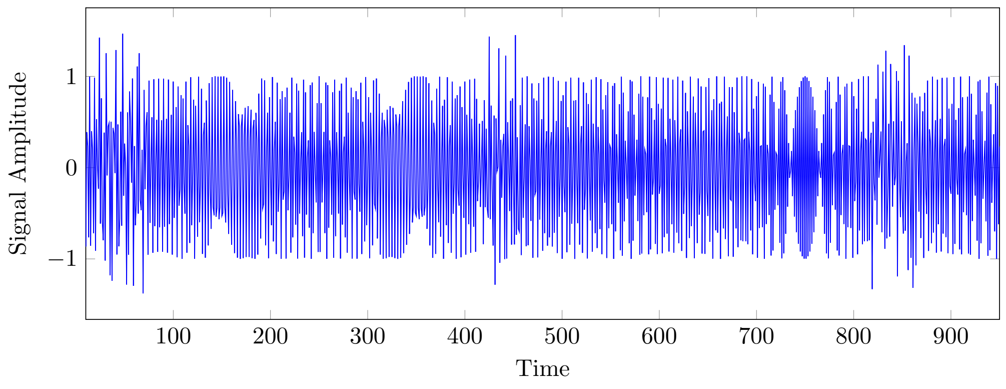


(C) Neural Network Detector Score

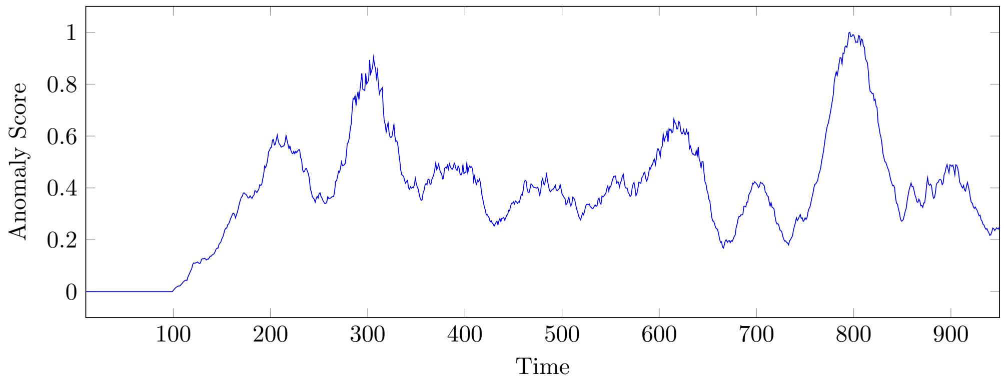
FIGURE 3.12: (A) This is the synthetically generated signal (B) The bitmap detector score, compared with the neural network (C), the bitmap technique significantly underperforms.

This performance trend is continued in Figure 3.13. Compared with the neural network detector, which correctly identifies the three anomalies, the bitmap detector only identifies the final anomaly correctly.

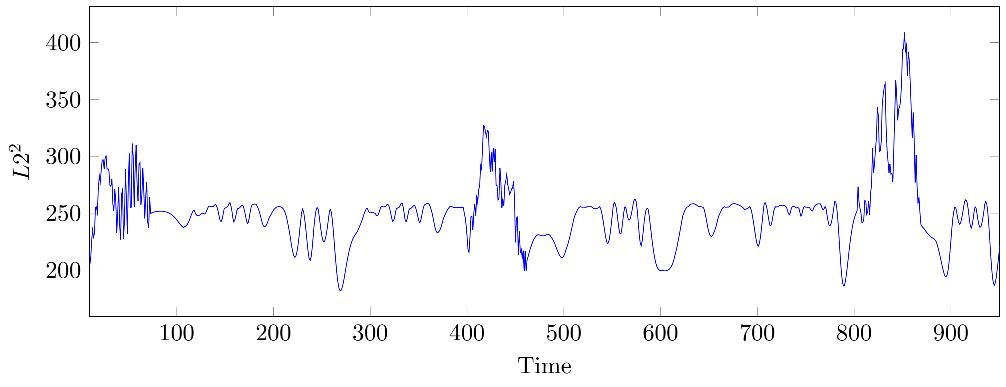
In regards to resource utilisation, the bitmap with 16 channels is compared against the standard neural network configuration in Table 3.7. Whilst both implementations target difference devices, the FPGA logic blocks are consistent across the two devices



(A) FM Signal



(B) Bitmap Anomaly Detector Score



(C) Neural Network Detector Score

Figure 3.13: (A) This is the recorded FM signal (B) The bitmap detector score, compared with the neural network (C), the bitmap technique significantly under-performs.

generations and are comparable in terms of FPGA resource utilistaion. As expected the simpler bitmap detector requires significantly less hardware resources than the neural network detector, using an order of magnitude less LUTs and 6× less DSPs. However, examining Figure 3.12 and more importantly Figure 3.13, illustrates that the bitmap detector significantly under performs, often incorrectly identifying non-anomalous data.

Table 3.7: Breakdown of autoencoder and bitmap performance and resource utilisation

Module	II	Latency (cycles)	BRAM	DSP	FF	LUT
Spectral Bitmap (16 Ch)	10	17	48	224	8656	3520
Neural Network (FFT+NN)	1	28	1196	1352	230132	29411

3.8 Summary

In this chapter, two different anomaly detection techniques were outlined, a neural network radio-frequency detector and a power spectra bitmap detector. The feasibility of single-chip, 200 MHz sample-at-a-time neural network anomaly detection, resulting in high throughput and ultra-low latency was demonstrated. This paves the way for the inclusion of real-time neural networks in sophisticated software defined radio systems, with potential applications in fault diagnosis, spectrum enforcement and collaborative spectrum sharing. Demonstrating the feasibility of addressing severely constrained real-time anomaly detection applications using FPGA technology.

One drawback of the neural network approach is the number of FPGA resources required to perform the computation. A bitmap detector was presented and compared against the neural network detector. While the bitmap detector required significantly fewer hardware resources, the detector performance was significantly lower than the neural network detector; only able to identify one out of the three anomalies correctly. Even though the neural network detector correctly identified all three anomalies, the detector's parallel performance was limited by the size of the network, which in turn is limited by the number of DSPs available on a given device. Given the flexibility of FPGA technology as opposed to other devices such as CPUs and GPUs, a low operating precision was chosen to facilitate a larger network, showcasing that the customizable logic allows for greater architecture exploration.

With this in mind, the next chapter presents a generalised matrix-multiplication framework, aimed at accelerating low precision neural network applications. The framework is implemented on the Intel HARPv2 [60], combining a 14 core Broadwell Xeon CPU and an Arria 10 GX1150 FPGA. It consists of a highly configurable hardware template with a streamlined software stack and runtime API, allowing for a wide range of different precisions, various core sizes and tuneable runtime configurable parameters.

Chapter 4

A Matrix Multiplication Framework

4.1 Introduction

To enable high performance machine learning on FPGAs, a framework which provides a consistent and reusable interface to the accelerator is necessary. Although machine learning algorithms differ in cost function and application, as discussed in Section 2.4 the majority of computation can be expressed as BLAS operations.

The previous chapter focused on machine learning applications and compared various precisions to their respective hardware utilisation. This chapter moves one level of abstraction lower and presents a framework designed for accelerating machine learning applications. The framework supports various precisions and operating modes as well as customisable modules designed for accelerating deep learning. The chapter contains a discussion on the design of low precision operating modes and optimisations for ensuring high compute efficiency. It begins with the relevant background on the Intel Heterogeneous Accelerator Platform Version 2 (HARPv2) and a brief look at reduced precision deep learning. Next, the framework, based on the GEMM compute algorithm discussed in Section 2.5.5, is presented along with an evaluation of its GEMM functionality. The chapter continues with a description of the deep learning specific functions added to the framework and concludes with an evaluation of three state-of-the-art reduced precision neural networks.

4.2 Contributions

The contributions addressed in this chapter are:

- A configurable heterogeneous GEMM implementation which supports arbitrary matrix sizes and offers a wide range of precision, blocking, fusing of operations, buffering schemes and load balancing.
- A systolic GEMM template that allows runtime customisation of memory interleaving, offering performance improvements of up to 2.7x on small matrices and 4x for certain neural networks. In addition, it incorporates a scheme for fusing operations so inline computation such as ReLU, Batch Norm and Clipping can be done in FPGA hardware, minimising CPU overhead.
- A dynamic dot product, enabling mixed precision training and binary inference which leverages the HARPv2 architecture, providing up to a 1.67x improvement over a 14 core CPU.
- An evaluation of performance using popular deep neural networks (AlexNet, VGGNet and ResNet) on the HARPv2 platform used for ILSVRC15[61], and a study on the efficiency of the hardware template and its impact on deep learning performance. The resulting binary implementation is, to our knowledge, the fastest and most flexible reported to date.

4.3 Intel Heterogeneous Accelerator Platform Version 2

The Intel HARPv2 is a heterogeneous CPU and FPGA system with an Intel Xeon E5-2600 v4 and an Arria 10 FPGA GX1155 [60]. The Xeon is a server-based CPU designed for parallel compute and energy efficiency using the x86 instruction set. Typically, the Xeon line of CPUs provide more parallel processing cores than their desktop equivalent, the Core i series. The E5-2600 has 14 cores all connected by two levels of shared memory, the cache and system memory. The cache is a small, but fast local memory utilised by the CPU to store its most accessed and used information. By comparison, system memory is much larger; at the cost of increased latency in accessing stored information.

Differing from the aforementioned heterogeneous CPU and FPGA systems in Section 2.2, the FPGA is connected to the CPU via three distinct links: the Quick Path Interconnect (QPI), and two PCI-e links. To support these links the FPGA is partitioned into two parts, a runtime static region containing the Blue Bitstream and a runtime reprogrammable region. As illustrated in Figure 4.1, the programmable section of the FPGA communicates to the Blue Bitstream$^{\text{TM}}$ via the Cache Coherent Interface (CCI). Memory requests through the CCI are taken by the Blue Bitstream and routed to the Xeon

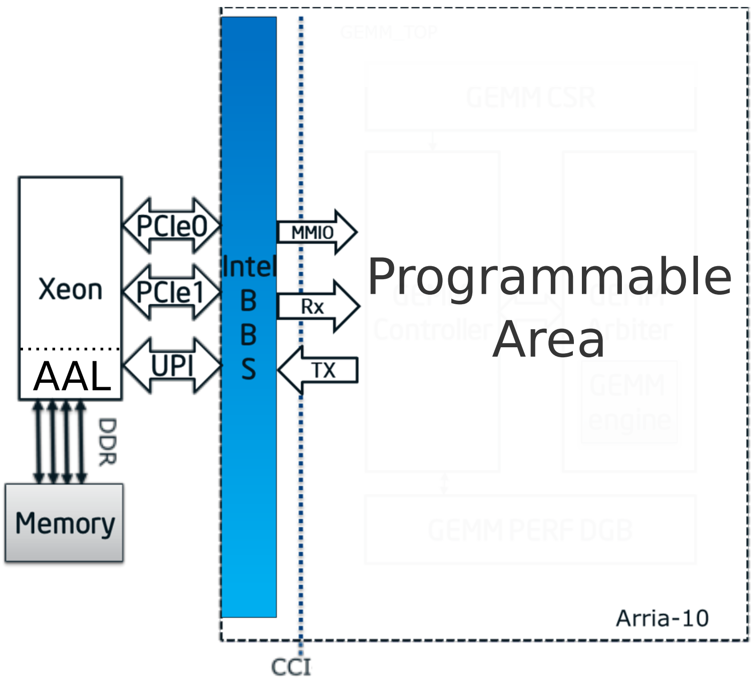


FIGURE 4.1: Intel HARPv2: The blue bitstream communicates directly with the Xeon and makes memory requests to system memory. The CCI interface abstracts away the complexity of handling the three links and provides a DMA like interface to the user.

via one of the three links, QPI (VL0), PCI-E 0 (VH0) or PCI-E 1 (VH1). Memory requests are made to data located in either the cache or system memory that is addressable to both the Xeon and FPGA. This capability allows for two unique opportunities: (1) since both the Xeon and FPGA are on the same package, the FPGA architecture can be targeted to accelerate only the important parts of the algorithm, taking advantage of any domain specific optimisations that are unavailable to the CPU. (2) Both the CPU and FPGA can work in the same address space, enabling fine-grained offloading and heterogeneous load balancing. Load balancing can be performed on a per-op basis as opposed to dataflow type approaches in which distinct parts of the algorithm are handled by each core. These key advantages allow the Xeon and FPGA to work harmoniously and take advantage of each device's strengths.

4.3.1 Intel Accelerator Abstraction Layer

Shown in Figure 4.1, the Accelerator Abstraction Layer (AAL) is a software stack that runs on the Xeon and provides the necessary runtime services and API to access the FPGA device. At a very high level, AAL services can be briefly classified into two categories:

- **AAL user-mode runtime:** These are interfaces that abstract the FPGA hardware via a service oriented model. Various services in the AAL user-mode runtime can be aggregated to build application specific services.
- **AAL kernel-mode driver:** These include interfaces for allocation of Direct Memory Access (DMA) buffers with shared addressing between the hardware and the user's application. It provides interfaces to access Memory Mapped IO (MMIO) registers in the hardware.

4.3.2 Intel Blue Bitstream

Intel Blue Bitstream (BBS), illustrated in Figure 4.1, is the infrastructure shell component in the FPGA. It abstracts the QPI and PCIe links to provide a simple, load-store-like, interface to the user's accelerator, the CCI. The Intel BBS also handles partial reconfiguration (PR), a method for updating the reconfigurable region without powering down the device, and provides AAL kernel visible MMIO registers for device enumeration and initialisation.

4.3.3 Existing accelerators on Xeon+FPGA

As the Xeon+FPGA platform continues to gain popularity, prior work has studied acceleration on this platform, such as [62, 63]. [62] studied CNNs with math optimisation. [63] studied irregular pointer chasing applications. Heterogeneous CPU-Accelerator platforms are quickly becoming pervasive throughout computational systems and clusters. With the fast adoption of machine learning and deep learning in business, the computational requirements of cloud and local distributed systems are increasing at an exponential rate.

Several studies [64–66] have focused on key workloads to better understand the requirements of these algorithms and their performance on CPU+FPGA systems. [64] provides a quantitative analysis of a QPI based CPU+FGPA system compared to a PCI-E based CPU+FPGA system. Key differences between the two platforms, such as different memory models and peak bandwidth, were highlighted and a decision tree based flowchart was provided as a guide to assist developers when choosing a platform. The main conclusion was that QPI based systems are desirable for applications where the CPU and FPGA need to operate on the same small set of data, whereas PCI-E based systems tend to perform better for offload type applications.

4.4 Reduced Precision Deep Learning

With significant research [11–17] indicating that 8 bit or lower precision is sufficient for inference, dedicated hardware such as the Google TPU [9] and the NVIDIA V100 GPU [67], which are optimised for lower precisions, have been reported. The benefit of moving to a reduced precision format for neural network computation lies in the efficiency of the multiply and accumulate operations. By moving from single precision floating-point to a 32 bit fixed-point, normalisation is removed and scaling is simplified, resulting in smaller hardware. Hence, by lowering the number of bits, n, in the representation, the area and complexity requirements of the multiplication reduce by factors of n^2 as per Equation 2.21 for x and y of the same bit width. In summary, this results in:

- A smaller memory footprint compared to the traditional single precision floating-point.
- The ability to replace the conventional multiply-accumulate with more area efficient implementations.

4.4.1 Binarised Neural Networks

Binarised Neural Networks (BNNs) are gaining significant traction in the neural network and FPGA communities [11–17]: The number of multiply-accumulates (MACs) needed by newer topologies is growing significantly and it often takes days or weeks to train these networks. As the number of layers in these networks increases, the size of the model can quickly surpass the available system memory when stored as single precision floating-point numbers. BNNs address these issues by changing the representation from single precision floating-point to a single bit for either the weights or both the activation and weights; resulting in a 32x reduction in storage. To support this new representation, the dot product $\boldsymbol{w} \odot \boldsymbol{a}$ changes to either a conditional negation and accumulation or an XNOR and signed bitcount in the case of both binarised weights and activations.

$$\begin{array}{|c|c|c|}\hline -1 & 1 & 1 \\ \hline 1 & 1 & -1 \\ \hline -1 & -1 & 1 \\ \hline \end{array} \times \begin{array}{|c|}\hline 0.4 \\ \hline 1.9 \\ \hline -4.2 \\ \hline \end{array} = \begin{array}{|c|}\hline -2.7 \\ \hline 6.5 \\ \hline -6.5 \\ \hline \end{array}$$

(a) Binary Weights and Real Activations

$$\begin{array}{|c|c|c|}\hline -1 & 1 & 1 \\ \hline 1 & 1 & -1 \\ \hline -1 & -1 & 1 \\ \hline \end{array} \times \begin{array}{|c|}\hline 1 \\ \hline 1 \\ \hline -1 \\ \hline \end{array} = \begin{array}{|c|}\hline -1 \\ \hline 3 \\ \hline -3 \\ \hline \end{array}$$

(b) Binary Weights and Activations

FIGURE 4.2: Two types of Binary Neural Network implementations: (a) demonstrates the real activations and binary weights case. (b) illustrates where both activations and weights have been binarised.

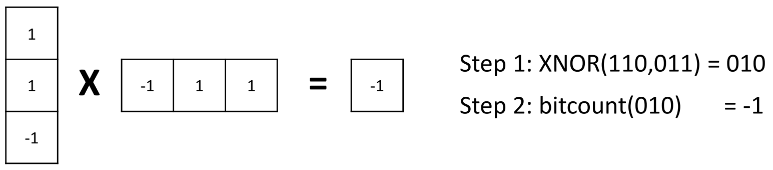


FIGURE 4.3: Example of a Binarised Dot Product

Figure 4.2 illustrates two different types of BNN network operations, the same process is applied to both the FC and CONV layers of the network. The first type of binarisation are binarised weights and full precision activations. This has the benefit of reducing the amount of storage for the weight matrix, as well as removing the need for a full precision multiplication. In the first case, the sign of the binarised weights is applied to the activations and an adder chain is used to accumulate into a single result. Binarised weights and full precision activations generally reduce the storage requirements of the weight matrix by 32x and replace multiplication with a conditional negation.

In the second case, both the weights and the activations are binarised. Traditionally, Equation 2.31 would use multiplications and additions to perform the dot product. However, as illustrated in Figure 4.3, since both activations and weights have been binarised, the operation can be replaced with an XNOR and a signed bitcount, similar to a population count, given that the representation follows: $1 = 1$ and $-1 = 0$ in binary. The bitcount operation performs a running sum where each 1 contributes a $+1$ and each 0 contributes a -1.

While BNNs allow a substantial number of matrix multiply operations to be converted to binary operations (XNOR and bitcount), there are still other types of operations required by the algorithm. A state-of-the-art BNN [15] typically continues to use full precision (FP32) for the first and last layers of the network during forward pass, in order to achieve acceptable accuracy. Prior FPGA accelerators proposed modifying the algorithm (e.g., [16, 17]) to use fixed precision data types that are more FPGA-friendly (i.e., do not require FP32 operations). However, their experiments targeted only smaller datasets (MNIST, CIFAR) instead of larger scale (ImageNet), where accuracy is often more sensitive to such data type changes.

Moreover, during the backward pass, the gradients are calculated in floating-point precision and require a matrix multiplication between the floating-point and binarised values (i.e., full precision gradients and activations against the binarised weights). Implementing the variety of computations needed for both training and inference has the potential to over-complicate the FPGA design. This may not be the optimal design choice and offloading computation to a more appropriate device may result in higher performance.

4.4.2 Previous Reduced Precision Deep Learning Accelerators

Interest in low precision CNN's has dramatically increased in recent years. Research has shown that similar accuracy to single precision floating-point can be achieved [14, 15, 68–72]. Due to the high computational requirements of CNN's, reduced precision implementations offer opportunities to reduce hardware costs and training times. Since FPGAs can implement arbitrary precision datapaths, they have advantages over the fixed datapaths of GPU's and CPU's. Moreover, the highest performance implementations on all platforms utilise reduced precision for a more efficient implementation.

Previous work [73–87] has mainly focused on general accelerators, implemented either as a sequence of instructions on fixed hardware, or accelerator platforms designed for linear algebra intensive computation.

Systolic array based architectures implement a grid of local-connected processing units to perform a matrix-to-matrix multiplication. Most notability, Jouppi et. al. [73] describe the Tensor Processing Unit (TPU), an Application Specific Integrated Circuit (ASIC) that utilises a systolic array to compute operations necessary with 8 or 16 bit weights and activations. It boasts a very high throughput on a variety of applications with a latency on the scale of ms, claiming a peak throughput of 92 TOps/sec. Venkatesh et. al. [82] use a method described in [88] to implement a VGG style network with ternary weights and half-precision floating point activations. They create an ASIC accelerator for training networks in addition to inference with a systolic array like structure. Due to the use of floating point activations, they achieve high accuracy on CIFAR10 of around 91%.

Differing from the systolic array approach, vector processors contain several independent processing lanes, with the capability of each determined by the lanes architecture. Chen et. al. [74] created a custom ASIC utilising 16 bit fixed point achieving a peak throughput of up to 42 GMAC/s. Their architecture contains an array of independent processing elements, each receiving operation instructions. In the programmable logic domain, Wang et. al. [75], Qiu et. al. [76] and Meloni et. al. [78] presented neural network accelerators for the Zynq CPU+FPGA. Each implemented a vector processor and utilised low precision to improve computational performance for object detection and image recognition. Wang et. al. [75], Qiu et. al. [76] and Meloni et. al. [78] achieved 2 TOps, 187 GOps and 169GOps respectively. Finally, Zhang et. al. [79] implemented an accelerator using the frequency domain representation of the convolution. Their datapath converts the activations into the frequency domain and uses a pointwise multiplication to perform to convolution; this is followed by an inverse FFT. The computation is performed in floating point and implemented on the Intel QuickAssist platform containing a CPU and Stratix V FPGA, achieving 123.5 GFLOPs.

In the past, FPGA devices did not have sufficient capacity to implement entire neural networks on-chip, and single-chip deep learning applications were intractable. In recent years, however, high-performance FPGA implementations of neural networks for inference have been reported. An illustrative example by Zhang et. al in 2015, achieved 62 giga floating-point operations per second (GFLOPS) in single precision floating-point [89], using a roofline model to balance computational resources and memory bandwidth. Very low precision implementations have also been reported. For example, binarised (1-bit) implementations of neural networks can achieve 12.3 million image classifications per second with $0.31\mu s$ latency on the MNIST dataset with 95.8% accuracy [17]. By reducing precision, it is possible to keep all weights on-chip, achieving higher performance with lower energy consumption. In a manner similar to the present design, both implementations used high level synthesis from a C description.

Beyond FPGAs, there have been many accelerators proposed for neural networks, and other machine learning algorithms in general. Several studies have proposed ASIC accelerators for neural networks (e.g., [90]), as well as other machine learning algorithms (e.g., [91]). GPUs have also been used to accelerate neural networks. As the Xeon+FPGA platform continues to gain popularity, there has been prior work that studied acceleration on this platform, such as [62, 63]. [62] studied CNN with math optimisation (e.g., FFT transformation).

4.5 Overview

To accelerate machine learning algorithms on the Intel HARPv2 platform, a matrix multiplication framework, illustrated in Figure 4.4, was designed and built to handle multiple precisions and a wide range of applications. Various HW/SW co-design and heterogeneous load balancing techniques are applied to achieve synergistic collaboration between Xeon CPUs and the FPGA. The framework supports arbitrary matrix sizes and offers a wide range of precision, blocking, fusing of operations, buffering schemes and load balancing. It contains a systolic GEMM template that enables runtime customisation of memory interleaving and a scheme for fusing operations; allowing inline computation to be performed on FPGA hardware while minimising CPU overhead.

The hardware template contains a dynamic dot product, enabling mixed precision datapaths in the same hardware. As illustrated in Section 3.6.2, the number of DSPs quickly became the bottleneck to further parallelising the design, since the network was fully

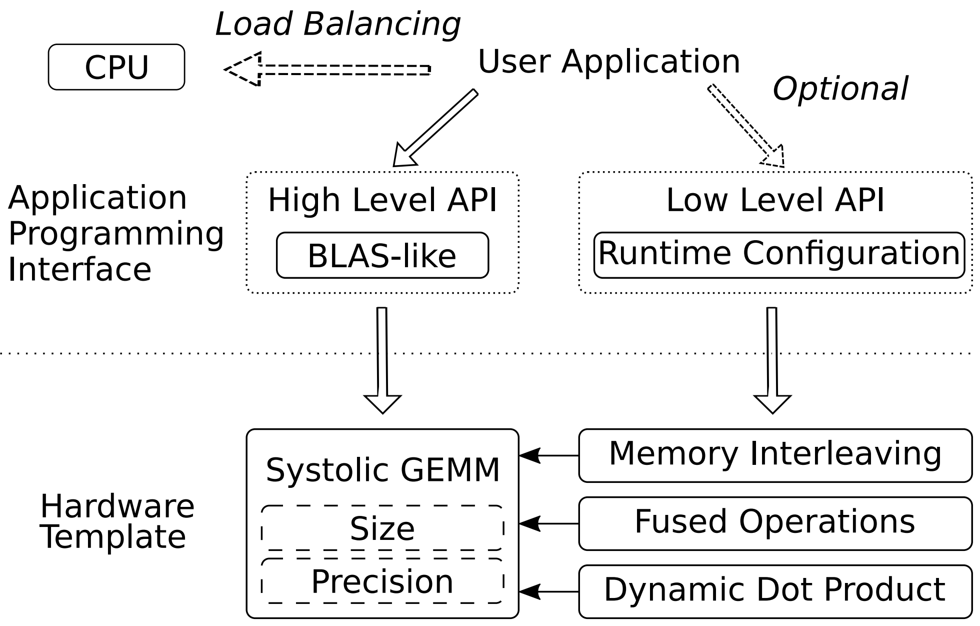


FIGURE 4.4: The framework consists of an API and hardware template. The high level API provides a function call similar to those used in BLAS libraries. The low level API is optional and allows the developer to configure certain aspects of the hardware template at runtime.

pipelined. To allow the anomaly detector to support larger compute graphs, a systolic based architecture, such as the one presented in this chapter, allows computation of arbitrarily sized layers to be staged. This removes the network size restriction at the cost of latency and throughput. The main difference between the two approaches is that the systolic array is reused multiple times to compute larger layer sizes, whereas the fully unrolled approach aims to map each multiplication and addition onto the FPGA fabric. The framework is designed for the HARPv2 architecture, however other FPGAs are supported via the CCI interface.

4.6 API

The hardware template presented in Section 4.7 is implemented on the Intel HARPv2 platform with an accompanying software stack and API. As illustrated in Figure 4.4, the API contains a high-level function interface for easy integration and a low-level template interface for fine-grained control. To maintain consistency with other GEMM implementations, the high level API is modelled on other linear algebra libraries. Given Equation 2.55 and previous BLAS libraries, the simplified GEMM signature for the single precision floating-point (FP32) version is:

```
void gemm(trans a, trans b, int m, int n, int k, float alpha, float* a, int
    lda, float* b, int ldb, float beta, float* c, int ldc);
```

This signature is provided in a set of libraries that are easily compiled into the developer's code base. For most projects, this should provide sufficient performance as optimisations are performed within the function without developer interaction.

Table 4.1 presents a list of the current parameters tuneable in the GEMM implementation. Both the precision and accumulator width are configurable at compile time when the systolic GEMM bitstream is generated. If multiple precisions are required for a workload, the API provides a single function for partial reconfiguration, allowing for fast precision switching, on the order of 300 ms. Post processing fused operations such as value scaling, clipping, rounding and a few deep learning specific operations such as Rectified Linear Unit (ReLU) and Batch Normalisation can be performed; while the results are transferred back to the system memory. These post processing operations are enabled at compile time to be added into the design and can be configured to be bypassed at runtime.

For precisions other than FP32, the developer can set the desired accumulator width at compile time, however this affects memory and logic resource utilisation. Similarly, the developer has access to the systolic array interleaving factors. These allow for fine-grained adjustments to trade off bandwidth with compute efficiency. Section 4.7.3 covers this in more detail. The maximum interleaving level is set at compile time and is bound by the number of memory resources available on the device. The exact level, however,(up to the set maximum) can be controlled at runtime via the lower level APIs.

4.6.1 Runtime Support

The API exposes various configurable parameters to the user-level software. Applications that use the framework, leverage the Intel HARPv2 user mode runtime and kernel driver to set these parameters. The integration of runtime software with the hardware accelerator template is shown in Figure 4.5. The hardware template and API chooses

Table 4.1: Tuneable Options

Parameters	Type	Options
Systolic Array Size (Section 4.7)	C	*Logic & Memory Limited
Precision (Section 4.7.1)	C	FP32, INT16, INT8, INT4, Ternary, Binary
Accumulator Width (Section 4.7.1)	C	*Logic & Memory Limited
Interleaving (Section 4.7.3)	C&R	*Memory Limited
Fused Ops (Section 4.9.1)	C&R	Scaling, Batch Norm, Clip, Rounding, ReLU

*Limited by the size of the systolic array and available hardware resources. Features are controllable at compile (C) time, runtime(R) or both (C&R)

between the different memory links. The default setting performs bandwidth balancing between all three.

```
template<typename T1, typename T2>
void fpga_gemm<T1,T2>::fpga_gemm(
trans a, trans b,
int m, int n, int k, float alpha, T1* a, int lda,
T2* b, int ldb, float beta, T1* c, int ldc
int i_a_lead_interleave, int i_b_lead_interleave, int i_feeder_interleave,
GEMM_MODE i_mode);
```

The low-level functions are templated to support different precisions and modes in Table 4.1. The number of elements packed into cacheline also changes depending upon the precision. The low level API is shown above. "a_rows" and "b_cols" refers to the number of rows and columns in A and B matrices respectively. The "common" parameter refers to the common dimension in both the matrices. "i_alpha" and "i_beta" refer to the scaling parameters and "i_mode" refers to the mode in which the hardware accelerator template is set. "i_a_lead_interleave", "i_b_lead_interleave" and "i_feeder_interleave" are the interleaving parameters. These parameters can be used by the runtime to control the memory interleaving and improve the compute efficiency. Internally, the application API uses the AAL user mode runtime to access and initialise the FPGA device.

Switching precision during runtime is supported by the dynamic configuration API. The API shown below, requires precompiled bitstreams and the AAL service, which internally uses partial reconfiguration to switch from one mode to the other.

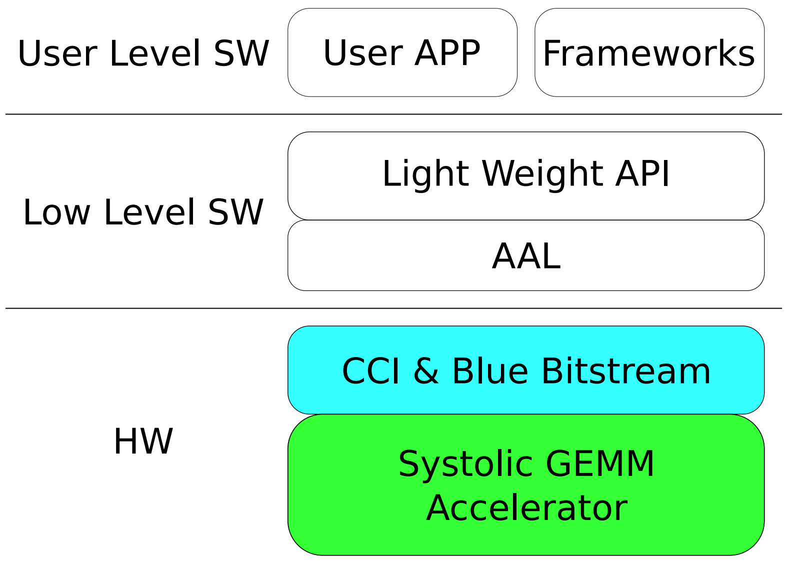


FIGURE 4.5: Software and Hardware Stack: The CCI, blue bitstream and AAL are provided by the HARPv2 platform.

```
int config_afu_sgemm(const char *pathname) {
gemmAAL<int , int> hardware_template;
hardware_template.setHW(true);
return hardware_template.configSGEMM(pathname); }
```

4.6.2 Heterogeneous Load Balancing

The hardware template also supports heterogeneous load balancing. At runtime the workload is partitioned across both the FPGA and CPU. In the case of a GEMM, the A and B matrices are divided into sub blocks and the computation is balanced across the two compute engines. This is useful for particularly large workloads in which the majority of the work is taken by the GEMM function.

4.7 Hardware Template

The hardware template illustrated in Figure 4.4 contains the systolic array and several modules which handle memory interleaving, a fused operation scheme and dynamic dot product. As illustrated in Figure 4.6, the hardware template is a systolic array of processing elements (PEs), each containing a dot product module and two memory buffers; named the cache buffer and drain buffer. The systolic array operates by iteratively processing chunks of the input matrices stored in the feeders. There are two orthogonal feeders that connect to their respective edges of the array. The design is fully pipelined, with each cycle's data fed into the array via the feeders and propagated along the appropriate rows and columns. The feeders are, by default, double buffered to ensure multiple read requests are in-flight; in order to saturate system bandwidth and minimise compute stalls due to insufficient memory. The data management unit (DMU) is responsible for requesting the input data, filling the feeders, draining out completed sections of the compute and generating write requests to the system memory. Within the systolic array, input vectors are interleaved into each PE to take advantage of data reuse and help meet the bandwidth requirements of the system. Since the input is interleaved, a small cache within each PE is necessary to store the partial results for accumulation used later in the computation.

4.7.1 Processing Element

The PE, illustrated in Figure 4.7, contains a dot product module with two memories: a partial results 'cache' and completed results 'drain'. Input vectors are passed into each

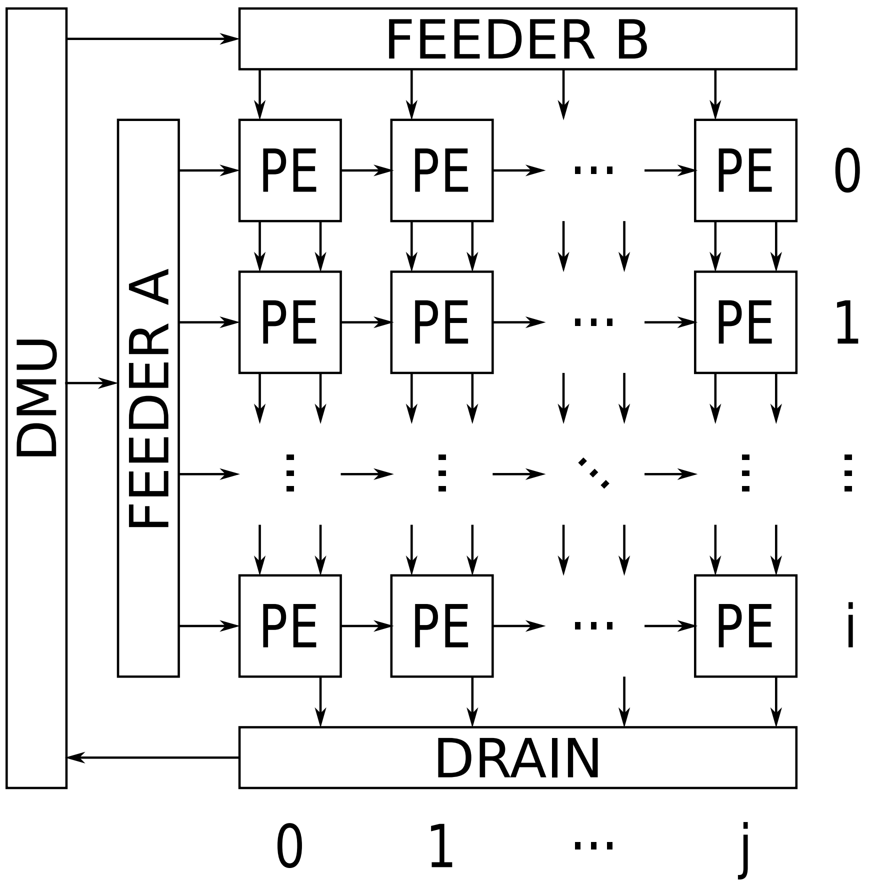


FIGURE 4.6: Systolic Array: The array size is configurable with one limitation, the drain bus width must be ≤ 64 bytes. For FP32 this limits the number of columns to $j = 16$.

PE every cycle where a dot product is performed and partial results are accumulated. In cases where the dot product is larger than the input vector length, the partial result is stored in the cache to be used later in the computation. If the dot product length is smaller than the input vector length, or more commonly, the final partial input vectors have been passed to the PE; the completed result is stored in the drain and is ready to be taken out of the array.

To ensure high throughput, reading out the array, i.e. 'draining', can be performed while partial results of the next chunk are produced and stored in the cache, as both memories operate independently. One exception to this, is when a set of complete results would be written into a non-empty drain. In this case, the computation is stalled until the drain is empty. The array control signals for the computation stage are passed across rows, whereas the control signals for the drain are passed across columns. Each PE is responsible for passing data in a linear fashion along its row and column. Additionally, each PE is fully pipelined so that the result of a single dot product is produced every cycle.

Depending on the desired bit width, the dot product is either performed in DSPs, constructed using logic resource, or a combination of both. The dual DSP and logic element dot product simply divides the multiplications and additions by first fully utilising the allocated DSP resources, then implementing the rest in logic. For example, in the 16 bit fixed-point case, the DSP resources are sufficient to implement all multiplications and additions. However in the case of 8 bit fixed-point, half of the operations are implemented using logic; the first 16 multiplications and additions performed in the DSPs and the second 16 performed using logic. In the case of 4 bit fixed-point this is more pronounced, since the ratio is 1:3, rather than the 1:1 of the 8 bit case. The PE currently supports single precision floating-point (FP32), 16, 8, 4 and 2 bit fixed-point (INT16, INT8, INT4, INT2) and more exotic data types for deep learning workloads: INT16 × Ternary, INT8 × Ternary, Binary × Binary (BIN × BIN).

For the fixed-point data types, the bit width of each stage of the adder tree in the dot product is increased by one. Apart from FP32, the accumulator bit width is configurable at compile time for all data types. After accumulation the results are truncated and rounded before they are stored into the 'cache' or 'drain'.

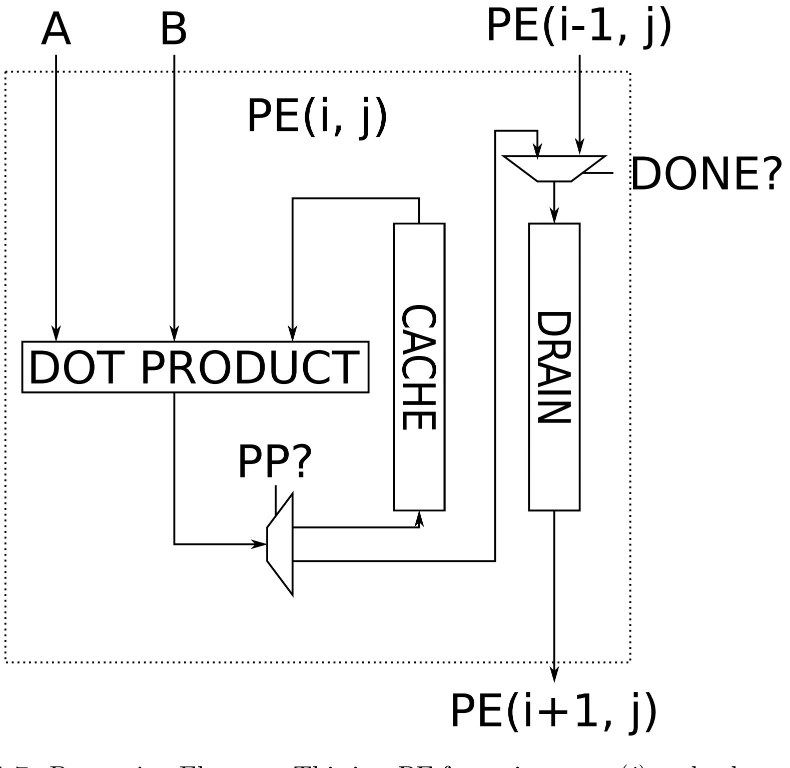


Figure 4.7: Processing Element: This is a PE for a given row (i) and column (j). The dot product data type is configurable at compile time and is the only data dependent module in the entire architecture.

4.7.2 Feeders & Drain

The feeders are memory modules that manage the flow of data into the array. By default, each feeder is double buffered allowing one buffer to be operated while the other is being filled. The number of buffers is configurable and more than one may be desirable when: (1) one matrix is significantly smaller than the other or, (2) when the CPU is operating under certain load conditions that reduce the overall bandwidth given to the FPGA.

By adding additional buffers, periods of poor performance due to low bandwidth can be reduced. The feeders operate in one of three stages, idle, loaded or full.

- Idle: All buffers are either empty or in the process of filling, no computation can be performed during this time.
- Loaded: One buffer is completely full and is available for computation, during this time the other buffers are empty and can be filled.
- Full: All buffers are full and at this time memory requests are stalled.

When the feeders are either loaded or full there is at least one complete buffer ready to be fed into the grid.

The drain is a large interconnect though which the results flows into the columns of the systolic array.

When signalled to drain, the memories at the bottom of the array start to empty. Each column acts as a large first-in, first-out memory (FIFO) that produces a result every cycle. As memory locations become available in the bottom row of the PEs, the PEs above start to empty and begin filling the PEs below them. By combining all columns into a single data bus, the grid produces one complete cache-line every cycle. Stochastic rounding and round to nearest are supported during draining by the framework and are configurable at compile time.

4.7.3 Blocking and Interleaving

During GEMM each element in matrix A is used n times and each element in matrix B is used m times. By aiming to store both A and B on-chip and reusing each element, bandwidth requirements are minimised. When dealing with larger matrices, the number of on chip memories quickly become the limiting factor. To handle larger matrix sizes both A and B are partitioned into chunks and transferred in batches. This is usually referred to as blocking and is a standard practice in GEMM implementations to achieve high performance.

Interleaving, on the other hand, is an architecture specific optimisation designed to enable data reuse on a fine-grained level. It takes advantage of the data reuse in a GEMM and operates by feeding the same vectors into the PEs in a specific order. Both the leading dimension of matrices A and B, m and n respectively, have independent interleaving factors that are controlled at both compile and run time. Figure 4.8 shows a simplified example, a 1x1 systolic array, of how interleaving operates within the hardware template. Each row in Feeder A and column in Feeder B are partitioned into two separate blocks, a_x and b_x respectively, the result of these partitions are accumulated to create the final 3x2 matrix. The interleaving factors for feeder A and B are 3 and 2 respectively.

At $t = 0$, a_0 and b_0 are passed into the PE and the partial result is stored in the first location in the cache. At $t = 1$, the pointer to memory location a in incremented, b_0 is reused and a_1 is passed into the PE with the partial result stored in the second location in the cache. This continues on to $t = 3$ where the a memory pointer is reset back to zero and the b memory pointer is updated, now a_0 is reused and b_1 is passed into the PE, the

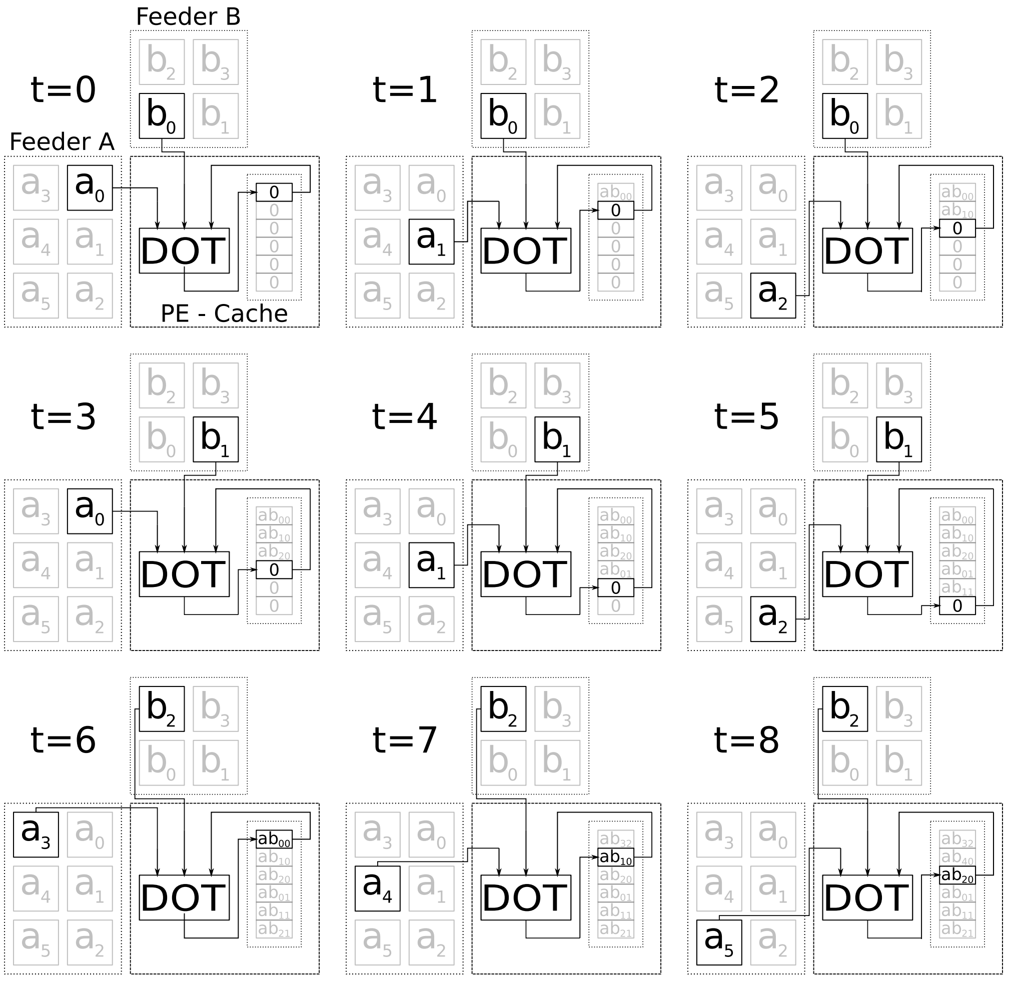


FIGURE 4.8: Interleaving Example: This shows a simplified example of how the PE operates and the concept of interleaving.

partial result is stored in the fourth location in the cache. At $t = 6$ the first column and row of A and B respectively, have been processed and moves on to the second row and column. Instead of accumulating zeros, the previous value from $t = 0$, a_0b_0, is added to the result of a_3 and b_2 and is stored back into the first location. This process continues in a similar fashion for $t = 7, ..., 11$ until all rows and columns have been processed.

A combination of blocking and interleaving are used together to improve performance. The blocking size is determined by the interleaving factor as well as the number of rows and columns in the systolic array. With a fixed interleaving size, the systolic array performed at near peak theoretical performance for large matrices, however there was a significant decrease in performance for smaller matrix sizes. This is due to the fact that both matrix A and B needed to be padded with zeros until they were a multiple of the block size. It was observed that by adding configurable memory interleaving support at runtime, the performance significantly improved as these restrictions were lifted. The performance improvements are discussed later in Section 4.8.2.

In most cases the optimal interleaving size can be directly calculated using:

$$I = \min_{x_L,\dots,x_H} modulo(DIM, x) \tag{4.1}$$

where x is the range of different interleaving values supported by the hardware template and DIM is the leading dimension of either the A or B input matrices. It follows that the size of the blocks leading dimension for A and B can be calculated using $S_A = I_A * HW_{ROWS}$ and $S_B = I_B * HW_{COLS}$ respectively.

Now, given the dot product size (S_D) and the length of the a block's common dimension (S_C), the number of cycles per block can be calculated using the A and B interleaving sizes (I_A and I_B) as illustrated in:.

$$Cycles = I_A * I_B * \frac{S_C}{S_D} \tag{4.2}$$

Given the blocks sizes of A (S_AxS_C), B (S_BxS_C) and C (S_AxS_B) and Equation 4.2, the read and write bandwidth requirements can be calculated directly using:

$$Bandwidth = f\frac{Bytes(S_AS_C + S_BS_C + S_AS_B)}{Cycles} \tag{4.3}$$

where f is the operating frequency of the design and $Bytes$ is the number of bytes used to represent each element of A, B and C.

Using Equation 4.3, the optimal configuration of the systolic array can be calculated, given the overall bandwidth of the system and the desired operating precision. Since the required bandwidth is a function of the A, B and C blocks sizes, S_A, S_B and S_C

respectively, modifying either the number of rows or columns, or common dimension buffer size can significantly change operating characterises for given number of DSPs. For 'skinny-tall' matrices increasing the number of columns, whilst decreasing the number of rows results in significant improvements in overall efficiency as demonstrated later in Section 4.8.2.

4.8 Implementation & GEMM Evaluation

This section covers the GEMM portion of the evaluation. The FPGA is compared against a state-of-the-art high performance GPU in terms of both raw performance, measured in tera-operations per second (TOPs), and performance per watt, measured in TOPs per Watt (TOPs/Watt). The hardware template is written in SystemVerilog and the API in C and C++. The FPGA available in the Intel HARPv2 is an Arria 10 GX1150 with 427.2K ALMs (1.150M logic elements (LEs)), 1518 hard DSP blocks and 2K M20K memory blocks. Synthesis and 'place and route' were performed using Quartus Prime Pro version 15.1. All FPGA results were gathered by measuring the execution time of the function call for each configuration on the HARPv2 system. The GPU results were measured in a similar fashion. The power of the FPGA is measured by a tool provided by the HARPv2, measuring the power consumption of the Arria 10. For the GPU, an NVIDIA power profiling tool is used to collect the power consumption over the workload.

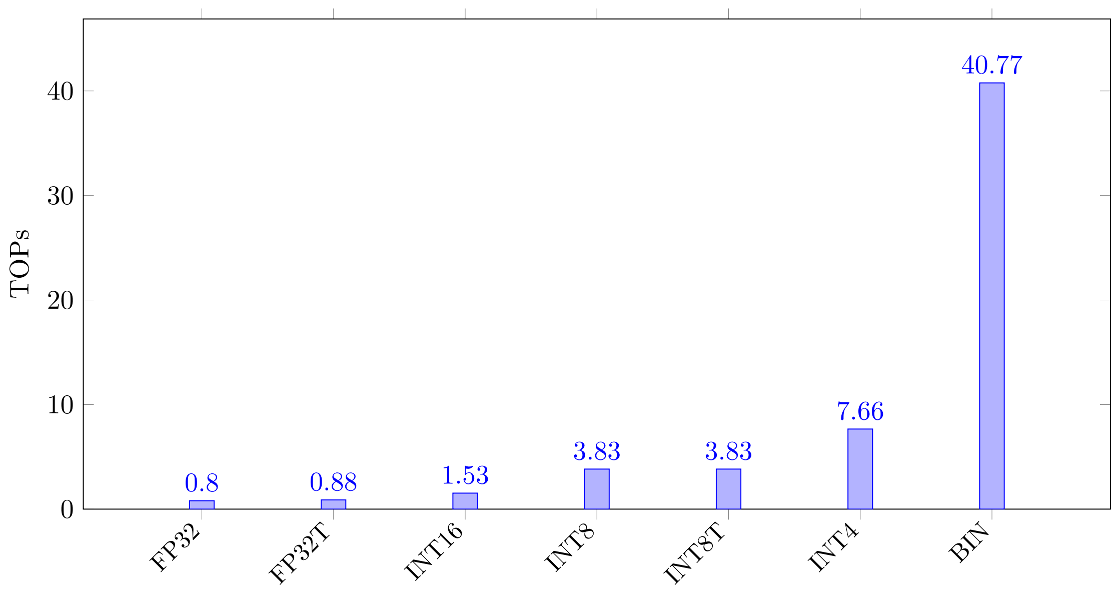


FIGURE 4.9: Arria 10 Peak Performance: the peak performance of the framework for a few selected precisions.

Since the systolic array size is configurable at compile time, it can be tailored to the resources available on the FPGA. In the case of the HARPv2, all of the available area was used to implement the hardware template minus the area taken by the Intel BBS. For FP32, this results in 160 PEs, with an array size of 10 rows and 16 columns, operating at 312.5MHz. With the dot product size configured to 8, this uses 1280 of the available 1285 DSPs, since 233 of the 1518 DSPs are used by the BBS. Even though the measurements are performed on the HARPv2, the hardware template is only dependent on the CCI interface and can be configured for larger or smaller FPGAs.

Figure 4.9 presents the performance of the GEMM architecture on the Arria 10. For all FP32 modes, a floating-point addition and multiplication is needed, hence all operations are performed in the DSP. While it is possible to implement these operations in logic, it quickly becomes very expensive as the design is constrained by routing resource when increasing the array size. Additionally, for lager bit widths such as FP32, the performance is limited by the number of DSPs available on the chip. For FP32T (FP32×Ternary) further optimisations can be made by removing the multiplication and implementing a simplified multiplexer unit.

For integer precisions a better than linear scaling of performance is observed. As the bit width becomes smaller, the dot product is a good fit for the FPGA architecture. INT16 doubles the FP32 performance since each DSP can perform two multiplications and two additions. The hardware template supports using logic resources when implementing a larger array. However specifically for INT16, the multiplication utilisation and routing resources become a significant issue and only result in a small improvement in the peak TOPs. Since the DSP architecture does not natively support 8 bit operations, doubling the performance is achieved by using a dot product built out from both DSPs and logic elements. For INT8 and INT4, extra rows were added to the array since there was sufficient logic resources available to implement additional math operations. Moving to INT8T (INT8×Ternary) provides a performance per watt improvement over INT8 since:

- Each multiplication is replaced by a multiplexer.
- With the removal of the multiplication, the accumulator bit width can be reduced.

The BIN GEMM is implemented completely in logic and uses an XOR and lookup-table based dot product, presented in Section 4.9.2.

4.8.1 GPU Comparison

GPUs are known for their linear algebra performance as shown in Figure 4.10, reported performance from previous studies [12, 92]. The GPU compared against in this thesis

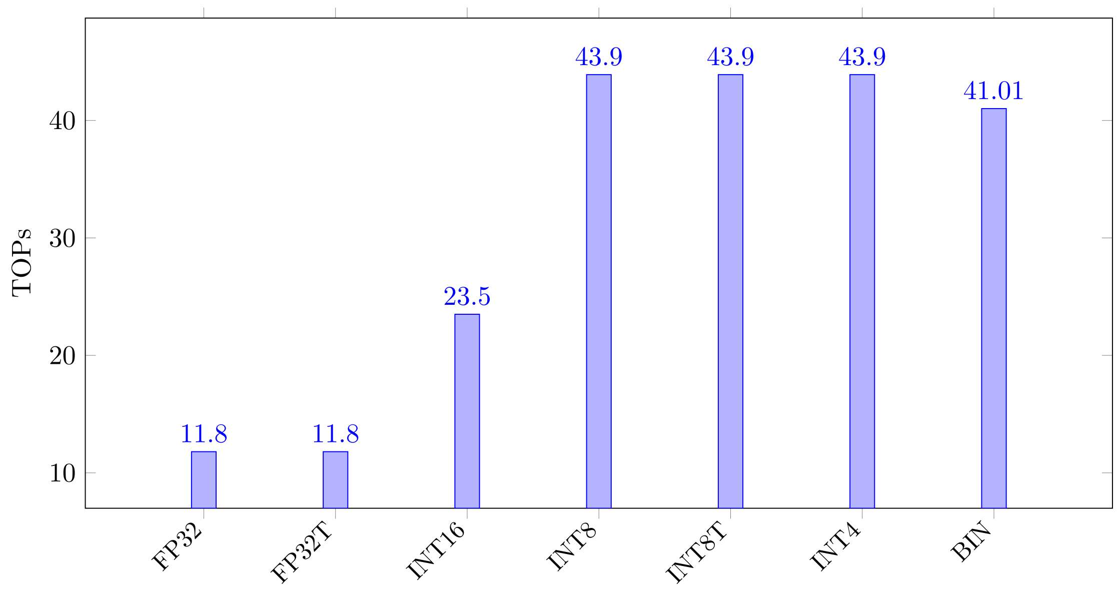


FIGURE 4.10: NVIDIA Pascal Titan X Peak Performance

is the NVIDIA Titian X P102. Where possible, the GEMMs are performed using the optimised CUDA BLAS library, otherwise optimised implementations have been used from previous studies [12, 92]. The matrices are blocked and loaded in the GPU. For the binarised GEMM computation, the population count instructions are supported in the GPU via the *__popc*() for 32 bit and *__popcll*() for 64 bit. A streaming multiprocessor (SM) in an NVIDIA Titan X can issue 32 *__popc*() instructions every cycle, resulting in 1024 binary ops per cycle. At an operating frequency of 1.531GHz and 28 SMs, the

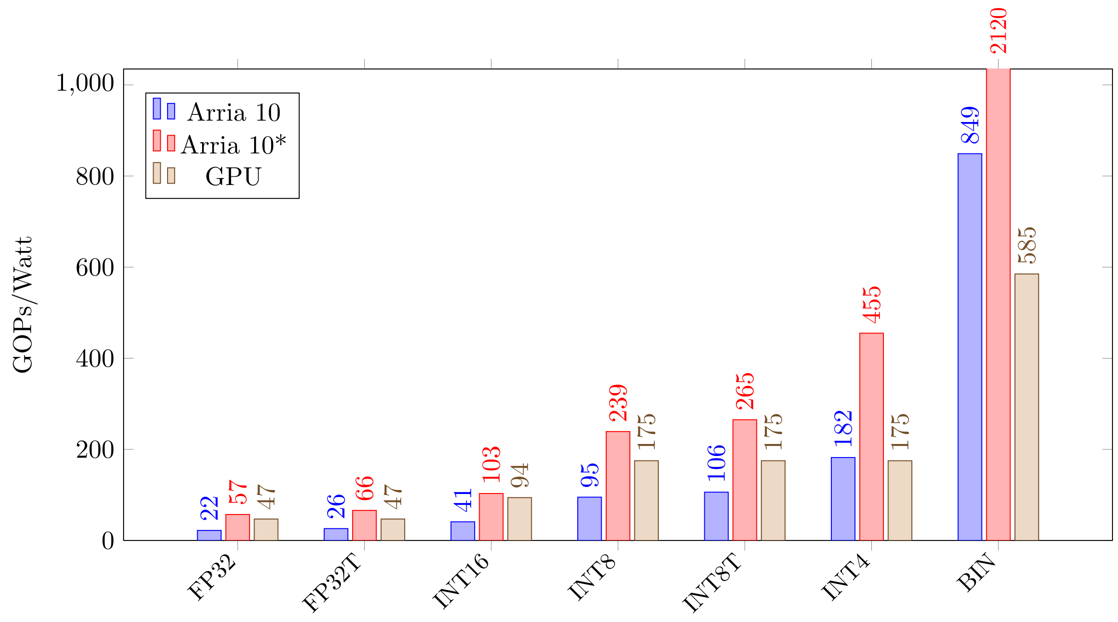


FIGURE 4.11: Performance per Watt: the performance per watt for both the GPU and Arria 10 as well as *modified Arria 10 results which normalise for process node.

peak GPU performance is 43.89 TOPs.

Compared to the Arria 10, the GPU has higher raw performance for all cases apart from the binary GEMM. Considering power, the FPGA shows superior performance for BIN and INT4 which are not a good fit for the GPU architecture. However, considering that the GPU is at a newer processes node compared to the FPGA, TSMC 16 nm and 20 nm respectively, a normalised Arria 10 result of the same design and frequency has been plotted, with power requirements cut by 60% [93]. In the normalised case, the FPGA outperforms the GPU especially for the binary GEMM.

4.8.2 Memory Interleaving

Efficiency is calculated by comparing the measured TOPs to the theoretical maximum value for a given design. The theoretical TOPs is calculated by taking the number of compute units and multiplying by the frequency, disregarding any time for data transfer. Figure 4.12 and Figure 4.13 illustrate the efficiency of the array at different matrix sizes as well as the improvements from the memory interleaving optimisation. The sizes tested were square matrices of the x axis labels, i.e., for 256 the A, B and C matrices are all 256×256. Regardless of matrix size, each precision required the same power as the test case presented in Figure 4.11.

For the smallest matrix size, 256, the efficiency for the unoptimised FP32, INT16 and INT8 designs (D) are all below 20%. In these cases the low efficiency is caused by an

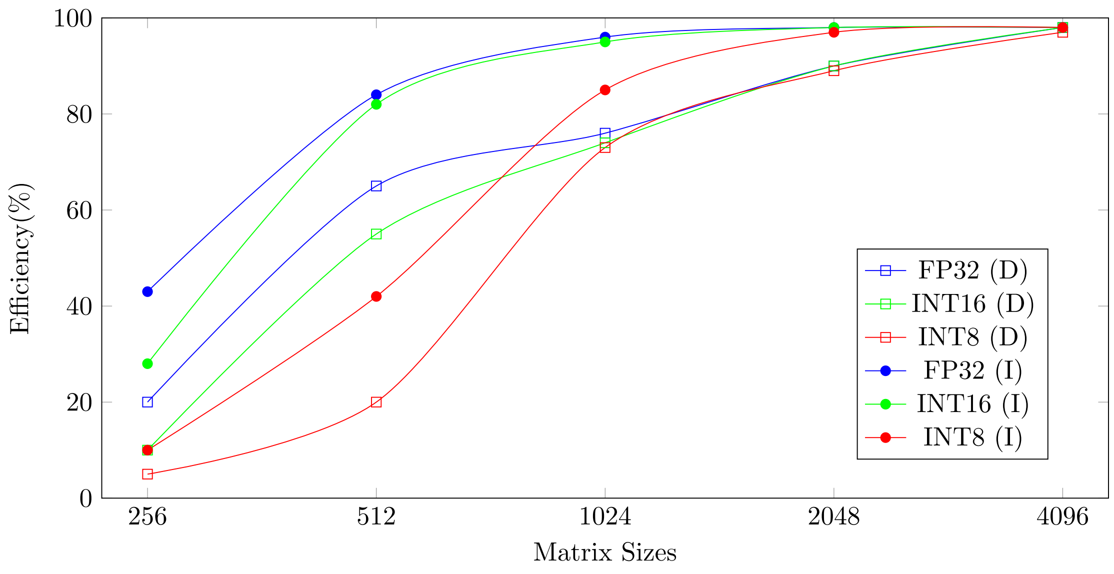


Figure 4.12: GEMM Efficiency vs Matrix Size: illustrates the difference between the framework with (I) and without (D) memory interleaving.

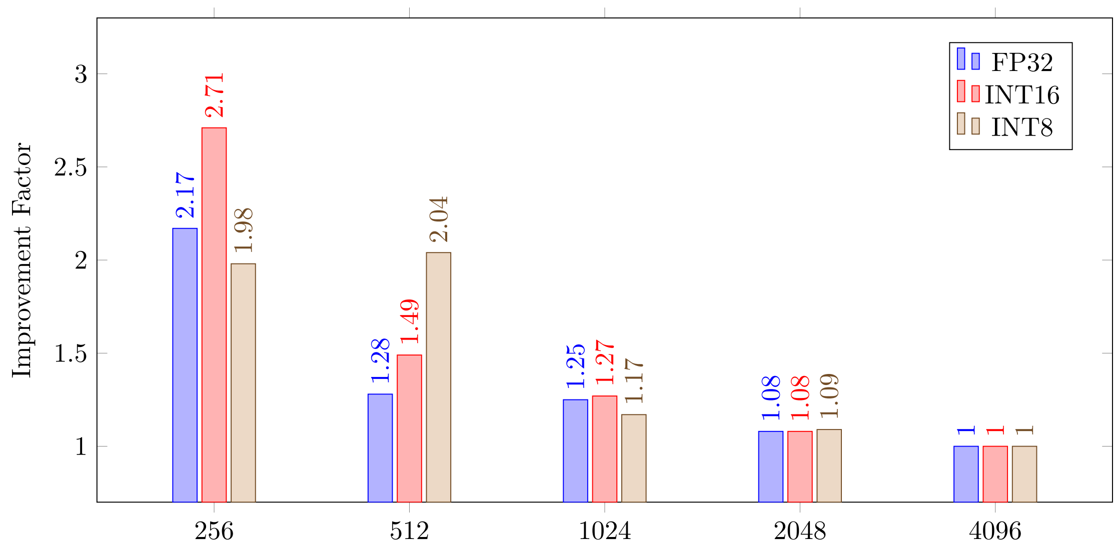


FIGURE 4.13: Memory Interleaving Improvements

inefficient use of blocking, thus memory interleaving alleviates these issues as per Equation 4.3. With runtime configurable memory interleaving (I), presented in Section 4.7.3, a 2.7x improvement in efficiency for the smaller matrix sizes is observed. In some cases even with memory interleaving, the efficiency of the design is quite low, specifically the 512 case for INT8. Usually the data transfer of the next chunk in the input matrix is hidden during the computation of the previous chunk. For smaller matrix sizes, the number of chunks are small and hence the transfer cost cannot be amortised by the compute. Therefore, the initial transfer time for the first chunks of A and B as well as the transfer of C account for the majority of the measured execution time. This issue can be resolved by staging multiple GEMMs such that transfer of the A, B and C chunks overlap with the compute from the previous GEMM. This is similar to how larger single GEMMs are performed. For matrix sizes past 512, the efficiency for all precision is above 80% and becomes 90% past 1024 where it quickly reaches peak performance. For square matrix size 4096 the effectiveness of memory interleaving has been diminished, however as discussed later in Section 4.10.2, non-square matrix sizes still see significant improvements.

4.8.3 Heterogeneous Load Balancing

Figure 4.14 presents the performance of the FP32 GEMM when load balancing is performed over the FPGA and 14 core Xeon CPU. Peak performance is achieved as the work split approaches 60% on the FPGA and 40% on the CPU. In this configuration the FPGA and CPU contribute similar peak performance. This is consistent for most block sizes apart from 20480. In this case, the whole compute is performed on either the CPU or the FPGA. Interestingly, for this particular workload the optimal partitioning size

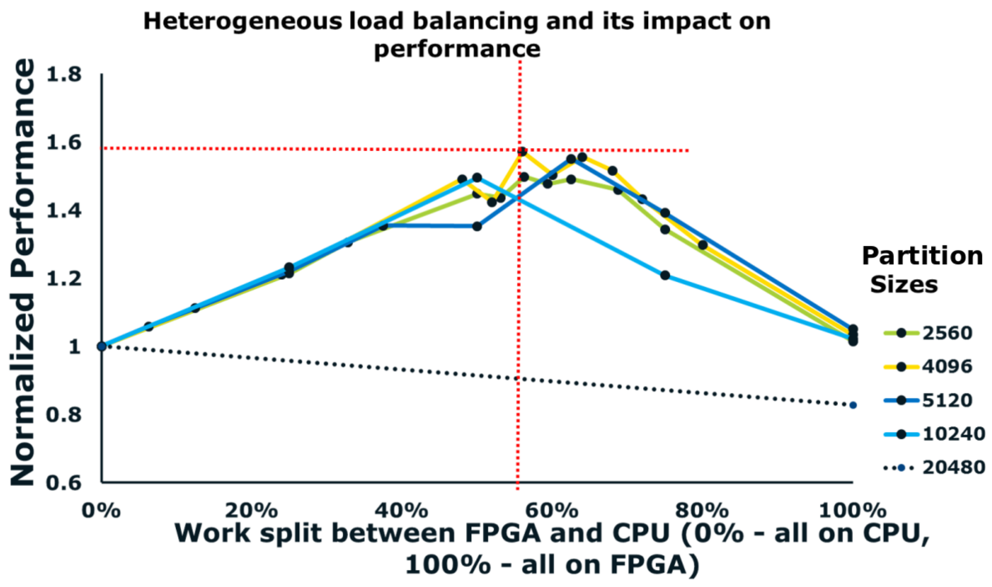


FIGURE 4.14: Heterogeneous Load Balancing

is 4096, showing that a finer-grained partitioning of the problem performs better than coarse-grained sizes such as 10240. This illustrates that the CPU and FPGA working in tandem can achieve a 1.6x improvement in performance over a 14 core only implementation. While the results shown in Figure 4.14 are only for FP32, the same software is used for the lower precisions, however at the time an optimised low precision GEMM was not available for the CPU.

4.9 Deep Learning Optimisations

This section discusses deep learning specific optimisations and hardware template features available in the framework. Specifically, the fused operations and dynamic dot product modules from Figure 4.4 are described. Results for three different binary network topologies, (AlexNet [8], VGGnet [18] and ResNet [4]) are presented and are evaluated against the GPU. The impact of memory interleaving on AlexNet at various batch sizes for FP32 is illustrated and further possible optimisations are discussed. Finally, several possible mixed precision implementations are investigated and their performance for both training and inference is evaluated.

4.9.1 Fused Operations

In the context of deep learning, often additional operations such as the activation function or batch normalisation are performed after the FC or CONV layers.

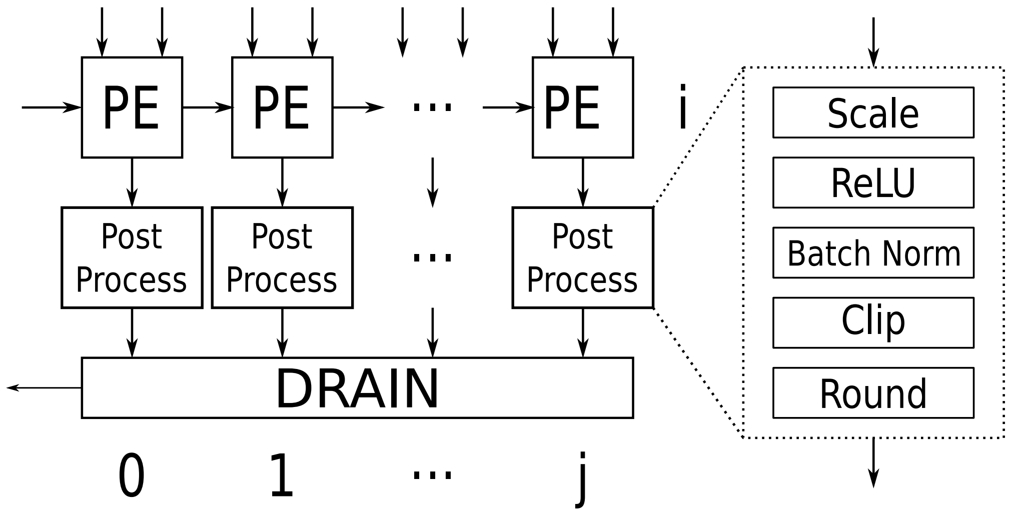


FIGURE 4.15: Fused Operations: The post processing module is configurable at compile time and runtime, and provides key neural network functions.

Presented in Figure 4.15, as the results are drained out of the systolic array, a post processing module can apply some basic operations. The modules are written such that they can be enabled at compile time, with a controllable bypass at runtime. For example, if only the GEMM function call is made, then any additional functions active in the post processing module will be bypassed to maintain the integrity of the GEMM results. The post processing module contains a common interface so that extra functions such as sigmoid, tanh or custom scaling schemes can be added without modification to the systolic array.

4.9.2 Binarised Dot Product

The architecture for the binarised dot product is presented in Figure 4.16. As described in Sec. 4.4.1 the weights and activations are represented as $+1$ and -1, however to take advantage of the XNOR and bitcount operations, -1 is represented as a 0 in binary.

Every cycle, two new input vectors are fed into the dot engine, the XNOR is performed and the result is partitioned across multiple look-up tables. The look-up tables perform a bitcount of their respective inputs and the result is fed into an adder reduction tree. After the reduction, any partial result passed in from the PE cache is accumulated and stored back into the PE cache for further computation.

4.9.3 Mixed Precision Neural Networks

Training of neural networks on FPGA hardware has been challenging since typically the gradient update step needs to be performed in higher precision. Additionally, in the case

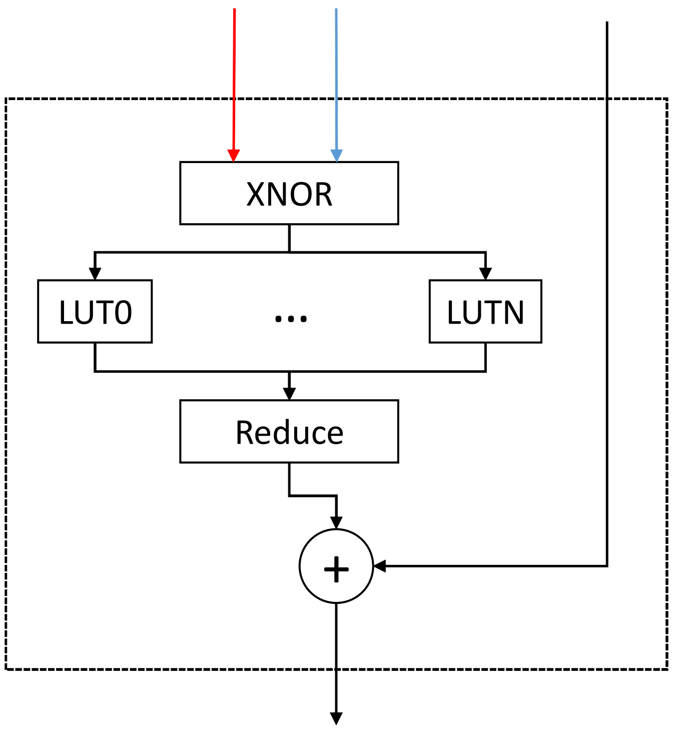


FIGURE 4.16: Binarised Dot Product: the two inputs are passed to the XNOR. The partial result is accumulated with result from the REDUCE module.

of convolution neural networks, the backpropagation stage, described in Section 2.5.4, is performed using matrix multiplications which are laid out differently to convolution in most FPGA architectures. Since most FPGA implementations to date have focused on only supporting native convolution, this restricts them to inference. On the Intel HARPv2, it is possible to stage the computation such that the FPGA performs the forward pass and the CPU the backwards pass. Specifically, The FPGA handles performing the reduced precision CONV and FC layers, allowing the CPU to perform the full precision batch normalisation/local response normalisation, SoftMax, gradient update step and the first and last CONV/FC layers. The working sets of these steps is staged such that both the CPU and FPGA can be fully utilised at all times. Inference is mainly handled by the FPGA with the CPU taking care of the normalisation and SoftMax layers. By removing the need to handle all possible layers on the FPGA, the architecture can be streamlined and best utilised for a single compute function.

From a framework perspective, such as TensorFlow or Caffe, a simple API call is provided by a shared library that performs either a CONV, FC, CONV-ReLU, FC-ReLU or GEMM. In terms of hardware, all necessary data transformations are performed either before or during the memory transfer; this way all complexity of managing the FPGA is hidden and abstracted away. As a result, the framework can decide which implementation is best for a particular section of the topology. In the case of the binarised CONV and FC layers, the scaling is performed either inlined by the CPU as the results are transferred from the FPGA into system memory, or by the fused post processing operations. Additionally, performing a sigmoid activation function instead of ReLU is handled by disabling the fused ReLU operator in the FPGA via a memory mapped register. Followed by performing the sigmoid functions on the Xeon as the results are streamed back into the Xeon's cache.

4.9.4 Dynamic Dot Product

In addition to mixed precision computation across both the FPGA and CPU, a dynamic dot product designed as a direct replacement for the traditional dot product is available in the hardware template. Recent work [92] has shown that through the use of a novel quantisation scheme, hardware friendly backpropagation can be supported via a mixed precision FP32x(Binary/Ternary/Integer) dot product. To support this, the ability to dynamically switch between dot product types during runtime was added. Figure 4.17 presents the necessary change to the PE. To illustrate the advantages of this change, a BIN×BIN dot product in DOT 1 and a FP32xFP32 dot product in DOT 2 was implemented. With the addition of dynamic dot product switching, both training and inference can be supported on the FPGA.

For a typical layer in a state-of-the-art BNN [92] the FPGA performs the BIN × BIN operations very efficiently, hence the CPU can be freed during that time to perform other tasks in its pipeline. Table 4.2 illustrates the required operations of the middle and end portions for a binarised AlexNet. It shows that different layers can be partitioned, for both forward and backward passes, across the FPGA, CPU or using heterogeneous load

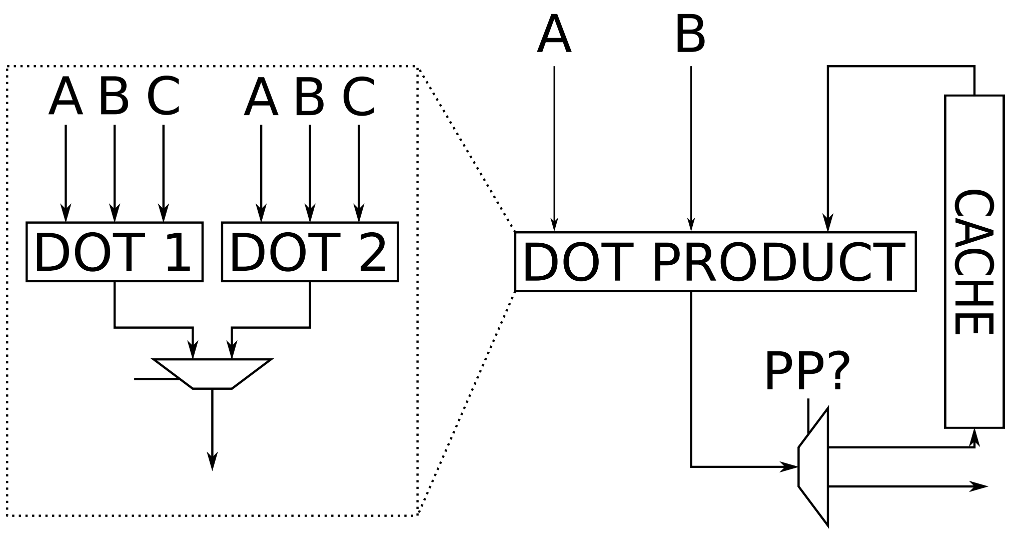


FIGURE 4.17: Dynamic Dot Product Switching: Dynamic dot product switching allows for greater micro-architecture exploration and flexibility when designing for reduced precision networks.

Table 4.2: Mixed Precision Inference and Training

Layer	Location		Type	
	Forward	Backward	Forward	Backward
...	...	...	...	...
conv	FPGA	FPGA+CPU	BIN×BIN	FP×BIN
c&r	FPGA	N/A	INT	STE
relu	FPGA	FPGA+CPU	INT	FP
norm	FPGA	CPU	FP	FP
pool	CPU	CPU	FP	FP
...	...	...	...	...
fc	CPU	CPU	FP×FP	FP×FP
prob	CPU	CPU	FP	FP

For the standard configuration of a BNN, a mixed precision implementation on the Xeon+FPGA utilising the dynamic dot product could operate in this manner. The straight through estimator (STE) is used for the clipping and rounding (c&r) layer, hence no operation is required on the backwards pass.

balancing (FPGA+CPU). The ReLU operation during the forward and backward pass can be performed on the FPGA or CPU depending on which device the result was calculated on. For the batch norm operation the forward pass can be performed on the FPGA as it is a scale and shift operation.

4.10 Deep Learning Evaluation

This section evaluates the framework on three deep learning workloads: AlexNet [8], VGGnet [18] and ResNet [4]. While the GEMM targets many different precisions, binary neural networks were chosen specifically since:

- From Figure 4.9, these clearly offer the best performance over a GPU.
- Recent work [92] has shown that implementations can achieve high accuracy even for binary weight and activation networks.

Additionally, a study on the effectiveness of memory interleaving on layer efficiency is provided, focusing on the AlexNet topology running in FP32. Finally, a mixed precision training and inference scheme is presented, targeted at leveraging heterogeneity and the dynamic dot product module.

The evaluation was performed on both the CONV and FC layers for inference with the standard mini-batch size used for each topology. The total network performance and Images per Second (IPS) is calculated based on the weighted average of the layer's operation contribution to the overall network. A runtime breakdown of each topology was collected using the 14 core Xeon Broadwell CPU running Caffe with Intel MKL2017.

4.10.1 Binary Network Performance

Theoretically, the FPGA is capable of performing 131072 binary ops per cycle. Running at 312.5MHz the peak FPGA performance is 40.96 TOPs. As shown in Figure 4.9, Figure 4.10 and Figure 4.11 the FPGA and GPU achieve 40.77 TOPs and 41.01 TOPs for the binary GEMM respectively, which is within 99% of their theoretical peak performance. Compared to the GPU, the FPGA achieved 849.38 GOPs per Watt which results in a 1.44x improvement in energy efficiency over the GPU at 585.86 GOPs per Watt.

4.10.1.1 AlexNet

As illustrated in Table 3.6, the FPGA, without load balancing, achieves 83% and 86% of the GPU raw performance for AlexNet and VGGNet. When measuring power, the FPGA provides a 1.1x improvement in average energy efficiency over the GPU. However as illustrated in Table 4.3, taking into account the distribution of the compute and the estimated IPS, the FPGA is within 99% of the GPU IPS and provides a 1.34x improvement in energy efficiency for inference. Compared to the CPU, both the FPGA and GPU provide a 2.6x improvement in IPS for inference, while the FPGA is 2.5x more

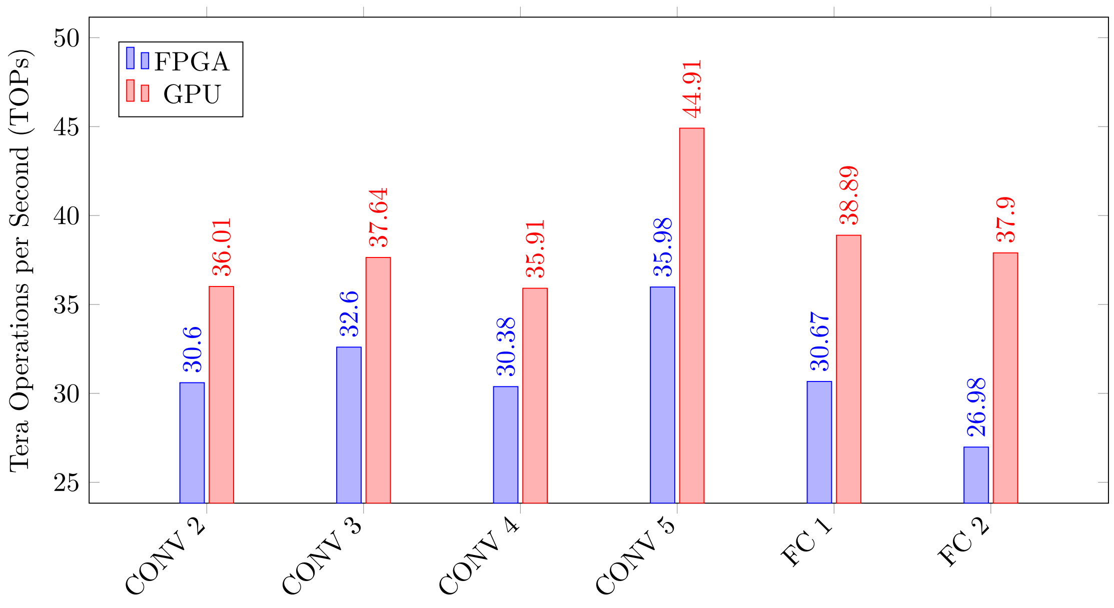


FIGURE 4.18: AlexNet Layer Performance

Table 4.3: AlexNet Estimated Topology Performance

	CPU(I)	FPGA(I)	GPU(I)	CPU(T)	FPGA(T)	GPU(T)
IPS	607.8	1610.4	1626.6	286.9	406.3	407.3
IPS/Watt	3.68	33.5	24.5	1.74	8.46	6.14

This shows results for both Inference (I) and Training (T)

energy efficiency compared with the GPU. When training is considered, the FPGA performance is within 99.7% of the GPU performance whilst providing a 1.37x improvement in energy efficiency. During training the CPU is capable of 286.9 IPS which is within 70% of the FPGA and GPU performance, however the FPGA and GPU provide 4.8x and 3.5x better energy efficiency respectively. This demonstrates that it is critical to consider the problem at the framework level as raw TOPs do not provide the full picture.

4.10.1.2 VGGNet

For VGGnet, presented in Figure 4.19, the FPGA is within 87% of the GPU on average. However, when looking at the middle layer of VGGnet, CONV 5, 6, 7, 8, 9, 10, 11 and 12 are all within 99% or even better than the GPU implementation. When power is considered, the FPGA provides a 1.05x improvement over the GPU. As illustrated in Table 4.4, the FPGA again achieves within 95% of the IPS of the GPU for inference and is 1.35x times more energy efficient over the entire topology. Compared to the CPU, the FPGA and GPU are 8.5x times faster and 20x time more energy efficient in

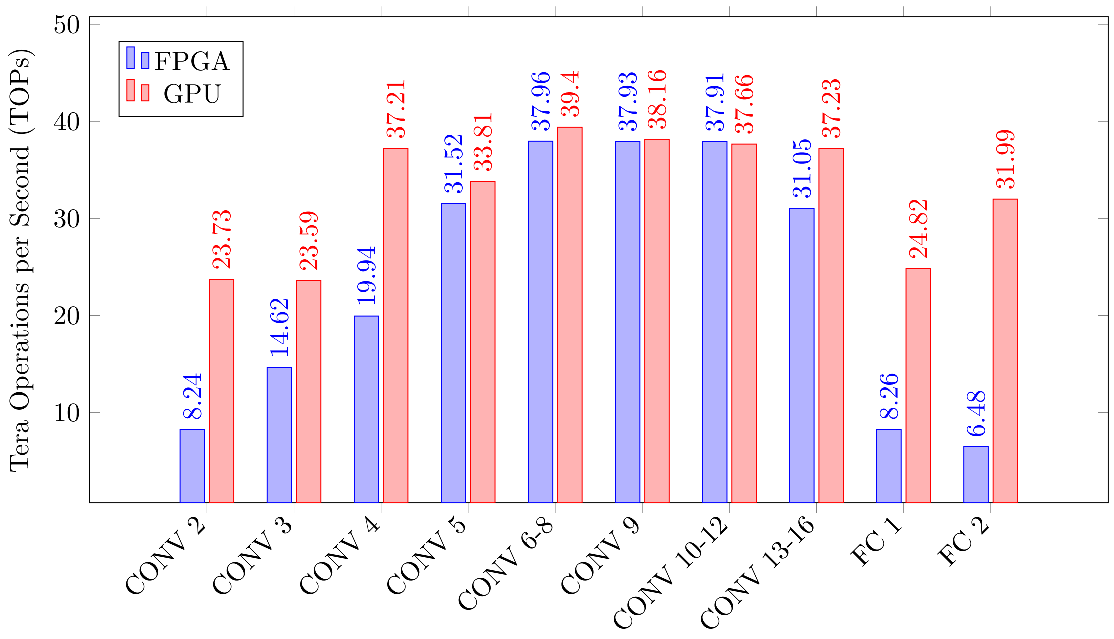


Figure 4.19: VGGNet Layer Performance

TABLE 4.4: VGGNet Estimated Topology Performance

	CPU(I)	FPGA(I)	GPU(I)	CPU(T)	FPGA(T)	GPU(T)
IPS	14.2	114.8	121.45	6.03	9.89	9.94
IPS/Watt	0.09	2.39	1.76	0.04	0.21	0.14

This shows results for both Inference (I) and Training (T)

inference, however for training the difference is less pronounced at 1.6x and 3.8x for IPS and IPS/Watt respectively; since 90% of the computation time is taken by the backward step. However, for training the FPGA is still within 99.5% of the GPU performance and provides a 1.43x boost in energy efficiency.

4.10.1.3 ResNet

For ResNet-34 the FPGA only achieves 70% of the GPU performance at 23.47 and 33.34 TOPs respectively. However it is on par for energy efficiency at 489.05 GOPs/Watt for the FPGA and 485.68 GOPs/Watt for the GPU, showing a 1.03x improvement. This drop in FPGA performance compared to AlexNet can be understood by examining the layer breakdown presented in Figure 4.20. The GPU outperforms the FPGA for the first three layer sets which contribute 50% of the total operations. While increasing the batch size can alleviate some of this discrepancy, this is undesirable since it can affect training rate and total execution time.

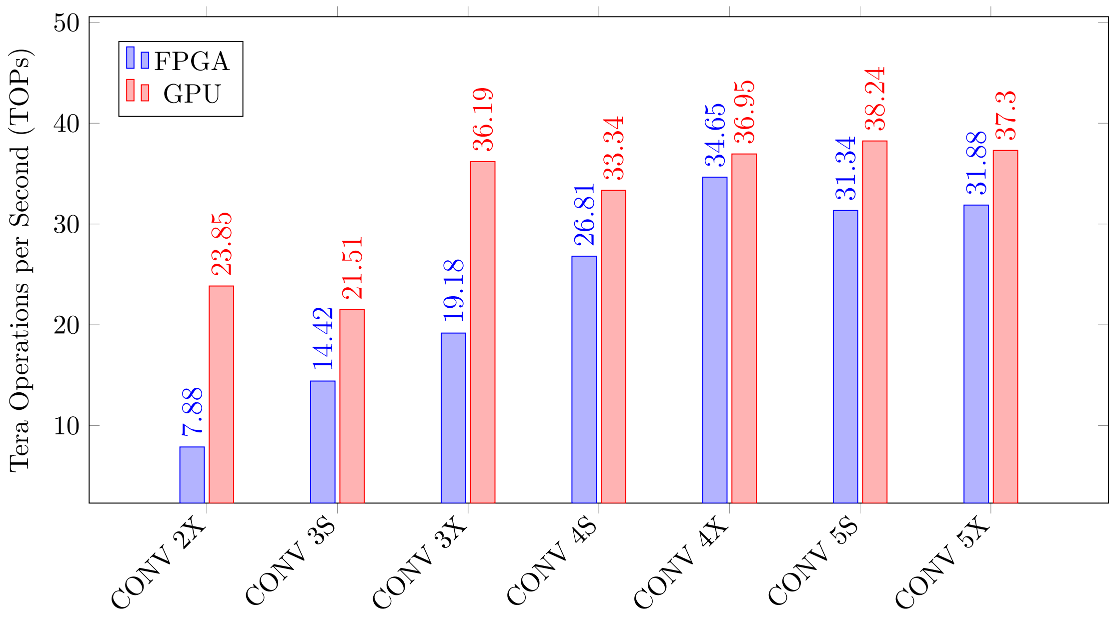


FIGURE 4.20: Binary ResNet Layer Performance

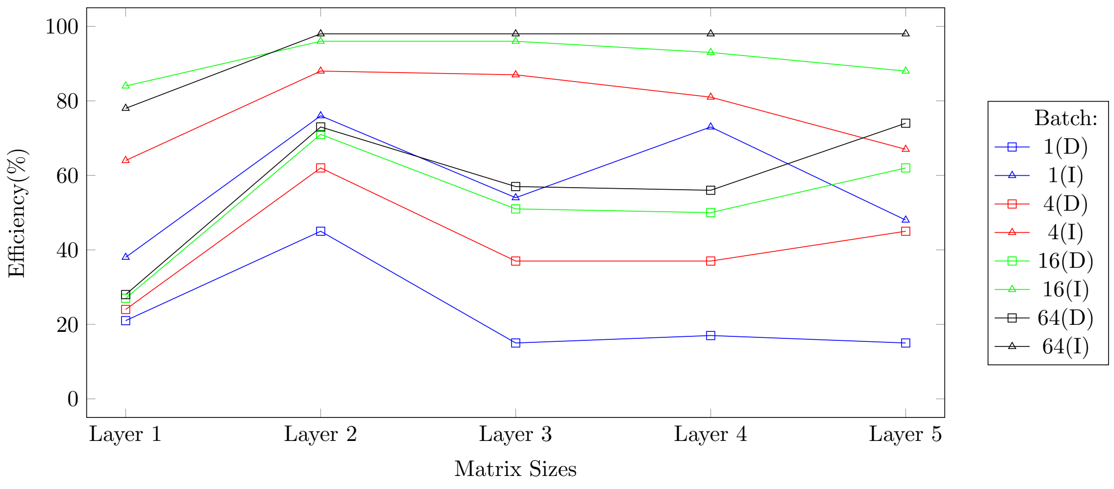


Figure 4.21: FP32 AlexNet Efficiency

Examining the first layer specifically, the FPGA achieves 7.88 TOPs, which is 19% of the measured peak performance presented in Figure 4.9. The main cause of this inefficiency is introduced by the padded zeros in the common dimension of the input matrices.

4.10.2 Effect of Memory Interleaving on AlexNet

Figure 4.21 and Figure 4.22 show the performance of the first five CONV layers in AlexNet for FP32. Different batch sizes ranging from 1 to 64 illustrate the performance improvement that is provided by memory interleaving. For a batch size of 1, A and

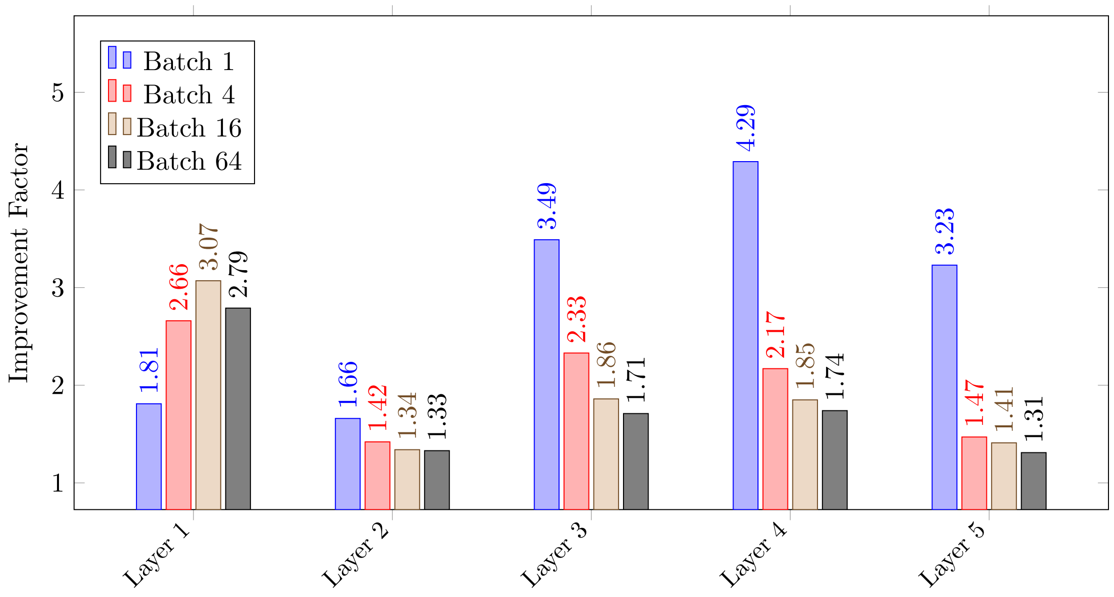


Figure 4.22: FP32 AlexNet Efficiency Improvement

Figure 4.23: A Breakdown of the static design (D) vs configurable interleaving (I) for different batch sizes on AlexNet at FP32 precision.

B are long-skinny/short-fat matrices. Hence with fine control of the interleaving, 1.8x and 1.6x improvement for the first two layers is observed, while for layer 3,4 and 5 the improvement is over 3x and up to 4x. For a batch sizes of 4, 16 and 64, the A and B matrices are becoming increasingly square. Hence the same level of improvement isn't reached, however it achieves near peak performance. The improvement for the first layer is more significant than the others since the first layer exhibits the worst long-skinny/short-fat matrix sizes. However it is clear that memory interleaving significantly improves the efficiency, achieving a 1.3x up to 4x improvement.

4.10.3 Dynamic Dot Product

As discussed in Section 4.9.3, mixed precision GEMMs are needed to handle both the forward and backwards pass. Typically, the first CONV and last FC layers are performed at full precision while the inner layers are performed at lower precision [92]. As shown in Table 4.2, during the backward pass, the gradients are computed using single precision floating-point hence the operation is a FP32×BIN GEMM. While the framework contains an API for switching between different precisions (Section 4.6.1) the latency is determined by the partial reconfiguration time. In the cases where the PR latency is too high, a dynamic dot product may be more appropriate.

By examining the breakdown for binarised AlexNet, similar to Table 4.2, three FPGA implementations were designed, two with dynamic dot product, targeting different portions of the topology: (1) a single BIN×BIN, (2) a BIN×BIN and FP×FP, (2H) a version of (2) that performs the FP×FP GEMM utilising heterogeneous load balancing (FPGA+CPU) and finally (3) a BIN×BIN and FP×BIN. All other layer operations not supported are assumed to be implemented in software on all cores of the Xeon.

TABLE 4.5: Implementation Peak Performance

Impl.	BIN×BIN (TOPs)	FP×FP (TFLOPs)	FP×BIN (TFLOPs)	Forward (ms)	Backward (ms)	Total (ms)
SW	-	-	-	421	471	892 (1x)
(1)	40.77	0	0	158 (F)	471 (C)	630 (1.41x)
(2)	25.4	0.8	0.8	164 (F)	537 (F)	702 (1.23x)
(3)	25.4	0	0.88	165 (F)	502 (F)	668 (1.33x)
(2H)	25.4	1.4	1.4	163 (F+C)	369 (F+C)	533 (1.67x)

Where possible, the operation was performed on the FPGA (F), the 14 core CPU (C) or using heterogeneous load balancing (F+C).

Implementations (1) and (2) were synthesised for the HARPv2 whereas (3) was extrapolated using the results of (2) and the analysis from Section 4.8. Additionally, (2) is able to perform the FPxBIN operations by representing the BIN values as floating-point. Given that inference and training take $421ms$ and $892ms$ respectively, implementation (1) accelerates the BIN×BIN operations which account for $272ms$ of the execution time (all in the forward pass). This corresponds to 65% and 30% of the total interference and training time respectively. (2) accelerates the most layers with the BIN×BIN and FP×FP operations accounting for $327ms$ (77%) and $735ms$ (82%) of the total execution time. Finally, (3) accelerates the BIN×BIN and FP×BIN operations accounting for $272ms$ (65%) and $593ms$ (66%) of inference and training time respectively.

As illustrated in Table 4.5, for pure inference, implementation (1) achieves the best results, reporting the fastest forward execution time. However for training (Forward and Backward), implementation (2H) and (3) show a greater speed up, achieving 1.67x and 1.41x improvement over a software only implementation. Implementations (2), (3) and (2H) use the dynamic dot product, hence the BIN×BIN performance suffers since a smaller systolic array size is used to accommodate the extra routing resources. Although implementation (2) implements all operations on the FPGA, this reduces the amount of hardware that can be dedicated to the bottlenecks, resulting in the lowest speed up, 1.23x.

The CPU+FPGA implementations (1), (3) and (2H) show that the best overall performance is achieved by leveraging the CPU and specialising the implementations for only the most profitable parts of the algorithm, or by utilising heterogeneous load balancing. Compared to other CPU+FPGA platforms the same network performance would not be achieved since these are often SoC type CPU that have relatively low FLOPS compared with the Xeon CPU.

Table 4.6: Previous Work

	[17]	[94]	[16]	This thesis
Platform	Zynq z7045	Kintex US KU115	Zynq 7Z020	Arria 10 GX1150
Logic Elements (LEs)	350K	1,451K	85K	1,150K
Power (W)	11.3	41	4.7	48
TOPs (Peak)	11.612	14.8	0.32	40.77
MOPs / LE	33.17	10.19	4.43	35.45
GOPs / Watt	1027.68	360.97	44.2	849.38

4.11 FPGA Comparison

Compared to other work, the hardware template in the BIN×BIN configuration achieves the highest peak TOPs, MOPs / LE and the second highest GOPs / Watt. Comparing across different devices and manufacturers is difficult and care was taken to ensure that accurate comparisons are made. While the results reported in [17] are for an older process node and lower frequency, [94] shows their framework performance on a device of similar size and process node as this thesis. It is expected that [17] should achieve better energy efficiency since the device is targeted at low power SoC applications, while the HARPv2 is targeted at data centres. One advantage of the hardware template is that it provides several other data types and customisations that benefit applications outside of deep learning. Additionally, with the Xeon CPU able to provide high floating-point compute power, any changes to the underlying neural network algorithms, such as those presented in [92] can be supported without changes to the hardware architecture. Whilst it is possible to support SoC type FPGAs, the framework was designed to operate on data centre based FPGA.

4.12 Summary

This chapter presented a customisable matrix multiplication framework that includes a simple software API and a hardware template for designing custom GEMM accelerators on the HARPv2. The framework consists of a highly configurable hardware template with a streamlined software stack and runtime application programming interface, which allow for a wide range of different precisions, various core sizes and tuneable runtime configurable parameters. This flexibility, allows the framework to adapt to the users requirements and alleviates performance bottlenecks that effect other systems. Various Deep Learning optimisations were presented, fused operations, dynamic dot product and the binarised dot product, allowing for fine-grained customisations for specific deep learning applications.

The framework was evaluated and it was illustrated that the in-package integrated FPGA can remain competitive with a high-end discrete GPU on raw performance. An evaluation of performance using popular deep neural networks (AlexNet, VGGNet and ResNet) on the HARPv2 platform was performed, using ILSVRC15[61]; and a study on the efficiency of the hardware template and its impact on deep learning performance was presented. Compared to previous FPGA work, the HARPv2 reports the highest TOPs, MOPs/LE and second highest GOPs per Watt. While the GPU provides better performance for particular layers, when considering total topology performance the FPGA achieves similar performance and better energy efficiency. Utilising the Xeon+FPGA, a flexible usage model capable of performing training and mixed precision methods was presented and evaluated.

In the case of binarised neural networks, the FPGA either matched or exceeded the GPUs performance and offered insight into some of the issues faced when designing for this system. Several heterogeneous implementations were evaluated, illustrating that dedicating FPGA resources to accelerating specific bottlenecks and utilising the CPU for other operations, results in higher performance implementations. While the HARPv2 is still an emerging technology, it stays competitive with a high performance discrete GPU by taking advantage of close collaboration between the FPGA and CPU.

The performance of the framework is largely dependent on the number of multiply-accumulates as well as the efficiency of those multiplications. In the next chapter, a novel two speed multiplication algorithm is presented. The multiplier is designed with deep learning and neural networks in mind, taking advantage of bit-patterns to perform less work. This characteristic is particularly useful to a matrix multiplication accelerator, since the most expensive operation is the multiplication.

Chapter 5

Multiplication for Machine Learning

5.1 Introduction

Work on optimising multiplication circuits has been extensive [95], with the modified Booth algorithm at higher radices generally accepted as the highest performing implementation for general problems [96]. Extending upon previous work, this chapter describes a new machine learning operator; a two speed serial multiplication algorithm and implementation. It conditionally adds the non-zero encoded parts of the multiplication, skipping over the zero encoded sections.

As illustrated in Chapter 4, reduced precision representations are often used to improve the performance of a design, striving for the smallest possible bit width to achieve a desired prediction accuracy [97]. At compile time, the precision is fixed and changes in requirements often involve redesign and reimplementation. Datapaths are designed with the largest required bit width to ensure the correctness of each result. Consequently, occasions when the datapath could operate at a lower precision, standard multipliers perform unnecessary computation in the most-significant bits, resulting in lower efficiency. To mitigate this, mixed-precision datapaths have been proposed [98, 99]. These attempt to use lower precision datapaths for the majority of the computation, only switching to a higher precision when necessary. However, these are normally implemented with two separate datapaths, resulting in increased area and reducing overall flexibility.

The two speed (TS) modified Booth multiplication algorithm and implementation presented in this chapter is able to skip over encoded all-zero or all-one computations, independent of location. The multiplier inputs all bits of both operands in parallel and is designed to be a basic building block, easily incorporated into existing DSP block, CPUs and GPUs, resulting in improved computational performance, given the appropriate input set. This chapter starts with an introduction of the radix-4 booth multiplication algorithm and is followed by a description of the two speed optimisation. A key element of the multiplier is that sparsity within both the input set and the number's internal binary representation both lead to performance improvements. The multiplier's datapath is divided into two subcircuits, each operating with a different critical path; it is able to take advantage of particular bit-patterns to perform less work.

5.2 Contributions

The contributions of this chapter are:

- A two speed Booth multiplication algorithm and implementation where the datapath is divided into two subcircuits, each operating with a different critical path.
- Demonstrations of how this multiplier takes advantage of particular bit-patterns to perform less work; this results in reduced latency, increased throughput and superior area-time performance compared to conventional multipliers.
- A model for estimating the performance of the multiplier and evaluation of the utility of the proposed multiplier via an FPGA implementation.

5.2.1 Previous Work on Reduced Precision Multiplication for Neural Networks

The most comparable work to this multiplier is the parallel-serial, or shift-add, multiplier described in Chapter 2. As described in Equation 2.23, the product p is iteratively calculated by examining individual bits of X each cycle and accumulating a scaled Y. An extended version of this multipliation technique is presented in this chapter, the Booth multiplication algorithm [24]. The TS multiplier is compared directly to a Booth version of the parallel-serial multiplier in the results section of this chapter.

Other relevant work in multiplcation circuits involve bit and digit serial approaches and has focused on on-line arithmetic [100] and efficient mapping of the algorithms to the FPGA architecture. Shi et. al. [101] analysed the effect of overclocking radix-2 on-line arithmetic implementations and quantified the error introduced by timing violations. They found a significant reduction in error for DSP based applications compared with conventional arithmetic approaches. Zhao et. al. [102] presented a method for achieving arbitrary precision operations utilising the on-chip block RAMs to store intermediate values.

In the domain of neural networks, Judd et. al. [97] presented a bit-serial approach for reduced precision computation. They showed a 1.3x to 4.5x performance improvement over classical approaches as their arithmetic units only perform the necessary computation for the particular bit width.

5.3 Radix-4 Booth Multiplication

Multiplication can be performed using the Booth algorithm [24]. This section reviews the radix-4 Booth algorithm [24], an extension to the multiplier presented in Section 2.3.2. Fundamentally, multiplication computes $x \times y$ where x and y are n bit two's complement numbers, the multiplicand and multiplier respectively; producing a $2n$ two's complement value in the product p. The multiplication algorithm considers multiple digits of Y at a time and is computed in N partitions:

$$N = \lfloor \frac{n+2}{2} \rfloor \tag{5.1}$$

A parallel version of the algorithm is given by:

$$p = (Y_1 + Y_0)x + \sum_{i=1}^{N} 2^{2i-1}(Y_{2i+1} + Y_{2i} - 2Y_{2i-1})x \tag{5.2}$$

Following the notation in Section 2.3.2, Y denotes the n length digit-vector of the multiplier y. The radix-4 Booth algorithm considers 3 digits of the multiplier Y at a time to create an encoding e given by:

$$e_i = Y_{2i+1} + Y_{2i} - 2Y_{2i-1} \tag{5.3}$$

where i denotes the ith digit. As illustrated in Table 5.1, apart from $Y_{i+2}Y_{i+1}Y_i = 000$ and $Y_{i+2}Y_{i+1}Y_i = 111$ which results in a 0, the multiplicand is scaled by either 1, 2, -2 or -1 depending on the encoding.

This encoding e_i is used to calculate a partial product $PartialProduct_i$ by calculating:

$$PartialProduct_i = e_i x = (Y_{2i+1} + Y_{2i} - 2Y_{2i-1})x \tag{5.4}$$

This $PartialProduct$ is aligned using a left shift (2^{2i-1}) and the summation is performed to calculate the final result p. Since the Y_{-1} digit is non-existent, the 0th partial product $PartialProduct_0 = (Y_1 + Y_0)x$. A serial (sequential) version of the multiplication is

TABLE 5.1: Booth Encoding

Y_{i+2}	Y_{i+1}	Y_i	e_i
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	$\bar{2}$
1	0	1	$\bar{1}$
1	1	0	$\bar{1}$
1	1	1	0

$\bar{2}$ and $\bar{1}$ represent -2 and -1 respectively.

performed by computing each partial product in N cycles:

$$\begin{aligned} p[0] &= 2^{n-2}(Y_1 + Y_0)x \\ p[j+1] &= 2^{-2}(p[j] + 2^n(Y_{2j+1} + Y_{2j} - 2Y_{2j-1})x) \quad \text{for } j = 1, \ldots, N-1 \\ p &= p[N] \end{aligned} \tag{5.5}$$

To better understand the two speed optimisation presented in the next section, Equation 5.6 is represented as an algorithm in Algorithm 5.1 and illustrated in Figure 5.1. Two optimisations are performed to allow for better hardware utilisation. Firstly, the product p is assigned the multiplier y ($p = y$), this removes the need to store y in a separate register and utilises the n least-significant bits of the p register. Consequently, as the product p is shifted right (p=sra(p, 2)), the next encoding e_i can be calculated from the three least-significant bits (LSBs) of p. The second optimisation removes the realignment left shift of the partial product (2^n) by accumulating the *PartialProduct*

Algorithm 5.1: Booth Radix-4 Multiplication: x, y are n bit two's complement numbers, p denotes the $2n$ two's complement result, and sra (the shift right arithmetic function). y is assigned to the n least-significant bits of p, hence the encoding, E, can be calculated directly from P.

```
Data: $y$: Multiplier, $x$: Multiplicand
Result: $p$: Product
1 $p = y$;
2 $e = (P[0] - 2P[1])$;
3 for $count = 1$ to $N$ do
4     $PartialProduct = e * x$;
5     $p = \text{sra}(p,2)$;
6     $P[2*B-1:B] += PartialProduct$;
7     $e = (P[1] + P[0] - 2P[2])$;
```

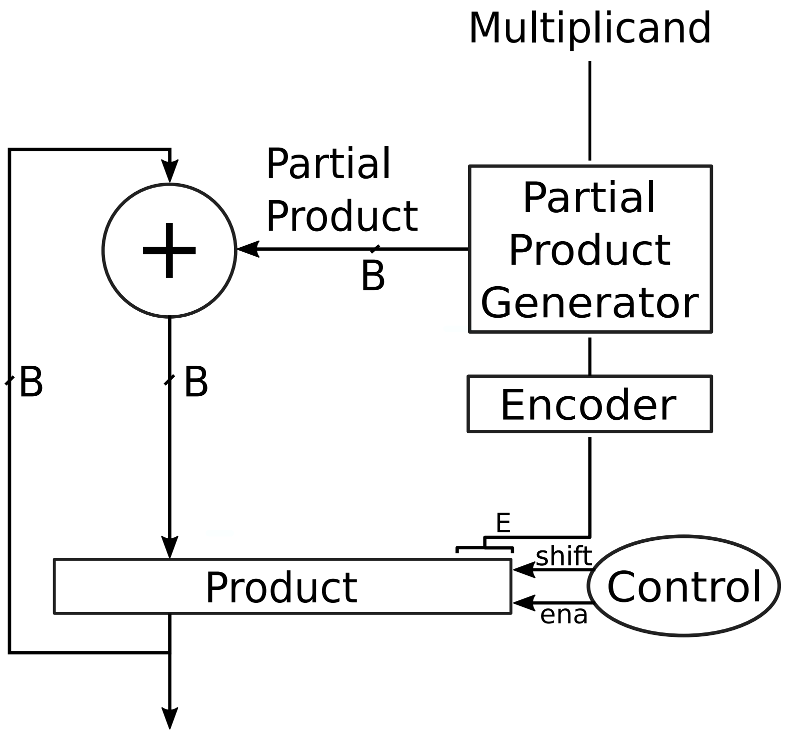


FIGURE 5.1: n bit Serial Multiplier. There are five key components to the standard radix-4 serial Booth multiplier: the shifter, encoder, partial product generator, control and adder. As the partial results are generated in the adder, they are accumulated in the n most-significant bits of the product register.

to the n most-significant bits of the product p ($P[2 * B - 1 : B] += PartialProduct$).

5.4 Two Speed Multiplier

This section presents the two speed extension to the serial Booth multiplication algorithm and implementation. The key modification to the serial Booth multiplier, is partitioning the circuit into two paths; each having critical path, τ and $K\tau$ respectively (see Figure 5.2). This results in the two speed multiplier, demonstrated in this thesis. The multiplier is clocked at a frequency of $\frac{1}{\tau}$, where the $K\tau$ region is a fully combinatorial circuit with a delay of $K\tau$. K is the ratio of the delays between the two subcircuits. $\bar{K} = \lceil K \rceil$ is the number of cycles needed for the addition to be completed before storing the result in the product register; used in the hardware implementation of the multiplier.

As illustrated in Algorithm 5.2, before performing the addition, the encoding, e, (the three least-significant bits of the product) is examined and a decision is made between two cases: (1) The encoding and $PartialProduct$ are zero and $0x$, respectively, or (2)

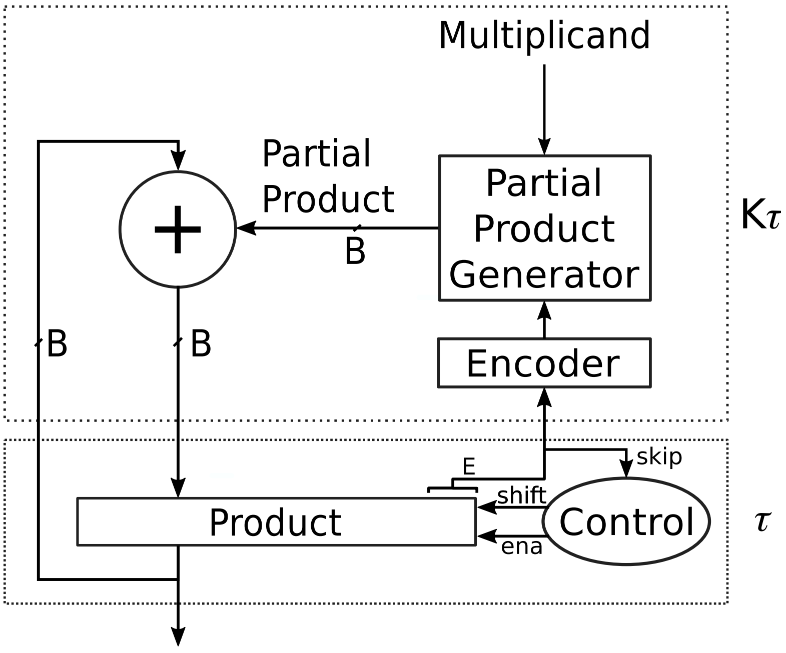


FIGURE 5.2: n bit Two Speed Multiplier. This contains an added control circuit for skipping and operating with two different delay paths.

the encoding is non-zero. These two cases can be distinguished by generating:

$$skip = \begin{cases} 1, & \text{if } P[2:0] \in \{000, 111\} \\ 0, & \text{otherwise} \end{cases} \tag{5.6}$$

When $skip = 1$ only the right shift and cycle counter accumulate need to be performed, with a critical path of τ. In the case of a non-zero encoding ($skip = 0$), the circuit is clocked $\bar{K}$ times at τ. This ensures sufficient propagation time within the adder and partial product generator, allowing the product register to honour its timing constraints.

Algorithm 5.2: Two Speed Booth Radix-4 Multiplication: When $E = 0$, zero encodings are skipped and only the right shift arithmetic function is performed.

```
Data: $y$: Multiplier, $x$: Multiplicand
Result: $p$: Product
1 $p = y$;
2 $e = (P[0] - 2P[1])$;
3 for $count = 1$ to $N$ do
4     $p = \text{sra}(p,2)$;
      // If non-zero encoding, take the $K\tau$ path, otherwise the $\tau$ path
5     if $e \neq 0$ then
          // this path is clocked $\bar{K}$ times
6         $PartialProduct = e * x$;
7         $P[2 * B - 1 : B] += PartialProduct$;
8     $e = (P[1] + P[0] - 2P[2])$;
```

Hence the total time T taken by the multiplier can be expressed as Equation 5.7, where N is defined by Equation 5.1, and O is the number of non-zero encodings in the multiplier's Y digit-vector.

$$T(O) = (N - O)\tau + O\bar{K}\tau \tag{5.7}$$

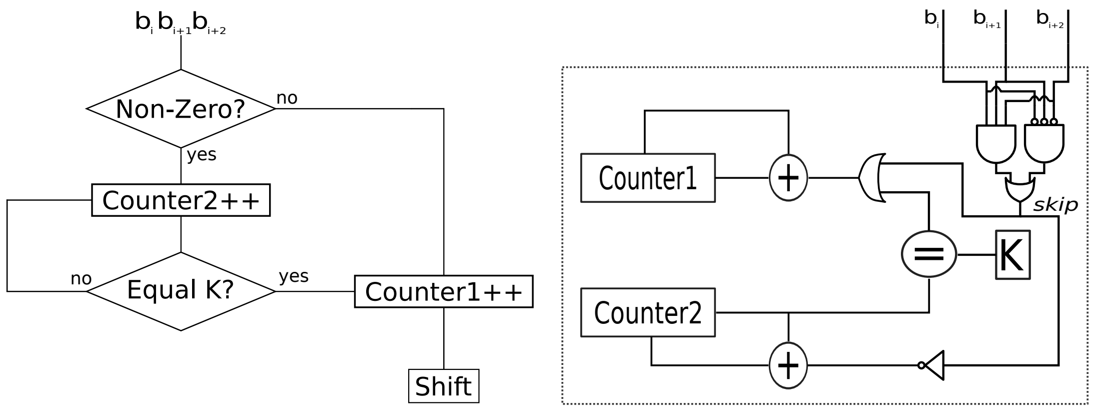
The time taken to perform the multiplication is dependent on the encoding of the bits within the multiplier y, an upper and lower bound can be calculated for the total execution time given $O = N$ and $O = 0$. From Equation 5.7, the max and min are:

$$N\tau \leq T \leq N\bar{K}\tau \tag{5.8}$$

The minimum execution time occurs when $y = 0$, in this case, all bits within the multiplier are 0 and every three LSB encoding result in a $0x$ scaling and $O = 0$. There are a few combinations that result in the worst case, when $O = N$. One case would be a number of alternating 0 and 1, ie. 1010101..10101..10101. In this case, each encoding results in a non-zero $PartialProduct$.

5.4.1 Control

As shown in Figure 5.3b and Figure 5.3a, the control circuit consists mainly of: one $\log_2(N)$ accumulator, one $\log_2(\bar{K})$ accumulator, three gates to identify the non-zero encodings and a comparator. $Counter2$ is responsible for counting the number of cycles needed for the addition without violating any timing constraints, i.e, $\bar{K}$. When the



(A) Controller flowchart (B) Control Circuit

FIGURE 5.3: Two counters are used to determine (1) when the multiplication is finished, and (2) when the result of the $K\tau$ circuit has been propagated.

encoding is non-zero, $Counter2$ is incremented. $Counter1$ accumulates the number of encodings that have been processed. As shown in Section 5.3, the number of cycles needed to complete a single multiplication is N, therefore the accumulator and $Counter1$ needs to be $\log_2(N)$ bits wide. $Counter1$ is incremented when the comparator condition has been met, $Counter2 = \bar{K}$, or a zero encoding is encountered. When $Counter1$ increments, the signal is given to perform the right shift.

The control needs to distinguish between the zero and non-zero encodings. It contains a three gate circuit, performing Equation 5.6; taking in the three LSBs of the multiplier y. Two cases of zero encoding exist. The three gates are designed to identify these non-zero encodings; an inverter is connected to the accumulator of $Counter2$, incrementing, in these cases.

5.4.2 Example

Figure 5.4 provides an example of the control operating in the multiplier and the time taken to perform the multiplication. Each cycle, the three least-significant bits of the multiplier y are examined and an action is generated based on their encoding. Since 000 results in a $0x$ partial product, the first action is a "skip" and only the right shift is performed in τ time. The next three bit encoding, 010, is examined and results in a $1x$ partial product. This generates the "add" action in which $Counter2$ is accumulated to $\bar{K}$ and the product register is held constant. After $\bar{K}\tau$ time, the value stored in the register has had enough time to propagate through the adder and the result is latched in the product register without causing timing violations. The multiplier continues operating in this fashion until all bits of y have been processed and the final result produced. In Figure 5.4, the total time is $3\tau + 3\bar{K}\tau$ since there are three "skips" and three "adds".

Bit Representation	Action	Time	PartialProduct
1 1 1 1 0 1 0 0 0 1 $\boxed{0\ 0\ 0}$	skip	τ	$0x$ X 2^0
1 1 1 1 0 1 0 0 $\boxed{0\ 1\ 0}$	add	$\tau + \bar{K}\tau$	$1x$ X 2^2
1 1 1 1 0 1 $\boxed{0\ 0\ 0}$	skip	$2\tau + \bar{K}\tau$	$0x$ X 2^4
1 1 1 1 $\boxed{0\ 1\ 0}$	add	$2\tau + 2\bar{K}\tau$	$1x$ X 2^6
1 1 $\boxed{1\ 1\ 0}$	add	$2\tau + 3\bar{K}\tau$	$-1x$ X 2^8
$\boxed{1\ 1\ 1}$	skip	$3\tau + 3\bar{K}\tau$	$0x$ X 2^{10}

FIGURE 5.4: Control example: Non-zero encodings result in an "add" action taking $\bar{K}\tau$ time, whereas zero encodings allow the "skip" action, taking τ time. For the first encoding, only the two least-significant bits are considered with a prepended 0 as described in Section 5.3.

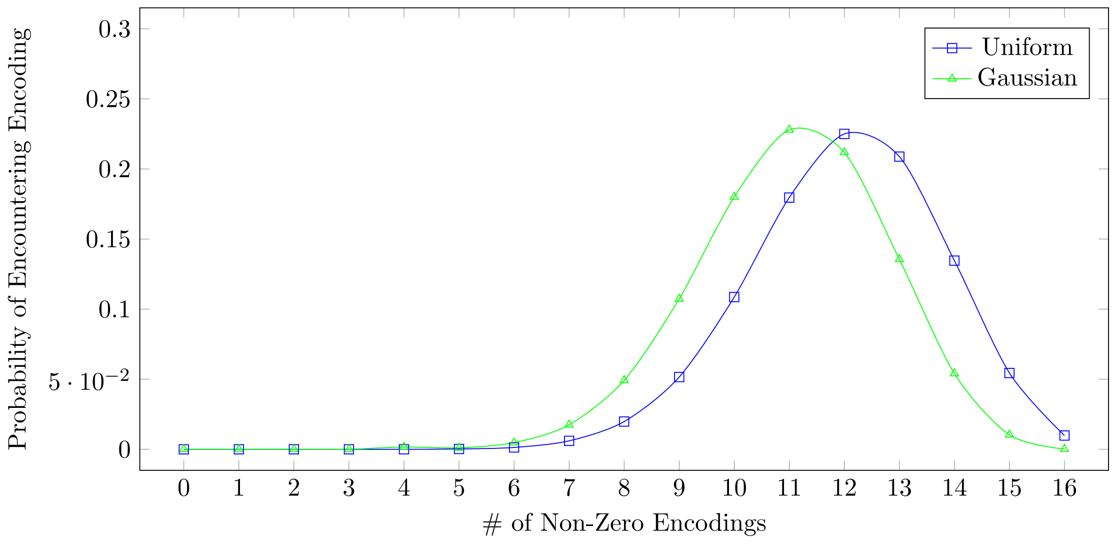


FIGURE 5.5: $p(i)$ 32 bit distribution: the distribution of the frequency of particular non-zero encoded numbers for the Gaussian and Uniform sets.

5.4.3 Set Analysis and Average Delay

Given an input set D of length l and a function $f(y)$ (given by Equation 5.9) that calculates the number of non-zero encodings for a given multiplier y, the probability distribution p of encountering a particular encoding can be calculated by Algorithm 5.3.

$$f(y) = \neg(Y_1 \oplus Y_0) + \sum_{i=1}^{N} (\neg(Y_{2i+1} \oplus Y_{2i}) \wedge \neg(Y_{2i} \oplus Y_{2i-1})) \tag{5.9}$$

where $\neg$, $\oplus$ and $\wedge$ are the logical 'not', 'xor' and 'and' symbols respectively.

Figure 5.5 shows the Gaussian and Uniform encoding probability distribution for 32-bits. There are significantly less numbers in the lower, non-zero encoding region compared with the higher, non-zero encoding region, resulting in increased computation time. However, as discussed in Section 5.5, for other workloads, the sets can shift and change depending on the problem and optimisation techniques used.

Algorithm 5.3: Probability of a Particular Encoding: $\oslash$ denotes the element-wise division.

```
Data: $D$: Input Set
Result: $p$: Product
1 $Count\{0, 1, \ldots, N\} = \{0\}$;
2 for $i = 0$ to $l$ do
3     $Count[f(X_i)] += 1$
4 $p = Count \oslash l$
```

Using the probability p, the average delay of the multiplier can be calculated using Equation 5.10.

$$\mathbb{T} = \frac{1}{N}\sum_{i=0}^{N} p(i)T(i) \tag{5.10}$$

where T is calculated using Equation 5.7 and $p(i)$ denotes the probability of encountering an i non-zero encoded number.

5.4.4 Timing

During standard timing analysis, the $K\tau$ path would flag a timing violation when analysing the circuit at the frequency $\frac{1}{\tau}$. There are two options to address the timing violation. The first involves a standard 'place and route' of each individual multiplier as it is instantiated in the design. An additional timing constraint is included to address the otherwise violated $K\tau$ path, allowing timing driven synthesis and placement to achieve the best possible layout. The second option is to create a reference post-'place and route' block that is used whenever the multiplier is instantiated. This ensures each multiplier has the same performance and is placed in exactly the same configuration.

There are downsides to each option. The first option gives the tools freedom to place the blocks anywhere, however the performance of individual instantiations may differ if the $K\tau$ and τ sections cannot be placed at the same clock rate. For the second option, placing a reference block requires availability of free resources in the layout specified. While this ensures high performance, placing the reference block may become increasingly difficult as the design becomes congested.

5.5 Evaluation

This section presents the results for the TS multiplier. The multiplier is compared against the standard 64, 32 and 16 bit versions of parallel-parallel and serial-parallel multipliers. For all configurations tested up to 64 bits, the K scaling factor in the $K\tau$ subcircuit was always less than two. Allowing the hardware to be further simplified to a bit-flip operation removing the need to compare $\bar{K}$ with an accumulator.

5.5.1 Implementation Results

The area and delay of different TS multiplier instantiations are given in Table 5.2 for an Intel Cyclone V 5CSEMA5U23C6 FPGA. These results were obtained using the Intel Quartus 17.0 Software Suite. During placement and routing the software performs static timing analysis across four different PVT corners, keeping voltage static. Specifically: (1) Fast 1100mv 0C, (2) Fast 1100mv 85C, (3) Slow 1100mv 0C and (4) Slow 1100mv 85C. The TS multiplier was 'placed and routed' using the timing constraint based methodology and all frequencies reported for each multiplier represent the upper limit for each one considered as a standalone module, incorporating these into existing designs may decrease their operating frequency. The $Area * Time$ is calculated by looking at the set of the inputs and using Equation 5.10. Unless otherwise specified, $Time$ is considered to be the result latency and $Area$, the number of logic elements. The TS multipliers were evaluated using the Gaussian and Uniform sets, as they are important sets in machine learning applications, as well as two neural network weight sets.

All sets were generated in single precision floating-point and converted to fixed-point numbers. The integer length was determined by taking the maximum value of the set and allocating sufficient bits to represent it fully, hence saturation did not need to be performed. The number of fractional bits are the remaining bits after the integer portion has been accounted for. The Gaussian set was generated with a mean of zero

TABLE 5.2: Multiplier Implementation Results

B	Type	*Area* (LEs)	Max Delay (ns)	Latency (Cycles)	*Power* (mW)
64	Parallel(Combinatorial)	5104	14.7	1	2.23
	Parallel(Pipelined)	4695	6.99	4**	9.62
	Booth Serial-Parallel	292	3.9	33	2.23
	Two Speed	304	1.83 (τ)	45.2*	5.2
32	Parallel(Combinatorial)	1255	10.2	1	1.33
	Parallel(Pipelined)	1232	4.6	4**	5.07
	Booth Serial-Parallel	156	3.8	17	1.78
	Two Speed	159	1.76 (τ)	25.6*	3.18
16	Parallel(Combinatorial)	319	6.8	1	0.94
	Parallel(Pipelined)	368	3.2	4**	3.49
	Booth Serial-Parallel	81	2.72	9	1.67
	Two Speed	87	1.52 (τ)	14*	4.35

For Two Speed, the Max Delay represents the τ subcircuit and $\bar{K} = 2$, hence 2τ is the delay of the adder subcircuit.
* This is the average latency over all of the tested sets.
** While the latency of the pipelined multiplier is four, the throughput is one.

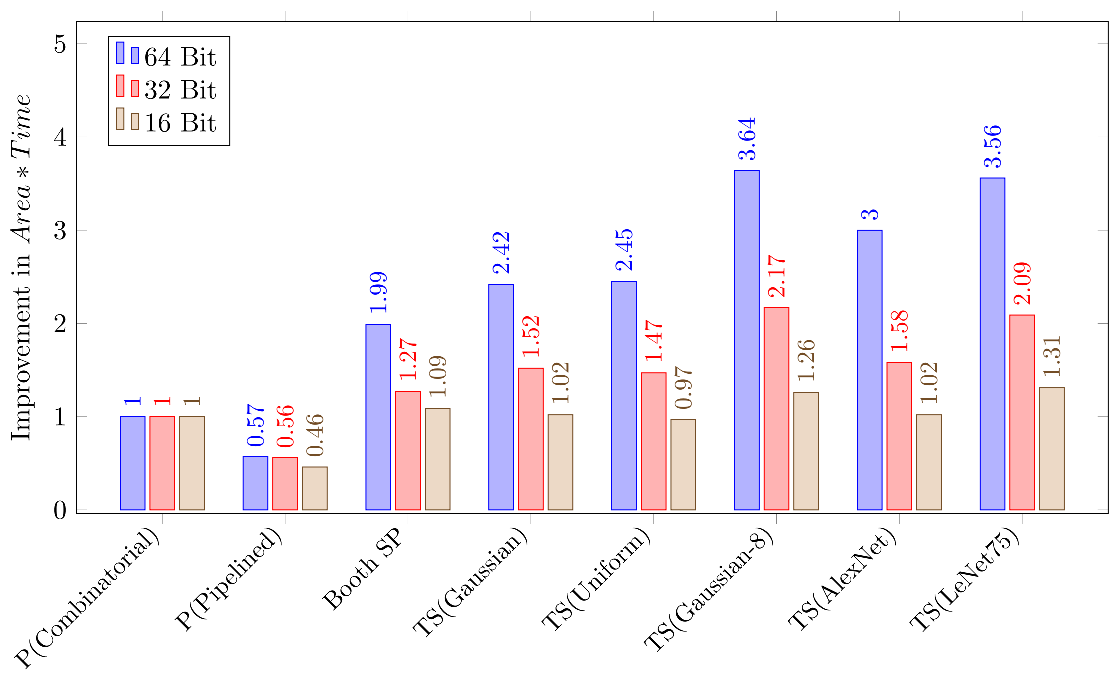


FIGURE 5.6: The improvement in $Area * Time$ for 4 different multiplier configurations respectively. Five different sets are presented for the TS multiplier.

and standard deviation of 0.1. For the Gaussian-8 set, the numbers were scaled such that they are represented in 8 bits. The uniform set was generated by selecting numbers between -1 and 1.

The neural network weight sets are from two convolutional neural networks, AlexNet [8] and a 75% spare variant of LeNet [40], LetNet75, trained using the methodology presented by Han. et. al [103]. The Parallel(Combinatorial)and Parallel(Pipelined) multipliers are taken from an optimised FPGA library provided by the vendor and are designed for high performance [104]. Since the performance of a Parallel (Pipelined) multiplier is a function of its pipeline depth, the reported values are the best results from numerous configurations to ensure a fair comparison. The Booth Serial-Parallel (SP) multiplier uses the radix-4 booth algorithm, illustrated in Algorithm 5.1 whereas the TS multiplier implements Algorithm 5.2.

Figure 5.6 presents the improvements in $Area * Time$ for the four different multipliers, with the Parallel(Combinatorial) illustrating baseline performance for each configuration. $Area*Time$ is an important metric for understanding architecture design attributes and the magnitude of possible tradeoffs between area and speed [105]. The fixed cycle times of the Booth Serial-Parallel, Parallel(Combinatorial) and Parallel(Pipelined) multipliers result in the same performance regardless of the input set. However, the TS multiplier is designed to take advantage of the input set and outperforms all other multipliers in the 32 bit and 64 bit configuration. In the 16 bit configuration, the TS multiplier exhibited similar performance to the baseline.

The highest performing set is the 64 bit Gaussian-8; showing a speed up of 3.64x. For the Gaussian and Uniform sets, the 64 bit configuration provides a 2.42x and 2.45x improvement respectively. At 32 and 16 bits, the TS multiplier's improvements range from 1.47-1.52x and 0.97-1.02x respectively. The Gaussian-8 set illustrates that inefficiencies introduced by using a lower bit representation are alleviated by the TS multiplier; the majority of the most-significant bits are either all 0's in the positive case, or all 1's in the negative case, allowing multiple consecutive "skips".

5.5.2 Set Analysis

Figure 5.7 shows the probability distributions of the five problems tested at 32-bits. It illustrates the differences between the Gaussian, Uniform, AlexNet, Gaussian-8 and LeNet75 sets and why particular sets perform better than others. For Gaussian-8, the majority of the encoding is in the $2-4$ range, resulting in a significant number of "skips" for each input. While the non-zero numbers in the LeNet75 set contain high encoding numbers, the set also contains 71% zeros, therefore the majority of the computations are "skips".

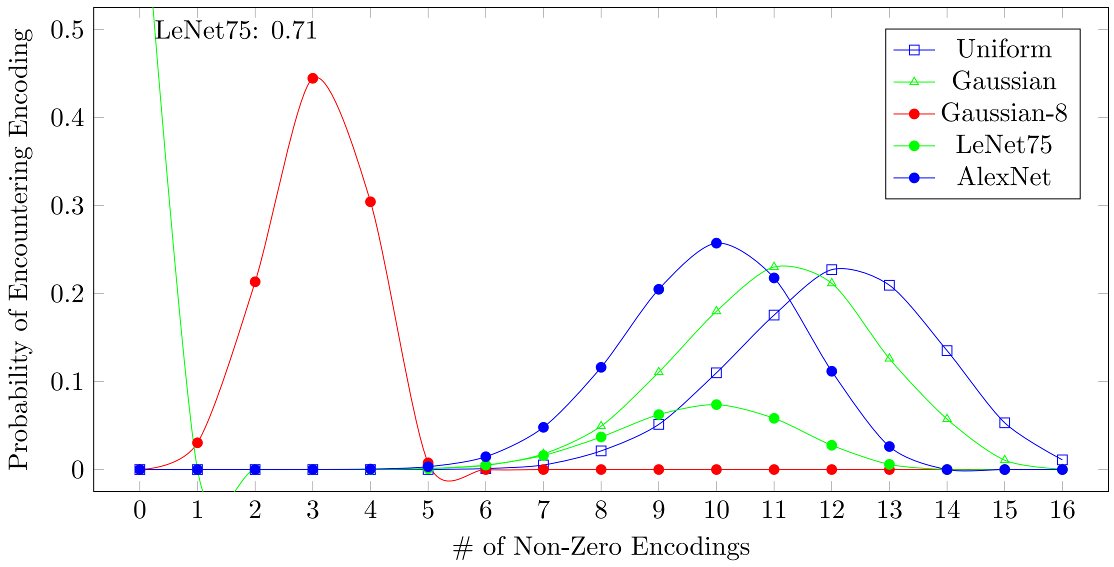


FIGURE 5.7: $p(i)$ 32 bit distribution: The probability that y will be a particular encoding.

5.5.3 Multiplier Rankings

Table 5.3 highlights the strengths and weaknesses of each multiplier in terms of six important factors. $Area$, $Time$, $Power$, $Area*Time$, $Time*Power$ and $Area*Time*Power$, often dictate which multiplier is most appropriate for the current design. Typically, tradeoffs are analysed and the variant with the highest performance is chosen. When designed for a small area, either the Booth Serial-Parallel or TS multiplier are the best choices as they have the smallest area footprint. Alternately, when both area and speed are factors, the TS multiplier outperforms the Booth Serial-Parallel multiplier as illustrated in Table 5.3 and Figure 5.6. If area isn't a concern, the Parallel(Combinatorial) multiplier is the best choice. When taking power into account, the Parallel(Combinatorial) multiplier outperforms the Parallel(Pipelined) multiplier.

As highlighted in Table 5.3, in terms of $Area * Time * Power$, the Booth Serial-Parallel offers the highest performance and is 1.9x better than the Parallel(Combinatorial) multiplier for a bit width of 64. However the TS multiplier still provides a sizeable improvement, achieving a 1.29x improvement on average, peaking at 1.5x for LeNet75 and Gaussian-8.

While the analysis focuses on latency, the Parallel(Pipelined) multiplier is generally throughput orientated. Calculating the $Area * Time$ with respect to throughput, shows that the Parallel(Pipelined) multiplier achieves a 1.84-2.29x performance improvement over the Parallel(Combinatorial) multiplier for bit widths 16, 32 and 64. The TS multiplier still shows favourable results for both the Uniform and Gaussian sets, while outperforming on the Gaussian-8 and neural network sets.

TABLE 5.3: Multiplier Performance Metrics - Latency

B	Type	$Area$	$Time$	$Power$	$Area * Time$	$Time * Power$	$Area * Time * Power$
64	Parallel(Combinatorial)	5104	**14.7**	2.23	75028	**32.78**	167314
	Parallel(Pipelined)	4695	27.96	9.62	131274	268.98	315716
	Booth Serial-Parallel	**292**	128,7	2.23	37696	287.00	**84062**
	Two Speed*	**304**	82.71	5.2	**25187**	430.12	130972
32	Parallel(Combinatorial)	1255	**10.2**	1.33	12808	**13.57**	**17034**
	Parallel(Pipelined)	1232	18.4	5.07	22678	93.29	114977
	Booth Serial-Parallel	**156**	64.6	1.78	10116	114.99	18007
	Two Speed*	**159**	45.05	3.18	**7186**	143.27	22852
16	Parallel(Combinatorial)	319	**6.8**	0.94	2169	**6.39**	**2039**
	Parallel(Pipelined)	368	12.8	3.49	4714	44.67	16452
	Booth Serial-Parallel	**81**	24.48	1.67	1987	40.88	3319
	Two Speed*	**87**	21.28	4.35	**1851**	92.56	8053

* Average over all tested sets, the individual results will change for specific applications.

5.6 Summary

This chapter presented a two speed multiplication algorithm and implementation, which is divided into two subcircuits, each operating with a different critical path. In real-time, the performance of this multiplier can be improved solely on the distribution of the bit representation. It was illustrated that for bit widths of 32 and 64, important compute timess, such as Uniform and Gaussian, and neural networks can expect substantial improvements of 3-3.56x using standard learning and spare techniques.

The cost associated with handling lower bit width representations, such as Gaussian-8 on a 64 bit multiplier are alleviated and show up to a 3.64x improvement compared with the typical parallel multiplier.

Chapter 6

Conclusion

In this thesis, FPGA architectures for low precision machine learning workloads has been presented. An approach to optimising these workloads was illustrated at the application, framework and operator levels. It was demonstrated that at each of these levels, there exists distinct techniques for improving area, latency and throughput without sacrificing functionality.

In Chapter 3, two different radio-frequency anomaly detection techniques were explored; a bitmap detector and a neural network autoencoder. A new on-line algorithm for spectral bitmap anomaly detection was proposed. Implementations of this new algorithm provided one to two orders of magnitude improvements in speed, latency, power and energy over a single threaded implementation compiled from the same C source code. It was shown that reducing the neural networks precision representation improved the hardware utilisation and the implementations performance. This demonstrated the feasibility of a single-chip, 200 MHz sample-at-a-time NN-based anomaly detection, resulting in high throughput and ultra-low latency. This illustrates that inclusion of real-time neural networks in sophisticated software defined radio systems, with potential applications in fault diagnosis, spectrum enforcement and collaborative spectrum sharing can address severely constrained real-time anomaly detection problems using FPGA technology.

Chapter 4 presented a customisable matrix multiplication framework. The framework includes a software API and a hardware template for designing custom GEMM accelerators on the HARPv2. Utilising the runtime API, the framework allows for a wide range of different precisions, various core sizes and tuneable runtime configurable parameters. This flexibility allowed the framework to adapt to networks requirements and alleviated performance bottlenecks that effect other systems. It illustrated that in-package integrated FPGAs can remain competitive with a high-end discrete GPU on raw performance and performance per watt. A study on other optimisations, targeted at deep learning, and their impact on performance was presented. It was demonstrated that for binarised deep learning workloads, the FPGA was either on par or exceeded the GPUs performance and offered insight into memory bandwidth constraints and the impact of configurable memory interleaving. Finally, four heterogeneous implementations were explored and evaluated in regards to total execution time. It illustrated that by dedicating FPGA resources to accelerating specific bottlenecks and utilising the CPU for other operations; results in higher performance implementations equipped to take advantage of close collaboration between the FPGA and CPU.

Chapter 5 presented a two speed multiplication algorithm. Divided into two subcircuits; each operating with a different critical path, the performance of this multiplier is improved solely on the distribution of the bit representation. The multiplier performs its optimisation in real-time, taking advantage of sparsity in the input distribution and individual multiplier inputs. Improvements up to 3x and 3.56x over traditional multipliers were demonstrated for the 32 bit and 64 bit compute distributions respectively. The 3.64x performance improvement offered by the 64 bit TS multiplier for the Gaussian-8 distribution, motivates a single mixed precision datapath implementation; providing the flexibility to compute any bit width without reconfiguration or recompilation.

6.1 Future Work

6.1.1 Anomaly Detection

The neural network detector showed significant promise in the area of FM anomaly detection. Future work could include a detailed study on other types of anomalies found in different radio signals such as WiFi and QPSK type signals. While the neural network implementation achieves high throughput and low latency, the network size is significantly smaller than the current state-of-the-art networks in other fields. Additional work could examine trade-offs between more complex and accurate networks, operating at a higher latency.

A standard multi-layered perception autoencoder was used to reconstruct the input window without the anomalies present. This was useful since an *unsupervised learning* technique allowed anomalies to be identified without labelled training data. However, this restricted the autoencoder to reconstruct only previously observed data. While the off-chip training and weight update technique aimed to alleviate this restriction via an on-line learning approach; future work could examine other network topologies that learn to identify the anomalies impact on the signal, rather than learning the underlying signal itself.

Finally, the current on-line learning approach uses an off-line processor to update the weights and bias, requiring significantly higher latency than inference. Implementing training on the FPGA was limited by the available hardware resources, however by reducing the autoencoder's latency and thoughput requirements, combined inference and training could be possible on the FPGA. Future work in this area could be directed at implementing an on-line learning algorithm on the FPGA. While backpropagation requires significant computational resources, other methods such as node or weight perturbation provide a more FPGA friendly method of computing the gradients.

6.1.2 GEMM Framework

Chapter 4 considered deep learning as a case study; while this is an important emerging technology, other more established problems could also benefit for GEMM acceleration on FPGAs. GEMM is used in a wide range of problems, and future work could include more detailed analysis of high performance computing and scientific computing applications, including other machine learning algorithms. The framework supported fused operations such as ReLU and Batch Norm, however these were only available during the forward pass of the computation. During backpropagation, only the GEMM portion of the framework can be used, as the fused operator do not support their derivative computation. Future work could extend these fused operators to support backpropagation, enabling more efficient training.

Two methods for performing mixed precision computation on the framework was offered, however in the dynamic dot product implementation, routing resources limited the binary computation. The other method involved dynamically reconfiguring the reprogrammable region with different precisions as needed. Unfortunately, the latency involved in reconfiguring the device made this approach intractable to real-time implementation. As future work, an alternative implementation could separate the systolic array into multiple smaller arrays, each dedicated to a particular precision. A mixed precision implementation that effectively utilises the FPGA resource could be achieved, tailoring the array sizes according to the amount of work required by the particular precision. Furthermore, scaling the implementation to larger FPGAs, such as the Intel Stratix 10, provides additional compute resources, however the cost of increased array size and memory bandwidth requirements needs further study and analysis.

6.1.3 Two Speed Multiplier

The two speed multiplication algorithm takes advantage of its inputs bit representation to speed up its computation. This thesis focused on typical input distributions such as the Gaussian and Uniform distributions; including already established input distributions for AlexNet and LeNet. Future work in this area could examine new neural network learning methodologies, aimed at taking advantage of the two speed multiplication algorithm. One possible method involves a new cost function that takes into account the accuracy of the machine learning algorithm, as well as the hardware cost in terms of area-time.

During the analysis, the layers in the neural networks were constrained to operate at the precision of the multiplier. However, as shown by the Gaussian-8 example, the two speed multiplier is able to provided significant performance improvements even when the majority of the most-significant bits are not needed. Future work could examine the performance improvements over traditional multipliers when the different layers operate at different precisions.

Finally, the neural network analysis was limited to the theoretical case where the multiplier operates at maximum efficiency. One side effect of the multiplier is the non-uniform computation latency requiring additional circuitry, such as FIFO and other buffers, to ensure the multiplier operates at peak efficiency. Future work could implement the multiplier in a larger system and evaluate its performance, comparing the theoretical results presented in this thesis.

6.2 Closing Remarks

The aim of this thesis has been to demonstrate precision optimisations in FPGA implementations of machine learning. The thesis opens with the application of reduced precision anomaly detection at reduced precision. Two anomaly detectors were evaluated; the first, using a neural network and the second, a bitmap approach. The neural network detector demonstrated that a reduced precision implementation enables larger and more complex models, resulting in better detection. With the benefits of low precision neural networks established, a generalised framework was presented; aimed specifically at the most compute intensive parts of a neural network. It was evaluated against three state-of-the-art binarised networks on ImageNet, showing comparable raw performance and higher energy efficiency when compared with GPUs. With multiplication as the fundamental building block of this framework, the two speed multiplication algorithm was presented aimed to improve computation time solely based on the bit distribution within the weights. Under typical use, the TS multiplier is capable of achieving substantial improvements in $Area * Time$. These optimisations combine to tell a compelling story. Realising high performance implementations of machine learning workloads, requires careful examination at three distinct levels; the application, the framework and the operator.

Appendix A

Convolution Layer Computations

A.1 Introduction

To reduce the computational complexity, or improve memory utilisation, three methods for computing a convolution have been studied as alternatives to the standard convolution method presented in Section 2.5.6. These methods are:

- im2col [106, 107]: This does not directly effect the number of computations; the layout of the computation is changed to improve memory utilisation.
- Winograd [108–110]: This is a minimal filtering algorithm $F(m\times m, r\times r)$, reducing the number of computations needed to perform the convolution.
- FFT [111]: Circular convolutions in the spatial domain are equivalent to pointwise products in the Fourier domain, effectively reducing the number of computations that are performed.

A.2 imc2col

Unlike the Winograd and FFT approaches, the GEMM approach does not reduce the overall computational complexity; it aims to reformat the computation to behave like a GEMM. One of main issues in efficient direct convolution algorithms is idle processor time due to cache misses. By transforming the computation to resemble a GEMM, techniques such as blocking and prefetching are used to alleviate any memory bottlenecks [39]. The transform, called Image-to-Column (im2col) is performed on the input and output images before and after the GEMM. A simplified *im2col* algorithm is illustrated in Algorithm A.1, where $I_H = I_W = N_I$ and $O_H = O_W = N_O$.

Algorithm A.1: im2col Algorithm.

```
Data: $I$: 3D Input Image
Result: $O$: 2D Output Column Format
1 for $i = 1$ to $I_D \times F \times F \times N_O^2$ do
      // Calculate Output indices
2     $p = F/N_O^2$;
3     $q = F \mod N_O^2$;
      // Calculate Input indices
4     $d = (p/F)/F$;
5     $i = q/N_O + (p/F) \mod F$;
6     $j = q \mod N_O + p \mod F$;
      // Input to Output Assignment
7     $O_{p,q} = I_{d,i,j}$;
```

Note that in this case, N_O is calculated using:

$$N_O = N_I - F + 1 \tag{A.1}$$

The im2col algorithm creates the A matrix $(F \times F \times I_D, O_W \times O_H)$ and is split into three parts; first the output indices are calculated, followed by the input indices and finally the assignment step. The B matrix is created by transforming the four dimensional filter array (F, F, I_D, O_D), into a two dimensional matrix $(O_D, F \times F \times I_D)$. This is done by stretching the (F, F, I_D) filters into single dimensional $(F \times F \times I_D)$ vectors. These vectors are then stacked, creating the $(O_D, F \times F \times I_D)$ output matrix B. C is then computed via a GEMM to produce the $(O_W \times O_H, O_D)$ output matrix. Multiple images can be batched, resulting in a $(F \times F \times I_D, O_W \times O_H \times B)$ A matrix.

A.3 Winograd

Winograd [110] illustrated that a 1D minimal filtering algorithm $F(m, r)$ for some selected values of m and r is:

$$Y = A^T[(Gg) \odot (B^T d)] \tag{A.2}$$

where B^T, G and A^T are constant transform matrices calculated for a particular m and r. g and d are the filter and input sections respectively, $\odot$ denotes the Hadamard product, an element-wise matrix multiplication, and Y the output section. By a nested application of the 1D case, the 2D case, $F(m \times m, r \times r)$ is expressed as:

$$Y = A^T[(GgG^T) \odot (B^T dB)]A \tag{A.3}$$

The Winograd algorithm for 2D convolution is split into three section. First, the input image and filter transformations are performed, $V = (B^T dB)$ and $U = (GgG^T)$ respectively. Second, the element-wise matrix multiplication $M = U \odot V$. Finally, the output transform $Y = A^T MA$. Further analysis of the algorithm is outside the scope of this thesis and the reader is referred to Winograd [110], or Blahut [109] for a modern interpenetration of the algorithm. Using the Winograd algorithm, the computation complexity of the batched convolution is calculated using:

$$\alpha'(1 + \beta'/O_D + \gamma'/P + \delta'/I_D) \cdot B \cdot N_I^2 \cdot I_D \cdot O_D \tag{A.4}$$

where $N_I = I_W = I_H$, $P = B \times \lceil N_I/m \rceil \times \lceil N_I/m \rceil$, β', γ' and δ' are the number of computations performed during the input, filter and output transformation steps, $\alpha' = \frac{(m+F-1)^2}{m^2}$. Achieving a large speed up requires that β', γ' and δ' are relatively small compared to their respective O_s, P and I_s. Compared to the direct convolution (Equation 2.60), in the best case the maximum speed up is r^2/α'.

A.4 FFT

The Convolution Theorem states that circular convolutions in the spatial domain are equivalent to pointwise products in the Fourier domain [112]. Letting $\mathcal{F}$ denote the Fourier transform and $\mathcal{F}^{-1}$ its inverse, the convolution between two functions f and g is calculated as:

$$f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g)) \tag{A.5}$$

It follows from Equation A.5, that the convolution of the filter array f, and a batch of input images g can be calculated by applying the Fourier transform to both f and g, performing a matrix multiplication, then finishing with an inverse Fourier transform. While an in-depth treatment of this approach is beyond the scope of this thesis, a modern interpretation can be found in Mathieu et. al [111]. The computational complexity for the batched case can be generalised to:

$$2Cn^2 \log n[B \cdot O_D + I_D \cdot B + I_D \cdot O_D] + 4B \cdot I_D \cdot O_D \cdot N_I^2 \tag{A.6}$$

where C represents the hidden constant in the FFT complexity. Comparing Equation 2.60 and Equation A.6, the FFT computation only consists of a product of at most four terms, whereas the direct convolution is a product of five terms.

Bibliography

[1] Eamonn Keogh, Xiaopeng Xi, Li Wei, and Chotirat Ann Ratanamahatana. The ucr time series classification/clustering homepage, 2011. URL http://www.cs.ucr.edu/~eamonn/time_series_data.

[2] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. A Symbolic Representation of Time Series, with Implications for Streaming Algorithms. In *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 2–11, 2003. doi: 10.1145/882082.882086.

[3] The Econmist. Why artificial intelligence is enjoying a renaissance. *The Economist*, 2016. URL https://www.economist.com/blogs/economist-explains/2016/07/economist-explains-11.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016. doi: 10.1109/CVPR.2016.90.

[5] Dana Boyd and Kate Crawford. Six provocations for big data. In *Social Science Research Network: A Decade in Internet Time: Symposium on the Dynamics of the Internet and Society*, 2011.

[6] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 2011.

[7] Don Clark. Computer chips evolve to keep up with deep learning. *The Wall Street Journal*, 2017. URL https://www.wsj.com/articles/computer-chips-evolve-to-keep-up-with-deep-learning-1484161286.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[9] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4892-8. doi: 10.1145/3079856.3080246.

[10] Karl Freund. Amazon and xilinx deliver new fpga solutions. *Forbes*, 2017. URL https://www.forbes.com/sites/moorinsights/2017/09/27/amazon-and-xilinx-deliver-new-fpga-solutions/#23cd8def2370.

[11] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84, Dec 2016. doi: 10.1109/FPT.2016.7929192.

[12] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 5–14, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021740.

[13] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pages 3123–3131, Cambridge, MA, USA, 2015. MIT Press.

[14] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[15] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. volume abs/1603.05279, 2016.

[16] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 15–24, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021741.

[17] 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, pages 65–74, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021744.

[18] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[19] Duncan J.M. Moss, David Boland, Peyam Pourbeik, and Philip H.W. Leong. Real-time FPGA-based anomaly detection for radio frequency signals. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018. under review.

[20] Duncan J.M. Moss, Zhe Zhang, Nicholas J. Fraser, and Philip H.W. Leong. An FPGA-based spectral anomaly detection system (with errata). In *Proc. International Conference on Field Programmable Technology (FPT)*, pages 175–182, 2014. doi: 10.1109/FPT.2014.7082772.

[21] Duncan J.M. Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. A customizable matrix multiplication framework for the intel HARPv2 platform. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018. to appear.

[22] Duncan J.M. Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. High performance binary neural networks on the xeon+fpga platform. In *2017 27th International Conference on*

Field Programmable Logic and Applications (FPL), pages 1–4, Sept 2017. doi: 10.23919/FPL.2017.8056823.

[23] Duncan J.M. Moss, David Boland, and Philip H.W. Leong. A two speed serial-parallel multiplier. In *IEEE Transactions on VLSI Systems*, 2018. under review.

[24] Andrew D. Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[25] Steve Kilts. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007. ISBN 0470054379.

[26] Stephen Chappell. Implementing floating-point dsp on fpgas. *EETimes*, 2007. URL https://www.eetimes.com/document.asp?doc_id=1275422.

[27] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, September 2013. ISSN 1539-9087. doi: 10.1145/2514740.

[28] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan. Xilinx vivado high level synthesis: Case studies. In *25th IET Irish Signals Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014)*, pages 352–356, June 2014. doi: 10.1049/cp.2014.0713.

[29] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 9780080556017, 9780123705228.

[30] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Strathclyde Academic Media, UK, 2014. ISBN 099297870X, 9780992978709.

[31] David Lewis, David Cashman, Mark Chan, Jeffery Chromczak, Gary Lai, Andy Lee, Tim Vanderhoek, and Haiming Yu. Architectural enhancements in stratix v™. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 147–156, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1887-7. doi: 10.1145/2435264.2435292.

[32] Ray Bittner. Bus mastering pci express in an fpga. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*,

FPGA '09, pages 273–276, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-410-2. doi: 10.1145/1508128.1508176.

[33] William Wong. Understanding fpga processor interconnects. *ElectronicDesign*, 2012. URL http://www.electronicdesign.com/fpgas/understanding-fpga-processor-interconnects.

[34] M.D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann Series in Comp. Morgan Kaufmann, 2004.

[35] Vivienne Sze, Yu-Hsin Chen, Joel S. Emer, Amr Suleiman, and Zhengdong Zhang. Hardware for machine learning: Challenges and opportunities. *CoRR*, abs/1612.07625, 2016.

[36] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991. ISSN 0360-0300. doi: 10.1145/103162.103163.

[37] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

[38] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[39] . An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002. ISSN 0098-3500. doi: 10.1145/567806.567807.

[40] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11):2278–2324, 1998.

[41] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009. ISSN 0360-0300. doi: 10.1145/1541880.1541882.

[42] N. B. Truong, Y. J. Suh, and C. Yu. Latency analysis in gnu radio/usrp-based software radio platforms. In *MILCOM 2013 - 2013 IEEE Military Communications Conference*, pages 305–310, Nov 2013. doi: 10.1109/MILCOM.2013.60.

[43] Victoria Hodge and Jim Austin. A survey of outlier detection methodologies. *Artif. Intell. Rev.*, 22(2):85–126, October 2004. ISSN 0269-2821. doi: 10.1023/B:AIRE.0000045502.10941.a9.

[44] Malik Agyemang, Ken Barker, and Rada Alhajj. A comprehensive survey of numeric and symbolic outlier mining techniques. *Intell. Data Anal.*, 10(6):521–538, December 2006. ISSN 1088-467X.

[45] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Comput. Netw.*, 51(12): 3448–3470, August 2007. ISSN 1389-1286. doi: 10.1016/j.comnet.2007.02.001.

[46] Rekha Hibare and Anup Vibhute. Feature extraction techniques in speech processing: A survey. *International Journal of Computer Applications*, 107(5), 2014.

[47] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965. ISSN 0025–5718. URL: `http://cr.yp.to/bib/entries.html#1965/cooley`.

[48] Timothy J. O'Shea, T. Charles Clancy, and Robert W. McGwier. Recurrent neural radio anomaly detection. *CoRR*, abs/1611.00301, 2016.

[49] A Das, D. Nguyen, J. Zambreno, G. Memik, and A Choudhary. An FPGA-based network intrusion detection architecture. *Information Forensics and Security, IEEE Transactions on*, 3(1):118–132, March 2008. ISSN 1556-6013. doi: 10.1109/TIFS.2007.916288.

[50] Kevin M Carter and William W Streilein. Probabilistic reasoning for streaming anomaly detection. In *Statistical Signal Processing Workshop (SSP), 2012 IEEE*, pages 377–380. IEEE, 2012.

[51] Paul Hayton, Simukai Utete, Dennis King, Steve King, Paul Anuzis, and Lionel Tarassenko. Static and dynamic novelty detection methods for jet engine health monitoring. *Philos Trans A Math Phys Eng Sci*, 365(1851):493–514, 2007. ISSN 1364-503X.

[52] Jeffrey D Scargle. Studies in astronomical time series analysis. ii-statistical aspects of spectral analysis of unevenly spaced data. *The Astrophysical Journal*, 263:835–853, 1982.

[53] Syed A. Pasha and Philip H.W. Leong. Cluster analysis of high-dimensional high-frequency financial time series. In *IEEE Symposium on Computational Intelligence for Financial Engineering & Economics - (CIFEr)*, pages 68–75, 2013.

[54] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-43108-5.

[55] Nitin Kumar, VN Lolla, and Eamonn Keogh. Time-series Bitmaps: a Practical Visualization Tool for Working with Large Time Series Databases. In *SIAM International Conference on Data Mining*, pages 531–535, 2005. doi: http://dx.

doi.org/10.1137/1.9781611972757.55BookCode:PR119Series:ProceedingsPages: 5ReadMore:http://epubs.siam.org/doi/abs/10.1137/1.9781611972757.55.

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

[57] Freescale. Complex fixed-point fast fourier transform optimization for altivec. Applications Note AN2114, 2015. URL https://www.nxp.com/docs/en/application-note/AN2114.pdf.

[58] Xilinx Inc. Vivado design suit user guide, high-level synthesis. UG902 (v2015.4), 2016. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug902-vivado-high-level-synthesis.pdf.

[59] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[60] PK Gupta. Accelerating datacenter workloads. In *FPL*, 2016.

[61] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[62] Chi Zhang and Viktor Prasanna. Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 35–44, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021727.

[63] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C. Hoe. A study of pointer-chasing performance on shared-memory processor-fpga systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 264–273, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3856-1. doi: 10.1145/2847263.2847269.

[64] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A quantitative analysis on microarchitectures of modern cpu-fpga platforms. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 109:1–109:6, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4236-0. doi: 10.1145/2897937.2897972.

[65] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 403–415, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4197-4. doi: 10.1145/3035918.3035954.

[66] Xuechao Wei, Yun Liang, Tao Wang, Songwu Lu, and Jason Cong. Throughput optimization for streaming applications on cpu-fpga heterogeneous systems. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 488–493, Jan 2017. doi: 10.1109/ASPDAC.2017.7858370.

[67] NVIDIA Corporation. Nvidia tesla v100 gpu architecture. Technical Report WP-08608-001, NVIDIA Corporation, 2017. URL https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf.

[68] Naveen Mellempudi, Abhisek Kundu, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. Ternary neural networks with fine-grained quantization. *arXiv preprint arXiv:1705.01462*, 2017.

[69] Julian Faraone, Nicholas Fraser, Giulio Gambardella, Michaela Blott, and Philip HW Leong. Compressing low precision deep neural networks using sparsity-induced regularization in ternary networks. In *International Conference on Neural Information Processing*, pages 393–404. Springer, 2017.

[70] Mitsuru Ambai, Takuya Matsumoto, Takayoshi Yamashita, and Hironobu Fujiyoshi. Ternary weight decomposition and binary activation encoding for fast and compact neural network. 2016.

[71] Yoonho Boo and Wonyong Sung. Structured sparse ternary weight coding of deep neural networks for efficient hardware implementations. *arXiv preprint arXiv:1707.03684*, 2017.

[72] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[73] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proc. 44th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2017.

[74] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[75] Song Han, Yu Wang, Shuang Liang, Song Yao, Hong Luo, Yi Shan, and Jinzhang Peng. Reconfigurable processor for deep learning in autonomous vehicles. 2017.

[76] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proc. 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM, 2016.

[77] Chaim Baskin, Natan Liss, Avi Mendelson, and Evgenii Zheltonozhskii. Streaming architecture for large-scale quantized neural networks on an FPGA-based dataflow platform. *arXiv preprint arXiv:1708.00052*, 2017.

[78] Paolo Meloni, Alessandro Capotondi, Gianfranco Deriu, Michele Brian, Francesco Conti, Davide Rossi, Luigi Raffo, and Luca Benini. Neuraghe: Exploiting CPU-FPGA synergies for efficient and flexible cnn inference acceleration on zynq socs. *arXiv preprint arXiv:1712.00994*, 2017.

[79] Chi Zhang and Viktor Prasanna. Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system. In *Proc. 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 35–44. ACM, 2017.

[80] Duncan JM Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW Leong. High performance binary neural networks on the Xeon+ FPGA™ platform. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–4. IEEE, 2017.

[81] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. A 7.663-TOPS 8.2-W energy-efficient FPGA accelerator for binary convolutional neural networks. *arXiv preprint arXiv:1702.06392*, 2017.

[82] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. Accelerating deep convolutional networks using low-precision and sparsity. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*, pages 2861–2865. IEEE, 2017.

[83] 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 *Proc. 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74. ACM, 2017.

[84] Nicholas J Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Scaling binarized neural networks on reconfigurable logic. In *Proc. 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pages 25–30. ACM, 2017.

[85] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. FP-BNN: Binarized neural network on FPGA. *Neurocomputing*, 2017.

[86] Jin Hee Kim, Brett Grady, Ruolong Lian, John Brothers, and Jason H Anderson. FPGA-based CNN inference accelerator synthesized from multi-threaded C software.

[87] Adrien Prost-Boucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, and Vincent Leroy. Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–7. IEEE, 2017.

[88] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

[89] 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 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3315-3. doi: 10.1145/2684746.2689060.

[90] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[91] Eriko Nurvitadhi, Asit Mishra, Yu Wang, Ganesh Venkatesh, and Debbie Marr. Hardware accelerator for analytics of sparse data. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pages 1616–1621. EDA Consortium, 2016.

[92] Asit K. Mishra, Eriko Nurvitadhi, Jeffrey J. Cook, and Debbie Marr. WRPN: training and inference using wide reduced-precision networks. *CoRR*, abs/1709.01134, 2017.

[93] Taiwan Semiconductor Manufacturing Company. Tsmc 16/12nm technology, 2013. URL http://www.tsmc.com/english/dedicatedFoundry/technology/16nm.htm.

[94] Nicholas J. Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Scaling binarized neural networks on reconfigurable logic. In *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '17, pages 25–30, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4877-5. doi: 10.1145/3029580.3029586.

[95] Jyoti Kalia and Vikas Mittal. A review of different methods for Booth multiplier. 07:60–63, 06 2017.

[96] B. Dinesh, V. Venkateshwaran, P. Kavinmalar, and M. Kathirvelu. Comparison of regular and tree based multiplier architectures with modified booth encoding for 4 bits on layout level using 45nm technology. In *2014 Intl. Conf. on Green Computing Communication and Electrical Engineering*, pages 1–6, 2014.

[97] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pages 1–12, 2016.

[98] G. C. T. Chow, W. Luk, and P. H. W. Leong. A mixed precision methodology for mathematical optimisation. In *2012 IEEE 20th Intl. Symp. on Field-Programmable Custom Computing Machines*, pages 33–36, 2012.

[99] Gary Chun Tak Chow, Anson Hong Tak Tse, Qiwei Jin, Wayne Luk, Philip H.W. Leong, and David B. Thomas. A mixed precision monte carlo methodology for reconfigurable accelerator systems. In *Proc. of the ACM/SIGDA Intl. Symp. on Field Programmable Gate Arrays*, pages 57–66, 2012.

[100] K. S. Trivedi and M. D. Ercegovac. On-line algorithms for division and multiplication. *IEEE Trans. on Computers*, 26(7):681–687, 1977.

[101] K. Shi, D. Boland, and G. A. Constantinides. Efficient FPGA implementation of digit parallel online arithmetic operators. In *2014 Intl. Conf. on Field-Programmable Technology*, pages 115–122, 2014.

[102] Yiren Zhao, J. Wickerson, and G. A. Constantinides. An efficient implementation of online arithmetic. In *2016 Intl. Conf. on Field-Programmable Technology*, pages 69–76, 2016.

[103] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Proc. of the 28th Intl. Conf. on Neural Information Processing Systems*, pages 1135–1143, 2015.

[104] Intel Corporation. Integer arithmetic ip cores user guide. Technical Report UG-01063, Intel Corporation, 2017. URL https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_lpm_alt_mfug.pdf?GSA_pos=1.

[105] Ian Kuon and Jonathan Rose. Area and delay trade-offs in the circuit and architecture design of FPGAs. In *Proc. of the 16th Intl. ACM/SIGDA Symp. on Field Programmable Gate Arrays*, pages 149–158, 2008.

[106] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), October 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

[107] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. Low-memory gemm-based convolution algorithms for deep neural networks. *CoRR*, abs/1709.03395, 2017.

[108] Andrew Lavin. Fast algorithms for convolutional neural networks. *CoRR*, abs/1509.09308, 2015.

[109] Richard E. Blahut. *Fast Algorithms for Digital Signal Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1985. ISBN 0201101556.

[110] S. Winograd. *Arithmetic Complexity of Computations*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1980. ISBN 9780898711639.

[111] Michaël Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *CoRR*, abs/1312.5851, 2013.

[112] G.B. Arfken and H.J. Weber. *Mathematical Methods for Physicists*. Academic Press international edition. Harcourt/Academic Press, 2001. ISBN 9780120598250.