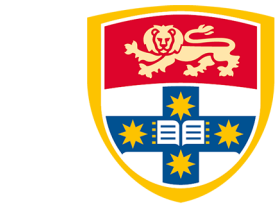
Specialized Architectures and Arithmetic for Machine Learning



THE UNIVERSITY OF SYDNEY

Mr. Sean Fox
School of Electrical and Information Engineering
Faculty of Engineering
The University of Sydney

October 2021

This thesis is presented as part of the requirements for the conferral of the degree:

Doctor of Philosophy (PhD)

Declaration

I, *Mr. Sean Fox*, declare that this thesis is submitted to fulfil the requirements for the conferral of the degree *Doctor of Philosophy (PhD)*, from the University of Sydney, is wholly my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualifications at any other academic institution.

Mr. Sean Fox

October 20, 2021

Abstract

Machine learning has risen to prominence in recent years thanks to advancements in computer technology, the abundance of data, and numerous breakthroughs in a broad range of applications. Unfortunately, as the demand for machine learning has grown, so too has the amount of computation required for training. Combine this trend with declines observed in performance scaling of standard computer architectures, and it has become increasingly difficult to support machine learning training at increased speed and scale, especially in embedded devices which are smaller and have stricter constraints.

Research points towards the development of purpose-built hardware accelerators to overcome the computing challenge, and this thesis explains how specialized hardware architectures and specialized computer arithmetic can achieve performance not possible with standard technology, e.g. Graphics Processing Units (GPUs) and floating-point arithmetic. Based on software and hardware experiments, for kernel methods and deep neural network (DNN) algorithms, this thesis shows that specialized arithmetic is crucial for accurately training large models with less memory, while specialized architectures are needed to increase computational parallelism and reduce off-chip memory transfers.

A systolic Field Programmable Gate Array (FPGA) implementation of the Fastfood algorithm is described which operates at a high frequency and supports simultaneous training and prediction for large-scale kernel methods. The architecture is specialized to minimize memory overheads and efficiently implement the Fast Walsh Hadamard Transform (FWHT) bottleneck. Empirical results show that 500 MHz clock rates can be sustained for an architecture that can solve problems with input dimensions up to 10^3 times larger than previously reported. This enables the use of kernel methods in applications requiring a rare combination of capacity, adaption, and speed.

Next, low-precision DNN training is investigated as an alternative to standard full-precision algorithms on FPGAs. A prototype training accelerator is described for Zynq All Programmable System on Chip (APSoC) devices using predominantly 8-bit integer numbers. Block floating-point (BFP) quantization and stochastic weight averaging techniques are applied during training to avoid any degradation in accuracy. Results of an implementation reveal memory savings and 17× speed-ups over processor-only systems on several

training tasks including image classification benchmarks, and online radio-frequency (RF) anomaly detection. Moreover, a 0.5pp accuracy improvement is achieved over previous work with results within 0.1pp of floating-point.

Lastly, a new representation called Block Minifloat (BM) is introduced for training DNNs end-to-end with only 4-8 bit tensors. While standard floating-point representations have two degrees of freedom, via the exponent and mantissa, BM exposes the exponent bias as an additional field for optimization. Crucially, this enables training with fewer exponent bits, yielding dense integer-like hardware for fused multiply-add (FMA) operations. For the task of image classification, 6-bit BM achieves almost no degradation in floating-point accuracy with FMA units that are $4.1\times$ ($23.9\times$) smaller and consume $2.3\times$ ($16.1\times$) less energy than FP8 (FP32). Furthermore, over a wide variety of learning tasks, an 8-bit BM format matches floating-point accuracy while delivering a higher computational density and faster expected training times.

In summary, this thesis shows how hardware specialization can lead to high-accuracy and large-scale machine learning training in hardware constrained devices, and enable the application of on-chip training in FPGAs. These outcomes are an important step towards moving more machine intelligence into e.g. mobile phones, video cameras, radios, and satellites.

Acknowledgements

First and foremost, I would like to thank my supervisor Philip Leong. His care and encouragement have been the guiding force behind the progression of this thesis. Working with Philip has been an immensely rewarding experience and I feel truly fortunate to have been his student.

This thesis would not have been possible nor as enjoyable without the helpful discussions and good times shared with friends and colleagues from the Computer Engineering Lab (CEL). Julian Faraone, Stephen Tridgell, Seyedramin Rasoulinezhad, Duncan Moss, and Siddhartha. Credit to each of you for being worthy card-playing opponents. Special thanks must also go to my co-supervisor David Boland for always being helpful and providing quality advice over the years. I'd also like to thank Graham Schelle, Yun Qu, and Patrick Lysaght for my internship at Xilinx, Barry Flower for helping to review this thesis, and Consunet for supporting me during my final year.

Lastly, I thank my parents, sister and brother. They never had much clue what I was talking about, but they pretended to listen anyways and have been tremendously supportive during my PhD, and frankly, throughout my entire life.

Authorship Attribution Statement

This thesis contains several chapters with material that was previously published during my PhD candidature under the supervision of Professor Philip Leong. The ideas and preparation behind each publication is primarily my own work, with all assistance that I received stated and acknowledged below:

- Chapter 3. Aspects of the systolic array architecture were assisted by discussions with David Boland and Stephen Tridgell.
- Chapter 4. The idea to implement SWALP on an FPGA came from Philip Leong and Xilinx Research Labs.
- Chapter 5. The hardware synthesis of Block Minifloat arithmetic units was conducted by Seyedramin Rasoulinezhad.
- The writing and presentation of each publication were improved through discussion and feedback from all co-authors.

In addition to the statements above, in cases where I am not the corresponding author of a published item, permission to include the published material has been granted by the corresponding author.

Sean Fox,
October 20, 2021

As supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Philip Leong,
October 20, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Aims

With access to large amounts of scientific and internet data, machine learning algorithms, which are designed to automatically discover patterns in data, are increasingly being used to solve complex problems in a wide variety of applications. Examples include self-driving cars [1], speech recognition [2], healthcare [3, 4], recommender systems [5], scientific discovery [6], and even playing board games [7]. The list of examples seems to grow by the day, as new startups emerge, and entire industries are transformed. Machine learning has arguably become this decade's most important technology.

Machine learning algorithms are able to solve problems effectively, in part, because they exploit large datasets and have flexible mathematical models. Unfortunately, the same features that make an algorithm accurate also make machine learning expensive to compute. Flexibility comes from a model with many parameters, while the size of the dataset controls the number of times a model needs to be evaluated during the learning process. This means that machine learning algorithms often require large amounts of memory and computation to reach top-tier performance, usually more than is available with standard computer systems. For instance, Megatron [8] is an 8.3 billion parameter state-of-the-art deep neural network (DNN) language model which was trained on a 37 GB dataset using 512 power-hungry Graphics Processing Units (GPUs). While such a model is extraordinarily large and thus massively overwhelms the energy profile of embedded applications, its existence reveals the direction that innovations in machine learning have taken for better or worse. For this reason, research into hardware accelerators has gathered pace and will be essential in combating the computing challenges of the past, present and future.

The training phase of machine learning is the most computationally intensive, and typically involves linear algebra operations on large matrices and vectors. Performance, as measured by operations per second, depends on the underlying hardware architecture and is either bound by the compute or memory limitations of the system. Compute bound problems (e.g. matrix-multiplication) can benefit from architectures with more parallelism and greater compute density, while memory bound problems (e.g. vector addition) will benefit from architectures that have greater memory bandwidth. Furthermore, performance can be architecture bound if the available compute density can not be utilised efficiently. This means idle compute cycles regardless of whether there is enough memory bandwidth. This often occurs with sparse computations or workloads with irregular data reuse patterns, where it's not always possible to fully pipeline the delivery of data.

Given the growing diversity of machine learning models and applications, designing hardware architectures that maximise performance in a large subset of workloads is non-trivial. There are no easy solutions. The decline of transistor scaling, known as the end of Moore's Law [9], means that finding extra parallelism is much harder for a given silicon area while increasing memory bandwidth comes at the cost of increased power consumption. To overcome these issues and deliver machine learning at greater speed and efficiency, computer architects must think outside the box. Hardware that is aligned more closely with the application, or in other words "specialized", can be particularly advantageous for machine learning. This thesis aims to investigate this idea for two types of algorithms, namely kernel methods and deep neural networks.

Reconfigurable architectures like Field Programmable Gate Arrays (FPGAs) have received considerable interest because they offer specialization at the hardware level and the potential for higher performance compared to conventional Central Processing Units (CPUs). For instance, FPGAs are now being used to accelerate machine learning inference in servers at Microsoft [10] and Intel [11], while FPGAs have always enjoyed an impressive portfolio of use cases in embedded applications. This thesis defines an embedded application as a program (software or firmware) that runs on hardware constrained devices in the field. Examples include mobile phones, radios, video cameras, and satellites, all of which are battery-powered and thus have limited computing capabilities. For machine learning in such devices, FPGAs have mostly been used for inference whereas training is almost exclusively performed using GPUs and Application Specific Integrated Circuits (ASICs) in the datacenter.

This thesis argues for different thinking, and aims to show that with specialized i.)

hardware architectures and ii.) computer arithmetic, FPGAs can also be used to efficiently train machine learning models.

1.1.1 Hardware Architecture

Systolic arrays [12] have emerged as the dominant parallel architecture for machine learning acceleration. This is because systolic arrays lead to high-frequency design and can exploit the maximum amount of parallelism in dense matrix multiplication. Hence for the vast majority of machine learning models, simply offloading all matrix multiplications to dedicated hardware is often sufficient for achieving high computational performance. However, conventional matrix multiplication architectures, such as Nvidia GPUs, are often reliant on making lots off-chip memory transfers and thus carry high energy costs. For this reason, some machine learning experts are diverting their models towards computational kernels which require fewer operations and less memory. Examples include the use sparsity [13], butterfly permutations [14], and random projections [15]. However, since each of these examples introduces some sort of irregularity into the dataflow, their benefits can not be properly realised with commercial GPU and ASIC accelerators. This is an architecture problem that can be resolved with more specialization.

Furthermore, even standard training workloads are difficult to fully optimize using commercial hardware with performance efficiency, in particular, being highly questionable. For example, the Nvidia DGX A100 station contains 8× A100 GPUs capable of 5 petaFLOPs of performance throughput. Such a system trains a ResNet50 [16] model in 28.77 minutes, corresponding to an effective throughput of approximately 700 teraFLOPs. Here lies the opportunity for computer architects. On a widely popular image classification benchmark, a GPU powered supercomputer only sustains about 14% of its peak performance [1]. Specialized architectures and specialized machine learning algorithms can be designed to close this gap.

1.1.2 Computer Arithmetic

The way that numbers are represented is a crucial design decision for accurate and efficient machine learning. Most commercial platforms, such as Nvidia GPUs, employ floating-point representations because they have a wide dynamic range and are known to perform well in mathematically intensive applications. Both factors are important for training DNNs, which rely on repeated calculations involving both very large and very small

[1] Number of operations for training ResNet50 $\approx 1.2 \times 10^{18}$. Throughput = $\frac{\text{Number of operations}}{\text{time in seconds}} \approx \frac{1.2\times10^{18}}{(28.77\times60)} \approx 700 \times 10^{14}$. Ratio between effective and peak throughput = $\frac{700\times10^{14}}{5\times10^{15}} = 14\%$

numbers. The problem with floating-point however lies in the hardware cost and memory overheads associated with standard 16-bit, 32-bit and 64-bit formats. Floating-point adders and multipliers are more than 10× and 1.4× larger and consume approximately 4× and 1.2× more energy than equivalent integer units [17, 18]. For this reason, alternate representations have been sought which are still robust and provide numerical stability, yet have much lower implementation costs. Types of arithmetic specialization may include:

- Reduced precision. Offers arguably the best solution since fewer bits increases computational density and lowers off-chip memory bandwidth requirements. The latter point is particularly significant, since decreasing off-chip memory transfers reduces power consumption, where power, not area is the main factor that limits computing throughput in modern architectures [17].

- Hybrid representations. Research has shown that narrow fixed-point representations can be adopted for machine learning inference without any loss of floating-point accuracy [19]. From the hardware perspective, fixed-point arithmetic offers higher compute density and improved energy efficiency compared to floating-point. However, training with fixed-point suffers from limited dynamic range and must usually be combined with other techniques for adequate convergence. An example is block floating-point [20–22].

- Flexible arithmetic. Resilience to low-precision rounding errors can vary between tasks and between operations within a given task. Therefore, arithmetic which can be configured to suit the needs of the algorithm offers a more robust solution.

This thesis aims to show the benefits of these opportunities for on-chip DNN training, with a particular focus on reduced precision and hybrid representations.

1.2 Contributions

This thesis shows that hardware specialization can be used to achieve machine learning performance not possible with standard computing technology, enabling large-scale and high accuracy training. A particular focus is placed on the embedded application domain, and a number of technical contributions are presented towards on-chip and reduced precision training. The contributions are described in detail below:

Large Scale Kernel Methods: The first FPGA implementation of a large-scale kernel method. The reported design can solve problems with an input dimensionality up to 3 orders of magnitude larger than previous FPGA implementations of kernel methods, and achieves 245× speed up over a single-core Central Processing Unit (CPU). A specialized systolic array architecture is described that preserves data locality and promotes resource reuse to sustain high clock rates and solve massive regression problems. In particular, the architecture efficiently implements the computational bottleneck (FWHT) with a statically configured dataflow and using only 18-bit integer adders. Such an approach is not possible with conventional CPU or GPU architectures.

On-chip DNN Training: The first FPGA implementation of a DNN training technique, which achieves the accuracy of floating-point using mostly 8-bit arithmetic. Specialized arithmetic, namely block floating-point (BFP), is used to reduce the computational cost of training, and flexible hardware/software partitioning is used for higher accuracy. A prototype accelerator is designed for portability across multiple embedded FPGA devices, and this technology is demonstrated for real-time anomaly detection in radio-frequency (RF) networks. Such an application requires hardware specialization to operate effectively under typically strict computation and power constraints.

Sub 8-bit DNN Training: A new number representation called Block Minifloat (BM) for training DNNs with a combination of higher accuracy and greater hardware efficiency than previous 8-bit regimes. BM formats are designed to have both a wide dynamic range and yield an efficient implementation via Kulisch accumulators. Training is demonstrated using sub 8-bit BM formats for all weights, activations and gradients of a DNN, and almost no degradation in floating-point accuracy is observed with arithmetic units that are 23.9× smaller and consume 16.1× less energy than floating-point. This shows that specialized arithmetic can be used to overcome both memory and computation bottlenecks, to deliver training at increased speed and energy efficiency.

1.3 Thesis Structure

The main contributions of this thesis have been published before in:

- Sean Fox, David Boland, and Philip HW Leong. "FPGA Fastfood-A High Speed Systolic Implementation of a Large Scale Online Kernel Method." ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2018.
- Sean Fox, Julian Faraone, David Boland, Kees Vissers, and Philip HW Leong. "Training Deep Neural Networks in Low-Precision with High Accuracy using FPGAs". International Conference on Field-Programmable Technology (ICFPT). 2019. (Best Paper Award Candidate)
- Sean Fox, Seyedramin Rasoulinezhad, David Boland, and Philip HW Leong. "A Block Minifloat Representation for Training Deep Neural Networks". International Conference on Learning Representations (ICLR). 2021.

However, this thesis takes the opportunity to explain and compare the material in more detail. In particular, chapters are written around each publication and describe hardware specialization for machine learning in terms of the underlying hardware architecture and computer arithmetic. The structure of technical chapters is thus encapsulated by Figure 1.1, and the overall thesis structure is given below:

- Chapter 2 provides background on machine learning algorithms: kernel methods and deep neural networks (DNNs), hardware architectures: Systolic Arrays and Field Programmable Gate Array (FPGA) technology, and computer arithmetic: Fixed-point, Floating-point and Block floating-point.
- Chapter 3 describes a hardware design for training large-scale online kernel methods, comprising a specialized systolic-array architecture and fixed-point arithmetic scheme.
- Chapter 4 considers the problem of high accuracy on-chip DNN training and proposes a specialized 8-bit algorithm and prototype accelerator for embedded FPGA devices.
- Chapter 5 shows that DNN training can be performed with fewer bits and higher computational density using greater specialization at the arithmetic level. A new representation called Block Minifloat is introduced.
- Lastly, conclusions of this work are discussed in Chapter 6.

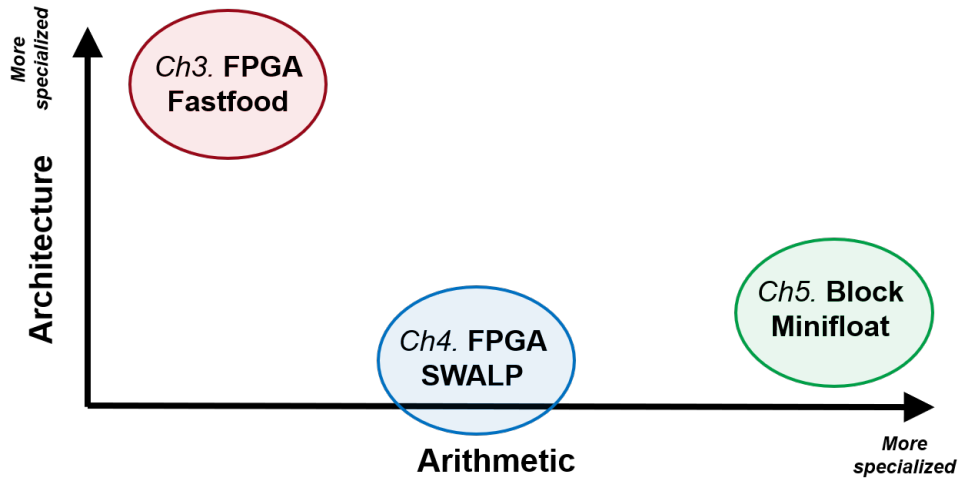


Figure 1.1: Thesis structure. Each chapter explores hardware specialization along two axes: hardware architecture (y-axis) and computer arithmetic (x-axis)

Chapter 2

Background

This chapter presents a brief background on machine learning and hardware specialization. This includes theory and descriptions for gradient-based learning algorithms, kernel methods, deep neural networks (DNNs), hardware accelerator design principles, systolic arrays, Field Programmable Gate Arrays (FPGAs), fixed-point arithmetic, floating-point arithmetic, and finally, block floating-point arithmetic.

The presentation of this chapter is organised based on the three main investigative themes of this thesis: Machine learning in Section 2.1, hardware architectures in Section 2.2, and lastly, computer arithmetic in Section 2.3.

2.1 Machine Learning

Machine learning defines a set of computer algorithms that use experience to automatically improve performance at some task. A more precise definition was given in (Mitchell, 1997) [23] which says:

Definition 2.1 *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

Here, experience refers to past information that is available to the learner and forms the training dataset while performance measures the ability of the learner to generalise to previously unseen data. For example, consider the task of image classification (T). A machine learning algorithm improves its classification accuracy (P) using experience obtained by training over large collections of already labelled images (E). Likewise, on the task of forecasting energy prices (T), machine learning uses temporal patterns extracted from historical price data (E) to predict future unknown prices with higher certainty and

lower error (P). These examples are summarized in Table 2.1.

Table 2.1: Examples of machine learning problems

task (T)	experience (E)	performance (P)
image classification (e.g. ImageNet [24])	large collection of labelled images	classification accuracy
language modelling	text corpora containing many words and sentences	accuracy of the next word probability model (i.e. word perplexity)
non-linear regression (energy prices)	historical timeseries of energy prices plus any related timeseries data	forecasting error, as measured by e.g. mean square error (MSE)
anomaly detection (RF data)	IQ samples from spectrum of radio frequency signals	correct detections of known/injected anomalies

To formulate a learning algorithm mathematically, we first need to define a model. The model refers to a function $f(x, \theta)$ that predicts a target y by inputting x and generating the output $\hat{y} = f(x, \theta)$. The precise form of the function is determined during the training or *learning* phase, by improving predictions on the training data, while performance is evaluated during the testing/inference phase on examples not found in the training set. The ability to perform well on both the train and test sets is called *generalisation* and is the main difference between machine learning and standard optimization algorithms.

This section is organised as follows: First, machine learning algorithms are described in more practical terms using linear regression as an example. This includes notation and terminology which is relevant to subsequent sections. While linear models provide a good understanding of how machine learning algorithms work, most practical problems require non-linear models for effective learning outcomes. This section also provides the required background and theory on two classes of non-linear algorithms, namely kernel methods and deep neural networks.

2.1.1 Example: Linear Regression

Linear regression [25] has been widely used for solving problems in machine learning and related fields of science. It has a solid mathematical foundation and provides a good working example of how machine learning algorithms operate.

Linear regression takes an input vector $\mathbf{x} \in \mathbb{R}^d$, where d refers to the number of input variables otherwise known as *features*, and generates a scalar output $\hat{y} \in \mathbb{R}$ that predicts a target value $y \in \mathbb{R}$. Importantly, the output is modelled as a linear function of the input, as

shown in Equation (2.1), where $\mathbf{w} \in \mathbb{R}^d$ is a vector of parameters called *weights*.

$$\hat{y} = \mathbf{w}^\mathsf{T}\mathbf{x} \tag{2.1}$$

Linear models are sometimes formulated as affine functions. The affine function includes an additional parameter $b \in \mathbb{R}$, called the offset or *bias*, so that $\hat{y} = \mathbf{w}^\mathsf{T}\mathbf{x} + b$. The affine function can be realised with Equation (2.1) by appending the bias term to the weight vector and augmenting the input vector with an extra feature that always equals 1. The augmented input and weight vectors then become $\mathbf{x} = \{x_1, x_2, \ldots, x_d, 1\}$ and $\mathbf{w} = \{w_1, w_2, \ldots, w_d, b\}$ respectively.

Let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\} \in \mathbb{R}^{N \times d}$ represent a dataset of N examples, and let $\mathbf{y} = \{y_1, y_2, \ldots, y_N\} \in \mathbb{R}^N$ represent the corresponding set of target values. In standard machine learning problems, model performance is measured by computing some metric of error, called a *loss function*, on the testing set $\{\mathbf{X}^{(\text{test})}, \mathbf{y}^{(\text{test})}\}$. For linear regression, mean square error (MSE) is commonly used.

$$\begin{aligned} \text{MSE}^{(\text{test})} &= \frac{1}{N}\sum_{i=1}^{N}(\hat{\mathbf{y}}^{(\text{test})} - \mathbf{y}^{(\text{test})})_i^2 \\ \text{MSE}^{(\text{test})} &= \frac{1}{N}\left\|\hat{\mathbf{y}}^{(\text{test})} - \mathbf{y}^{(\text{test})}\right\|_2^2 \end{aligned} \tag{2.2}$$

However, before MSE performance can be evaluated on a testing set, a model must first be trained on a training set $\{\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})}\}$. This requires an algorithm for finding weights, $\mathbf{w}$, that minimises MSE on the training data. Formally, the problem can be written as:

$$\underset{\mathbf{w}}{\text{minimize}} \quad \frac{1}{N}\left\|\hat{\mathbf{y}}^{(\text{train})} - \mathbf{y}^{(\text{train})}\right\|_2^2 \quad , \tag{2.3}$$

from which a direct solution for $\mathbf{w}$ is found by calculating where the gradient equals zero, according to Equations (2.4) to (2.9) below.

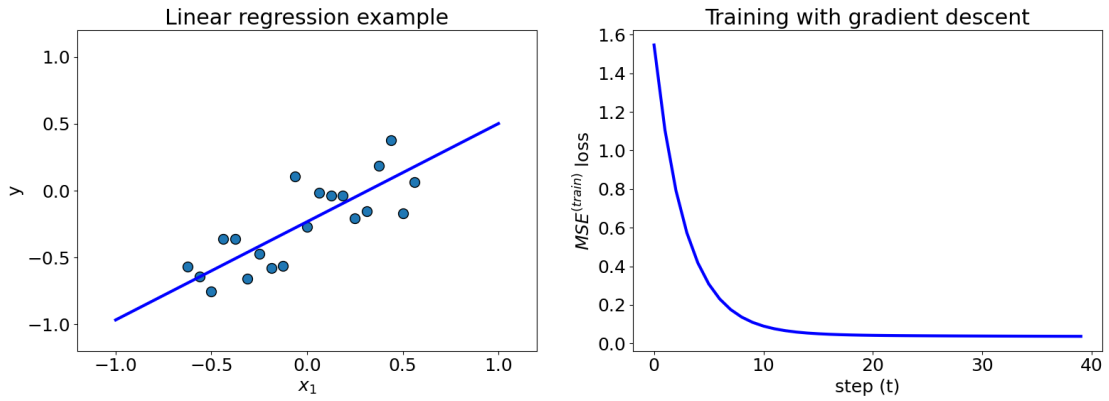


Figure 2.1: Example of linear regression and gradient descent on a toy problem

$$\frac{\partial \text{MSE}^{(\text{train})}}{\partial \mathbf{w}} = 0 \tag{2.4}$$

$$\frac{\partial}{\partial \mathbf{w}}\left(\hat{\mathbf{y}}^{(\text{train})} - \mathbf{y}^{(\text{train})}\right)^{\mathsf{T}}\left(\hat{\mathbf{y}}^{(\text{train})} - \mathbf{y}^{(\text{train})}\right) = 0 \tag{2.5}$$

$$\frac{\partial}{\partial \mathbf{w}}\left(\mathbf{X}^{(\text{train})}\mathbf{w} - \mathbf{y}^{(\text{train})}\right)^{\mathsf{T}}\left(\mathbf{X}^{(\text{train})}\mathbf{w} - \mathbf{y}^{(\text{train})}\right) = 0 \tag{2.6}$$

$$\frac{\partial}{\partial \mathbf{w}}\left(\mathbf{w}^{\mathsf{T}}\mathbf{X}^{(\text{train})\mathsf{T}}\mathbf{X}^{(\text{train})}\mathbf{w} - 2\mathbf{w}^{\mathsf{T}}\mathbf{X}^{(\text{train})\mathsf{T}}\mathbf{y}^{(\text{train})} + \mathbf{y}^{(\text{train})\mathsf{T}}\mathbf{y}^{(\text{train})}\right) = 0 \tag{2.7}$$

$$2\mathbf{X}^{(\text{train})\mathsf{T}}\mathbf{X}^{(\text{train})}\mathbf{w} - 2\mathbf{X}^{(\text{train})\mathsf{T}}\mathbf{y}^{(\text{train})} = 0 \tag{2.8}$$

$$\mathbf{w} = \left(\mathbf{X}^{(\text{train})\mathsf{T}}\mathbf{X}^{(\text{train})}\right)^{-1}\mathbf{X}^{(\text{train})\mathsf{T}}\mathbf{y}^{(\text{train})} \tag{2.9}$$

Since it is not possible to derive closed-form solutions (like above) for all learning models and functions, another way of solving for $\mathbf{w}$ is to train via *gradient descent*. In gradient descent, the weights are updated iteratively, by taking small steps along the direction of the negative gradient. Mathematically, the update algorithm is written as:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta\frac{\partial H}{\partial \mathbf{w}} \tag{2.10}$$

where t is the step number, η refers to the step size otherwise known as the *learning rate* and $H = \text{MSE}^{(\text{train})}$ is the loss function. Equation (2.10) defines the loss with respect to the training set, and therefore the entire training set must be processed at each step until convergence. This means that gradient descent does not scale particularly well on large datasets. An example of gradient descent is shown in Figure 2.1. Here, a simple 2D linear regression model is trained on a toy dataset comprising 20 examples, reaching convergence after approximately 20 steps, where $\mathbf{w}_1 = 0.74$ and $b = -0.23$.

Algorithm 1: Pseudocode for Stochastic Gradient Descent (SGD). Note that η is the learning rate, $\mathcal{D} = (\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$ defines the training set, the minibatch size is given by K, and t is the step number

Initialise the model weights: $\mathbf{w}(t = 0)$
while Not converged **do**
  sample a minibatch $\mathbb{B} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_K, y_K)\}$ uniformly from $\mathcal{D}$
  $\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \frac{1}{K} \sum_{i=1}^{K} (\mathbf{w}(t)^\mathsf{T} \mathbf{x}_i - y_i)\mathbf{x}_i$
  $t = t + 1$
end while

Due to performance scalability issues associated with gradient descent, a similar algorithm called **Stochastic gradient descent (SGD)** [26] is often preferred for machine learning training. SGD, as shown in Algorithm 1, works by estimating the gradient using a smaller set of examples, known as a minibatch, which are sampled uniformly from the dataset. Crucially, the size K of the minibatch can be held constant while the training set size N increases. This reduces the computational cost for each step by N/K times. Furthermore, since the loss function is now stochastic, in the sense that evaluation depends on the randomly sampled minibatch, the learning algorithm tends to be better at escaping from bad local minima. This has proven especially important for training deep neural networks, which is the main subject of Chapters 4 and 5.

2.1.2 Kernel Methods

To adapt linear models for **non-linear** problems, the inputs should first be transformed into a higher dimensional feature space using a function $\phi(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}^D$. This is because features become more clearly separable in high dimensions, and thus the output can be computed using simple linear techniques. This is shown in Equation (2.11), where $\mathbf{w} = \{w_1, \ldots, w_D\}$ is a vector of weights.

$$f(\mathbf{x}) = \sum_{i=1}^{D} w_i \phi(\mathbf{x}) \tag{2.11}$$

Although the linear part of the equation is computationally efficient, $\phi(x)$ is itself expensive to compute for very high dimensional feature spaces. In such cases, it would be better to define the model using kernel methods [27].

Kernel methods rely on the observation that inner products between pairs of points in high dimensional feature spaces can be computed using a kernel function, $\kappa(\mathbf{x}_i, \mathbf{x}_j)$, without

having to explicitly transform into that space, as shown in Equation (2.12).

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = <\phi(\mathbf{x}_i), \phi(\mathbf{x}_j)> \tag{2.12}$$

This enables high-dimensional features to be extracted inexpensively from the input, as the computation of $\phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j)$ can be avoided. This is commonly known as the *kernel trick* [28]. The only caveat is that a length M subset of the input data must be stored in memory, also known as the *dictionary* or *support vectors*. If $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\} \in \mathbb{R}^{N\times d}$ and $\mathbb{D} = \{\mathbf{v}_1, \ldots, \mathbf{v}_M\} \in \mathbb{R}^{M\times d}$ represents the input data and dictionary respectively, then $f(\mathbf{x})$ can be computed using Equation (2.13) for the jth input.

$$f(\mathbf{x}_j) = \sum_{i=1}^{M} w_i \kappa(\mathbf{v}_i, \mathbf{x}_j) \tag{2.13}$$

Different types of kernel functions exist for solving different types of problems, as shown in Table 2.2. The most common kernel however is the Gaussian kernel, also known as the Radial Basis Function (RBF) kernel. All kernel functions compute some measure of similarity between two input examples. This can be done using the inner product $\mathbf{x}_i^T\mathbf{x}_j$, as with the linear kernel, or by the distance $\left\|\mathbf{x}_i - \mathbf{x}_j\right\|_2^2$, in the Gaussian kernel. Broadly speaking, the similarity calculation is the main source of computational cost in the kernel function.

Table 2.2: Examples of kernel functions

linear kernel	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T\mathbf{x}_j$
gaussian kernel	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma\|\mathbf{x}_i-\mathbf{x}_j\|_2^2}$
polynomial kernel	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T\mathbf{x}_j + c)^b$
bessel kernel	$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathcal{J}_{v+1}(\sigma\|\mathbf{x}_i-\mathbf{x}_j\|)}{\|\mathbf{x}_i-\mathbf{x}_j\|^{v+1}}$

To train kernel methods, the machine learning algorithm must update both the dictionary and weights in order to minimise some predefined loss function. This is in contrast to linear methods which only require updates for the weights. On small datasets (less than 10^4 examples), Equation (2.13) can be computed efficiently. However, this is not possible on large datasets because M tends to grow with the number of input examples, N, [29]. This significantly limits kernel methods in applications such as image classification and language modelling, where for example, the widely used ImageNet dataset contains $N = 10^6$ images and $d = 10^5$ input dimensions [30].

2.1.2.1 Random Kitchen Sinks

To address the problem associated with a large M, Rahimi and Recht [31] proposed random kitchen sinks. The main idea is to create a function $z(\mathbf{x})$ such that the inner products $\langle z(\mathbf{x}_i), z(\mathbf{x}_j) \rangle$ are approximately equal to the high-dimensional features extracted from $k(\mathbf{x}_i, \mathbf{x}_j)$. In this case $f(\mathbf{x}_j)$ can be computed by Equation (2.14). Rahimi and Recht demonstrated that this is guaranteed for shift-invariant kernels ($\kappa(\mathbf{x}, \mathbf{x}') = \kappa(\mathbf{x} - \mathbf{x}', 0)$), if we define $z(\mathbf{x})$ by Equation (2.15).

$$f(\mathbf{x}_i) = \sum_{j=1}^{n} \alpha_j z(\mathbf{x}_i) \tag{2.14}$$

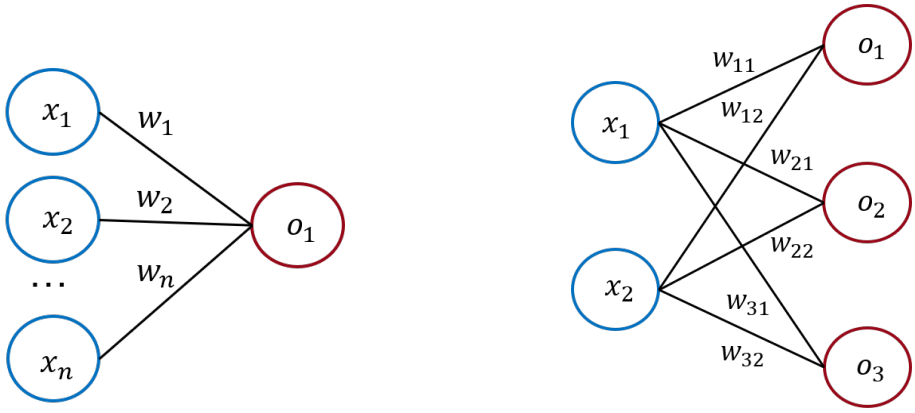
$$z(\mathbf{x}_i) = \frac{1}{\sqrt{n}} cos(\mathbf{W}^T \mathbf{x}_i) \tag{2.15}$$

In this function, $\mathbf{W}$ is a $d \times m$ matrix randomly sampled from the Fourier transform of the kernel function. This is of particular interest for the Gaussian kernel for which the Fourier transform is itself a Gaussian. Crucially, m is fixed and represents the number of random basis functions (or expansion dimensions) used to approximate kernel methods with M dictionary elements. This means that random kitchen sinks can be trained independent of the dataset size, making it much faster on problems with many inputs. In fact, on the CIFAR-10 dataset [32], which has $N = 50,000$ images and $d = 3,072$ dimensions, exact non-linear kernel methods require very large M and are intractable. On the other hand, random kitchen sinks with $m = 16,384$ achieves an accuracy that is among the top two for shift-invariant kernel representations [33].

Unfortunately, although the computational cost does not grow with the dataset, N, random kitchen sinks still must store $\mathbf{W}$ and compute $\mathbf{Wx}$ for each $(\mathbf{x}_i, y_i)$ pair. For high dimensional datasets, i.e. $d > 1000$, this consumes time and energy because memory and compute requirements of vector-matrix products scale as $O(md)$.

2.1.3 Deep Neural Networks

Deep Neural Networks (DNNs) are powerful multi-layered feature extractors [34]. They transform raw inputs via a sequence of computational layers into simpler yet more abstract features at the output. Let $\mathbf{x} \in \mathbb{R}^n$ represent an input vector of dimensionality n, and let $f'(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ represent the function to be computed at each layer. A DNN comprising



(a) Neuron with n inputs

(b) Layer with 2 inputs and 3 outputs

Figure 2.2: An example DNN layer. Layers are composed of multiple neurons, where each neuron computes a weighted sum of inputs for a single output.

L layers can then be written as:

$$\phi(\mathbf{x}) = f^L(\ldots(f^2(f^1(\mathbf{x})))) \tag{2.16}$$

Each layer is composed of smaller computational units called neurons, as shown in Figure 2.2. Neurons are inspired by biological processes in the brain and compute a function of the form: $g(\mathbf{w}^T\mathbf{x} + b)$, where $\mathbf{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$ are trainable weights and bias parameters of a linear model, and $g(.)$ is a non-linear function such as ReLU, Sigmoid and Tanh functions (from Table 2.3). Each neuron computes a single output feature, and thus multiple neurons are required to produce layers with multiple output features. This changes the main computation from a dot-product to matrix-multiplication as formulated below:

$$f(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}) = g(\begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} . \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix})$$

$$f(\mathbf{x}) = g(\mathbf{W}^T\mathbf{x} + \mathbf{b}) \tag{2.17}$$

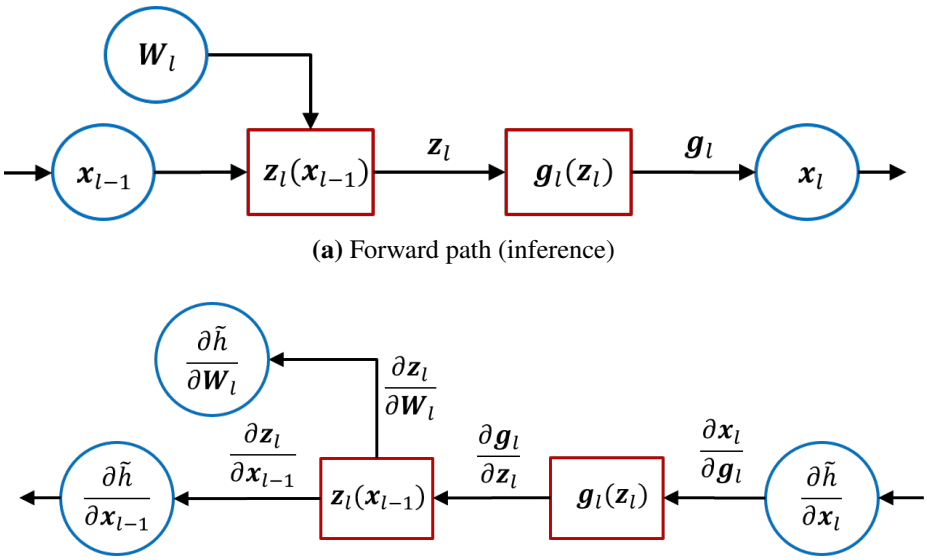
where $\mathbf{x} \in \mathbb{R}^n$ and $f(\mathbf{x}) \in \mathbb{R}^m$ refer to the input and output of the layer, commonly called the *activations*, and $\mathbf{W} \in \mathbb{R}^{n \times m}$ and $\mathbf{b} \in \mathbb{R}^m$ are the weight matrix and bias vector respectively. Equation (2.17) describes the fundamental computation at the heart of all layers.

The model capacity of a DNN is controlled by the depth (number of layers) and width (neurons per layer) of the network, while the connectivity between neurons and layers is

Table 2.3: Examples of non-linear activation functions

	$g(z)$	dg/dz
ReLU	$\begin{cases} 0 & \text{for } z < 0 \\ z & \text{for } z \geq 0 \end{cases}$	$\begin{cases} 0 & \text{for } z < 0 \\ 1 & \text{for } z \geq 0 \end{cases}$
Sigmoid	$\frac{1}{1+e^{-z}}$	$g(z)(1-g(z))$
Tanh	$\frac{2}{1+e^{-2z}} - 1$	$1 - g(z)^2$

also part of the modelling framework. Modern DNNs are designed with many parameters (i.e both wide and deep) which are free to learn very complex feature representations. This provides DNNs with powerful generalisation properties but means that each prediction also requires a vast number of large matrix multiplication operations.



(a) Forward path (inference)

(b) Backward path (training)

Figure 2.3: Summary of the forward and backward computations required for a given DNN layer.

2.1.3.1 Training

Training is performed by first establishing and then minimising an objective of the form:

$$H(\mathbf{W}) = \frac{1}{K} \sum_{i=1}^{K} h_i(\mathbf{W}) \tag{2.18}$$

Here, $h(\mathbf{W})$ is a loss function (e.g. $\text{MSE}^{(\text{train})}$, cross-entropy etc.) and K is the number of training batches. For ease of notation, the loss is defined as a function of the weights only.

Stochastic gradient descent (SGD) in Algorithm 1 is commonly used to minimise the objective. On each iteration, SGD will select a random batch from the training set, corresponding to a function $\tilde{h}$ from $\{h_1, h_2, ..., h_m\}$, and update the weights in the direction of steepest descent, according to the following equation and learning rate η.

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta \frac{\partial \tilde{h}}{\partial \mathbf{W}^t} \tag{2.19}$$

Here, $\frac{\partial \tilde{h}}{\partial \mathbf{W}}$ refers to the weight gradient or alternatively the derivative of the loss with respect to the weight. This needs to be computed for every weight at every layer in the network, and requires application of the chain rule to propagate the loss back through the network. This is called backpropagation and the loss at each layer is called the activation gradient (or deltas), denoted $\boldsymbol{\delta}_l = \frac{\partial \tilde{h}}{\partial \mathbf{x}_l}$. If $z(\mathbf{x}) = \mathbf{W}^T\mathbf{x}$, then at each computation layer we must compute the forward path by Equation (2.17) and the backward path by Equations (2.20) and (2.21), for the activation gradient and weight update gradient respectively.

$$\frac{\partial \tilde{h}}{\partial \mathbf{x}_{l-1}} = \frac{\partial z}{\partial \mathbf{x}_{l-1}} \frac{\partial g}{\partial z} \frac{\partial \mathbf{x}_l}{\partial g} \frac{\partial \tilde{h}}{\partial \mathbf{x}_l} \tag{2.20}$$

$$\frac{\partial \tilde{h}}{\partial \mathbf{W}_l} = \frac{\partial z}{\partial \mathbf{W}_l} \frac{\partial g}{\partial z} \frac{\partial \mathbf{x}_l}{\partial g} \frac{\partial \tilde{h}}{\partial \mathbf{x}_l} \tag{2.21}$$

A summary of the computation is illustrated in Figure 2.3. Since $\frac{\partial z}{\partial \mathbf{x}} = \mathbf{W}$ and $\frac{\partial z}{\partial \mathbf{W}} = \mathbf{x}$, then both equations reduce to matrix multiplication by the activation gradient $\tilde{\boldsymbol{\delta}}_l = \frac{\partial g}{\partial z} \frac{\partial \mathbf{x}_l}{\partial g} \frac{\partial \tilde{h}}{\partial \mathbf{x}_l}$.

DNN training introduces an additional two matrix multiplications at each computation layer, and furthermore, the input batch at each layer of the forward path must be saved to calculate the weight gradient in the backward path. Specialized hardware architectures and specialized arithmetic can be used to reduce these computation and memory costs, such as systolic arrays and low-precision number representations.

2.2 Hardware Architecture

Machine learning algorithms usually require many parameters and operations to reach high levels of accuracy. Software running on a general-purpose central processing unit (CPU) is typically too slow and consumes too much energy for effective computation. This reality together with the growing size and diversity of machine learning applications has pushed the need for purpose-built hardware accelerators [35], such as Graphics Processing Units (GPUs), Application Specific Integrated Circuits (ASICs) and Field Programmable Gate

Arrays (FPGAs). This section presents background on hardware acceleration, focusing on FPGAs and architectures for high-performance and efficient machine learning.

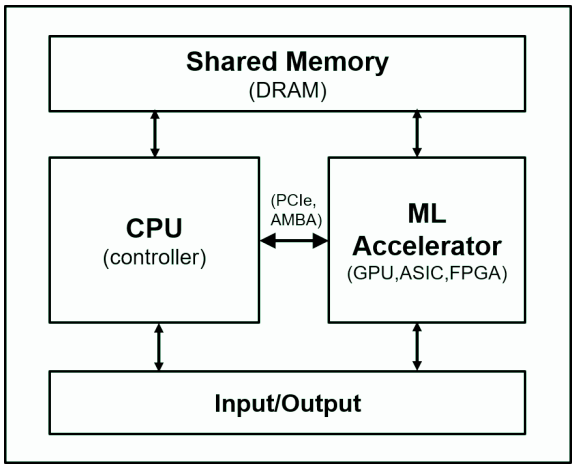


Figure 2.4: Heterogeneous machine learning accelerator

Most hardware accelerators are designed to operate alongside a CPU or an equivalent microcontroller and are thus described as heterogeneous, shown in Figure 2.4. The CPU controls the execution of the program by offloading the computationally intensive parts of the machine learning algorithm to the accelerator, where an offload is performed by transferring data between the CPU and accelerator address spaces in shared memory. The CPU and accelerator can be connected on the same integrated circuit (or chip) via an interconnect standard such as AMBA (for ARM processors) and QPI (for Intel processors), or otherwise as separate devices by external interfaces like PCI-e or Ethernet. This thesis focuses on embedded devices, where the hardware accelerator is usually packaged alongside other components and forms a system on chip (SoC) [36].

2.2.1 Design Objectives

Machine learning accelerators are designed to perform tasks more efficiently than standard computer technology. However, the extent to which the hardware accelerator actually delivers superior performance depends on the machine learning model as well as the application itself. Therefore, realising high-performance hardware is very much a moving target for computer architects [37]. Below, four objectives are outlined that are commonly used to guide the design process.

Accuracy. Performance is first and foremost measured by the model's ability to make effective and accurate predictions. In terms of hardware, numerical accuracy is determined by the underlying computer arithmetic. The number of bits in the number representation, and the amount of range and precision allowed. This is discussed in detail in Section 2.3.

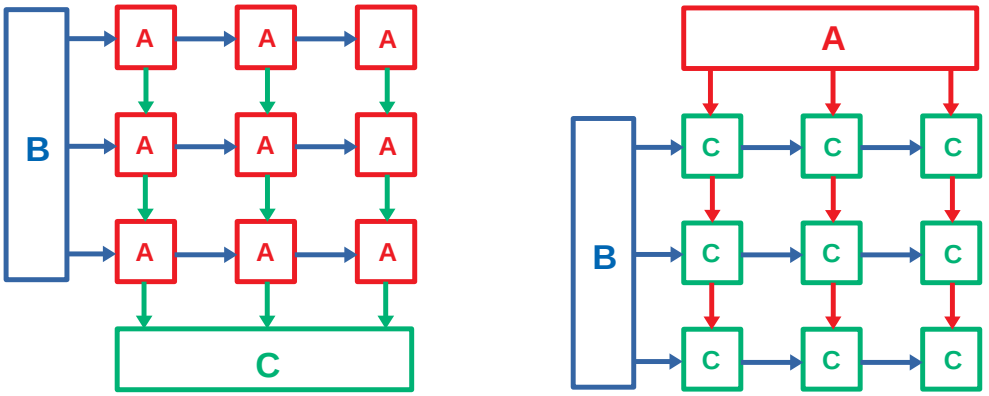
Throughput and Latency. The time taken for inference and training depends on the computational throughput and latency of the underlying hardware architecture. Throughput measures the number of operations or inputs which can be computed per second, while latency refers to the time taken (in seconds) for a single operation or input. For achieving both high throughput and low latency, hardware architectures must be able to maximise computational parallelism while sustaining a high operating frequency and preventing idle time as much as possible. This requires a large silicon area and enough memory bandwidth to keep the computational units busy. Nvidia GPU's are the dominant platform for accelerating training workloads.

Energy and Power. Energy efficient architectures are based on keeping data close to the computation units and avoiding off-chip memory transfers. According to Horowitz [17] this is more important for further performance improvements than increasing the transistor count. Brainwave [10] is an FPGA chip for inference which ties the entire model to 28 MB of on-chip memory, and thus eliminates expensive DRAM transfers. Doing the same for training is more difficult because of increased memory requirements, i.) more bits are typically needed to preserve numerical accuracy, and ii.) the input activations must be stored for updating the weights via gradient descent techniques. Specialized hardware together with specialized learning algorithms is desirable to keep more data on-chip and reduce energy costs associated with training. This is especially important for embedded devices which have strict physical constraints.

Flexibility. An architecture that achieves high throughput (or low latency) on one task, might be unable to reach similar performance levels on other tasks. For example, architectures that are optimised for high data reuse (e.g. matrix multiplication, convolution) may perform poorly on models with less data reuse (e.g. vector operations, depthwise convolution). This could be due to insufficient memory bandwidth, but also because the network on chip (NoC) architecture is not flexible enough to deliver data efficiently to the computation units for different workloads. This thesis focuses on the training phase, and thus greater flexibility is required to effectively support gradient descent techniques. For DNNs in particular, backpropagation introduces two additional matrix multiplications (Equation 2.20 and 2.21) which can have vastly different data reuse properties.

2.2.2 Systolic Arrays

A systolic array (SA) defines a parallel computer architecture comprising a large number of locally interconnected processing elements (PEs), which are arranged in a spatial structure such as a linear or two-dimensional array [12]. Computation is performed in a pipelined



(a) Spatial dataflow **(b)** Temporal dataflow

Figure 2.5: (a) Spatial dataflow: B is reused horizontally and C is reused vertically. (b) Temporal dataflow: B is reused horizontally and A is reused vertically.

manner by passing data through the array in a regular pattern, where each PE works on part of the computation in parallel. Inputs to the array are buffered and thus synchronised with a specific dataflow pattern. Hardware benefits of this architecture include:

(i) A modular and regular structure which is flexible and easily extensible. The array can be shaped and resized to meet specific I/O bandwidth and computational requirements with simple modifications and without needing to change the PEs.

(ii) Local interconnections. Each PE is only connected to its neighbours, preserving data locality within the PE and ensuring that wire lengths and subsequent routing delays are minimal. This means the design can operate at high frequencies and thus achieve higher throughput and lower latency.

(iii) Massive parallelism. Systolic arrays exploit parallelism in computationally intensive workloads, such as convolution and matrix multiplication, by reusing the input data multiple times as it moves through the pipelined array. This improves the efficiency and speed of machine learning applications, which are often constrained by the amount of computation required. Google's Tensor Processing Unit (TPU) [38] is perhaps the most well-known machine learning accelerator based on systolic array dataflows.

Systolic arrays require careful scheduling to ensure the input data arrives at each PE on the correct clock cycle. The shape of the array and the supported dataflow is also important for the execution of a given instruction/operation. Below, two dataflows are presented for matrix multiplication.

2.2.2.1 Dataflow for Matrix Multiplication

There are two main types of systolic arrays for matrix multiplication. The spatial reuse architecture and temporal reuse architecture, as shown in Figure 2.5. For $C = A \times B$, each dataflow works as follows:

Spatial Dataflow. The output, C, is accumulated spatially by propagating partial sums vertically down the array. B is reused horizontally (right), while A is cached in local PE memory and is therefore stationary. In the context of neural networks, where A and B refer to the weight and activation tensors respectively, this dataflow is commonly known as the *weight stationary* dataflow [39].

Temporal Dataflow. The output, C, is cached in local PE memory and accumulated temporally. A and B on the other hand are reused vertically (down) and horizontally (right). For neural networks, this dataflow is also known as the *output stationary* dataflow [39].

Importantly, both dataflows rely on memory bandwidth and data reuse in certain dimensions of A, B, and C to achieve high parallelism and performance. Hence, in terms of hardware, the "best" dataflow depends entirely on the machine learning workload. Given that two models for the same task can have widely varying data reuse patterns, it is challenging for systolic arrays to maximise performance for all workloads and applications.

In Chapter 3, a highly specialized systolic array architecture is presented for a low data reuse algorithm comprising structured sparse matrices and log-linear time complexity. Then Chapter 4, another systolic array is described for training deep neural networks. On this occasion, the architecture is a standard 2D array with spatial data reuse.

2.2.3 Field Programmable Gate Arrays

Field Programmable Gate Array (FPGA) is a type of integrated circuit that can be configured to a particular application or functionality by the consumer after manufacturing [40]. This distinguishes FPGAs from other implementation options, such as ASICs, which require a newly fabricated chip for every new design [41].

Hardware flexibility can be especially advantageous for machine learning, where both the algorithm and accelerator architecture are moving targets in an optimal implementation. For example, techniques such as sparsity and low-precision arithmetic do not perform with the same accuracy for all models and applications yet they require specialized hardware to realize their benefits. Apart from flexibility, FPGA architectures also deliver very high

computational performance. In fact, recent FPGAs provide upwards of 7 tera-operations per second (TOPs) of 8-bit throughput, have tens of megabytes (MB) in on-chip memory, multiple terabytes per second (TB/s) of low-latency communication bandwidth, and operate well above 700 MHz [42] [1]. Furthermore, FPGAs can be easily integrated with other hardware components to form full system applications.

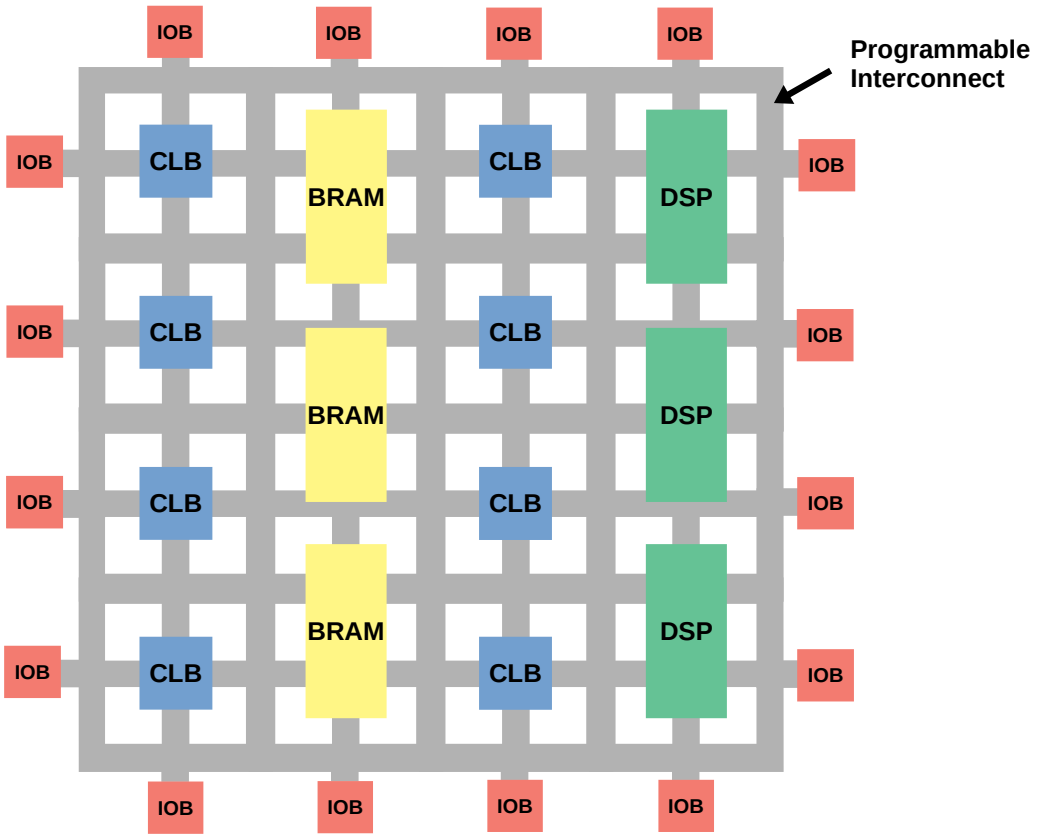


Figure 2.6: Components of an FPGA architecture

Modern FPGA architectures are composed of five key components: configurable logic blocks (CLB), digital signal processing (DSP) blocks, memory blocks known as block ram (BRAM), I/O blocks (IOB), and programmable routing, as shown in Figure 2.6. An FPGA implements a given circuit by programming each of the aforementioned blocks and configuring switches in the routing architecture to make all necessary connections. As a result, the overall routing architecture (including the number of wires in each channel) is particularly important to the FPGA's flexibility, speed and area efficiency. Figure 2.6 shows an "island-style" FPGA architecture, where logic blocks are surrounded by routing channels on all four sides. In terms of logic, CLBs use look-up tables (LUTs) for functionality and flip-flops (FF) for memory, with local connections between them. A k-input LUT implements any function of k inputs as a truth table and requires 2^k memory cells and

[1] Numbers assume a Xilinx Ultrascale+ VU3P device

a 2^k input multiplexer. The use of LUTs means that CLBs can also be configured as a form of distributed memory. DSPs are dedicated circuits for math operations and contain adders and multipliers which would otherwise consume large amounts of CLB resources. Similarly, BRAMs provide a dedicated resource of memory, usually tens of kilobits per BRAM, which can be configured as single-port or dual-port memory. Both these blocks are fully pipelined and can thus perform computations and read/write operations every clock cycle at high frequencies. Most FPGAs contain many DSP and BRAM resources in their device architecture, taking up most of the area on the chip, the use of which is vital for achieving high performance in computationally intensive applications.

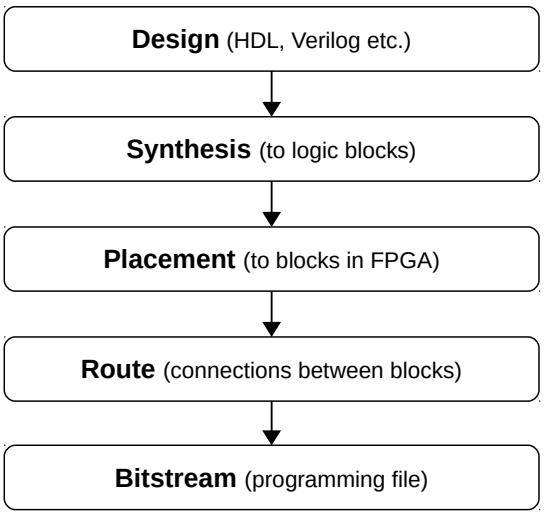


Figure 2.7: FPGA design flow

FPGA programming is a multi-step process, as shown in Figure 2.7. A hardware description language (HDL) is typically used to describe a circuit at a high level of abstraction in the design phase. The circuit, as described by the HDL file, is then converted into a netlist of FPGA resources (e.g. LUTs, FFs, DSPs, BRAMs, etc.). This step is called synthesis, the aim of which is to maximize the circuit speed while simultaneously minimizing the number of resources needed. Next, the netlist is mapped to physical blocks on the FPGA using placement and routing algorithms which optimize the timing of the circuit. Finally, the circuit is written to the FPGA as a sequence of bits, called a bitstream. The bitstream implements the desired functionality of the circuit by configuring the thousands (possibly millions) of resources and switches on the FPGA.

Modern FPGAs provide programmability and high performance, both of which are advantageous for machine learning acceleration.

2.3 Computer Arithmetic

Computer arithmetic concerns the study of number representations, algorithms for manipulating and operating on numbers, and hardware circuits and software routines for their efficient implementation [43]. This section presents fixed-point, floating-point and block floating-point arithmetic types, which together generalise nearly all number systems used for machine learning.

As discussed above, Field Programmable Gate Arrays (FPGAs) consist of logic blocks that can be reconfigured to implement any type of computer arithmetic. However, the extent to which the implementation is efficient in terms of area, speed and power consumption, varies between arithmetic types. With that in mind, this section provides a brief discussion on the hardware implications of each representation.

2.3.1 Fixed-Point

Fixed-point is a digital number representation that operates similarly to integers [43]. As with integers, fixed-point numbers are represented uniformly which means that gaps between successive numbers are the same throughout the representation. However, fixed-point can also represent fractional numbers. Specifically, fixed-point assigns a portion of the representation to the fractional part of real numbers, called fractional bits F. This is equivalent to scaling an integer by 2^{-F} and can be interpreted as trading off representable range for more numerical precision. Putting this together, a bit vector $(x_{K-1}x_{K-2}\dots x_0.x_{-1}x_{-2}\dots x_{-F})_2$ in fixed-point, represents the real value given by:

$$(x_{K-1}x_{K-2}\dots x_0.x_{-1}x_{-2}\dots x_{-F})_2 = \sum_{i=-F}^{K-1} x_i \times 2^i \tag{2.22}$$

Here, $B = K + F$ is the bit width of the number, where K is the number of bits after the decimal point otherwise known as the integer width.

Notably, this formulation does not handle translations from signed fixed-point numbers, which are usually represented in two's complement form [43]. In two's complement, the most significant bit encodes the sign, where zero and one represent positive and negative numbers respectively. For positive numbers, Equation (2.22) can be applied to the bit vector directly, but for negative numbers, the bit vector must first be negated. This can be done using Equation (2.23) (for a generic B-bit integer) and involves reversing each bit and adding one.

$$(y_{B-1}\dots y_1y_0)_2 = (\bar{x}_{B-1}\dots \bar{x}_1\bar{x}_0)_2 + 1 \tag{2.23}$$

Numerical errors arise when there is not enough range and/or precision to exactly

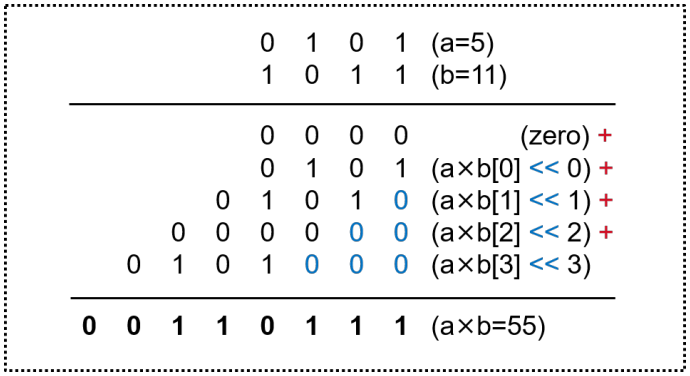


Figure 2.8: Unsigned integer multiplication ($N = 4$). The conventional algorithm requires N shift and add operations.

represent a given number. This can occur due to limitations in the fixed-point representation itself or from overflow and underflow in the arithmetic units. For a fixed-point representation in two's complement the range is given by $[-2^{B-F-1}, 2^{B-F-1} - 1]$ while precision is measured by the relative roundoff error $\epsilon = 2^{-F-1}$. The relative roundoff error is simply an expectation of the error from rounding or truncation. When the gap between successive numbers is 2^{-F}, rounding will produce errors that range between $[0, 2^{-F}]$. As an expectation this gives $\epsilon = 2^{-F-1}$. Overflow and underflow occur when the absolute value of a number is too large or too small for a given representation. In such cases, fixed-point representations simply require more range and precision respectively. Unfortunately, for a fixed number of bits, more range usually requires the sacrifice of precision and vice-versa. For this reason, other representations like floating-point are often preferred.

2.3.1.1 Hardware Implementation

In terms of hardware, fixed-point arithmetic (or integer arithmetic) has much lower complexity compared to floating-point. For addition and subtraction, efficient implementations are available using full-adders and carry chain logic, while for multiplication, modified Booth's algorithm with carry-save adders and reduction trees are known to produce fast and small multiplier circuits [43]. An example of unsigned multiplication is shown in Figure 2.8. Here, it takes N cycles to multiply two N-bit numbers using N shift and N add operations. Modern multipliers are much more efficient as they compute multiple bits of the product in parallel. In particular, FPGA architectures make use of such multiplier circuits in the DSP blocks (in Section 2.2.3). Together with dedicated carry chain logic in the CLBs, this makes FPGAs especially well suited to fixed-point arithmetic and multiply-add operations.

2.3.2 Floating-Point

The problem with fixed-point representations is that they often suffer from limited range and/or precision since more range can only be provided by sacrificing precision or vice-versa. On the other hand, floating-point representations have both a wide dynamic range and high precision and can accommodate both very large and very small numbers [43].

Floating-point representations have three distinct components: the sign S, exponent $E = (\hat{e}_{e-1}\hat{e}_{e-2}\ldots\hat{e}_1\hat{e}_0)_2$, and mantissa $M = (\hat{m}_{m-1}\hat{m}_{m-2}\ldots\hat{m}_1\hat{m}_0)_2$ in binary form. The exponent and mantissa bits, e and m, control the bandwidth, range and precision of the representation. Hence, floating-point formats are usually expressed as tuples e.g. (e, m).

$$\overbrace{(\ \hat{s}_0}^{S}\quad\overbrace{\hat{e}_{e-1}\hat{e}_{e-2}\ldots\hat{e}_1\hat{e}_0}^{E}\quad\overbrace{\hat{m}_{m-1}\hat{m}_{m-2}\ldots\hat{m}_1\hat{m}_0\ )_2}^{M}$$

Together, these three components represent the real number:

$$x = (-1)^S \times (1. + M) \times 2^E \tag{2.24}$$

- **Sign.** Floating-point numbers are mostly represented in signed magnitude form, where the sign is kept separate from the rest of the number. Other options include using two's complement to represent the mantissa component.
- **Mantissa.** The mantissa provides precision to the representation and operates as an m-bit unsigned fixed-point number with all bits fractional, i.e. $F = m$. Therefore the mantissa is also commonly called the *fraction*. Furthermore, the leading 1 bit in Equation (2.24) is implicit and describes floating-point in normalized form. Together, the mantissa therefore represents real numbers in the range $[1, 2)$ via $(1.\ \hat{m}_{m-1}\hat{m}_{m-2}\ldots\hat{m}_1\hat{m}_0)_2$. While normalized formats are the most common, unnormalized numbers can be realized using a leading 0 bit instead, the use of which provides more precision to numbers around zero.
- **Exponent.** The exponent is used to shift the mantissa and thus provides floating-point representations with a wide dynamic range. Although exponents operate as signed integers they are usually encoded as unsigned integers by adding a bias. A bias, $\beta = 2^{e-1} - 1$, incurs minimal overhead in hardware, and aligns the exponent so that $E = 0$ and $E = 2^e - 1$ represents the smallest (most negative) and largest (most positive) numbers. This helps with zero detection and error handling (e.g. infinity and NaN numbers).

The range of supported floating-point numbers is symmetrical around zero and comprise

intervals given by $[-min, -max]$ and $[min, max]$. Overflow and underflow occur when numbers fall outside of this range, i.e. when the absolute value is larger than max (overflow) or smaller than min (underflow).

$$max = \text{largest mantissa} \times 2^{\text{largest exponent}}$$
$$min = \text{smallest mantissa} \times 2^{\text{smallest exponent}}$$

Since max and min are often extremely large and small numbers respectively, the range is sometimes expressed in decibels (dB) using $20\log_{10}(max/min)$. In addition, floating-point precision, as measured by relative roundoff error, is given by $\epsilon = 2^{-m-1}$.

2.3.2.1 IEEE-754 Floating Point Standard

Table 2.4: Summary of IEEE-754 floating-point number formats

Scheme	Format (e, m) exponent	mantissa	Bias[3] (β)	Normal (max^+)	Denormal (min^+)	Range[1] (dB)	Precision[2] (ϵ)
FP16	5	10	-15	6.5e4	6.0e-8	241	2^{-11}
FP32	8	23	-127	3.4e38	1.4e-45	1668	2^{-24}
FP64	11	52	-1023	1.8e308	4.9e-324	12631	2^{-53}

[1] dynamic range in decibels $20\log_{10}(max^+/min^+)$
[2] relative round-off error, i.e. $2^{-m} \times 2^{-1}$
[3] exponent bias $\beta = -(2^{e-1} - 1)$

Given the flexibility of floating-point, and the number of specialized formats available, the Institute of Electrical and Electronics Engineers (IEEE) introduced the IEEE-754 standard for floating-point representations to promote interoperability across many hardware platforms and systems [44]. The standard defines rules around arithmetic and error handling, and specifies a set of industry approved floating-point formats which differ in numerical accuracy and hardware cost. The three most important formats are: half-precision (FP16), full-precision (FP32) and double-precision (FP64), which refer to 16-bit, 32-bit and 64-bit representations. A IEEE-754 floating-point number is formally given by Equation (2.25) below:

$$x = \begin{cases} E = 0, & (-1)^s \times 2^{1-\beta} \times (0 + M) \quad \text{(denormal)} \\ \text{otherwise}, & (-1)^s \times 2^{E-\beta} \times (1 + M) \quad \text{(normal)} \\ E = 2^e - 1, & \text{NaN or Infinity} \end{cases} \tag{2.25}$$

Compared to the standard format (given in Equation 2.24), IEEE-754 also represents the unnormalized floating-point region, also known as denormal numbers. Additionally, IEEE-

754 formats have biased exponent encodings and explicitly represent special numbers (e.g. NaN and Infinity), which are used for error signalling. Table 2.4 provides configuration details for different FP representations.

Although IEEE-754 formats are dominant for CPU and GPU platforms, designers of special-purpose machine learning systems may prefer alternate representations for performance or cost purposes. For example, Google TPU adopts a modified 16-bit floating-point format called BrainFloat (BFloat16) [45] which has 8 exponent and 7 mantissa bits. This presents a significant deviation from standard FP16 which only has 5 exponent bits. The extra exponent bits are deemed necessary for training DNNs specifically, which generally require a wide range for representing gradient numbers.

2.3.2.2 FP Addition and Multiplication

Machine learning algorithms typically require a large number of multiply-add operations for their computation. Next, we present floating-point arithmetic for multiplication and addition via example. In both examples, a and b represent normalized floating-point numbers according to Equation (2.24), except the sign bit and leading 1-bit are excluded for ease of notation. For FP multiplication, the result is calculated by:

$$\begin{aligned} a \times b &= (\pm M_a \times 2^{E_a}) \times (\pm M_b \times 2^{E_b}) \\ &= \pm (M_a \times M_b) \times 2^{(E_a + E_b)} \end{aligned}$$

Importantly, the mantissa and exponent can be calculated separately - the mantissa by standard fixed-point multiplication and the exponent by integer addition, as shown above. The only exception is when $2 \leq (M_a \times M_b) < 4$. In this case, the mantissa must be right-shifted 1-bit with one added to the exponent to compensate. This returns the result to normalized form as required, but comes at the cost of extra logic in hardware. FP addition is more complicated, and requires a right shift to align exponents before the operation:

$$\begin{aligned} a + b &= (\pm M_a \times 2^{E_a}) + (\pm M_b \times 2^{E_b}) \\ &= (\pm M_a \times 2^{E_a}) + (\pm \frac{M_b}{2^{E_a - E_b}} \times 2^{E_a}) \\ &= \pm (M_a + \frac{M_b}{2^{E_a - E_b}}) \times 2^{E_a} \end{aligned}$$

Here, we assume that $E_a \geq E_b$ and therefore M_b is right-shifted by $2^{E_a - E_b}$ bits to align the mantissas. With the mantissas aligned, addition becomes simple using standard fixed-point adders. Like multiplication above, the adder result might need to be converted back into

Table 2.5: Comparison of hardware cost for fixed-point and floating-point arithmetic. Energy and area numbers come from (Horowitz, 2014) [17] and (Dally, 2015) [18]

Operation	Area[1] (μm^2)	Energy[2] (pJ)
INT8 Add	36	0.03
INT16 Add	67	0.05
INT32 Add	137	0.1
FP16 Add	1360	0.4
FP32 Add	4184	0.9
INT8 Mul	282	0.2
INT32 Mul	3495	3.1
FP16 Mul	1640	1.1
FP32 Mul	7700	3.7

normalised form. This either requires a simple 1-bit right shift, when the sign of a and b are equal, or a large left shift, when the signs are different and the result is very close to zero. This is called the normalization shift (or postshift) as opposed to the alignment (or preshift) which happens before the addition. Basically, these shifters (together with associated sign and control logic) contribute most of the difference between floating-point and fixed-point arithmetic units in hardware.

2.3.2.3 Hardware Implementation

Floating-point arithmetic requires extra circuitry to handle signs, preshift, postshift, rounding, and special numbers (e.g. denormals, Infinity, NaN, zero). Hence, floating-point units are larger in area and consume more energy than equivalent fixed-point units. This is shown in Table 2.5 for addition and multiplication under a 45nm technology node.

Notably, there is a particularly large disparity between integer and floating-point adders, which is mainly due to preshift logic. For a 32-bit adder, the preshift can in principle be implemented using 32-to-1 multiplexers (32× of them) in a single stage. However, due to problems associated with high fan-in and fan-out, larger multistage designs are usually preferred. The high cost of FP addition is only exacerbated on FPGAs, which for the most part, do not have dedicated shifting logic and instead map multiplexers to CLBs which are highly inefficient. In fact, FP32 multiply-add units require an additional 705 LUTs and 1092 FFs on a Xilinx Ultrascale+ FPGA platform [2]. For this reason, Xilinx FPGAs are mainly used for computationally intensive fixed-point applications, e.g. machine learning inference. On the otherhand, Intel FPGAs, such as the Arria10 and Stratix10 device families, have dedicated floating-point units and are thus more amenable to training

[2]Target platform is *Xilinx Ultrascale+ FPGA xczu9eg ffvb1156* [46]

workloads.

2.3.3 Block Floating-Point

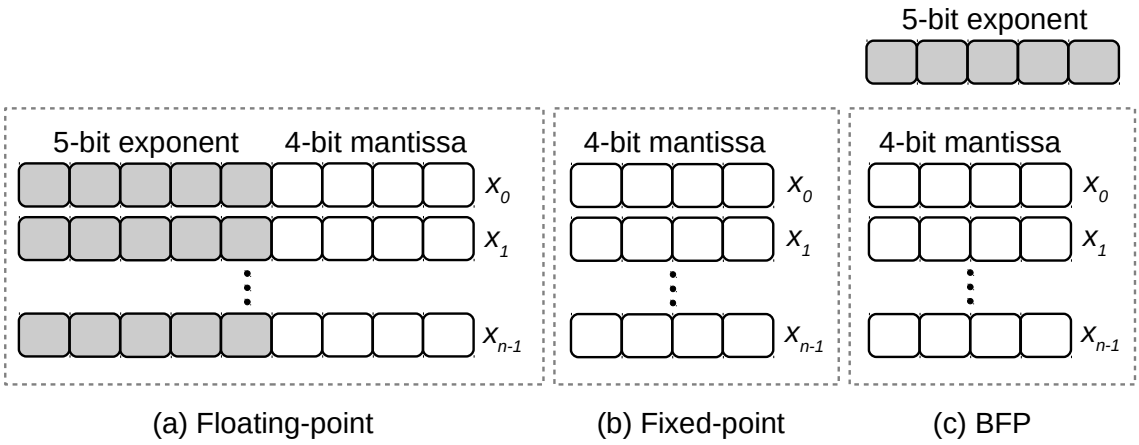


Figure 2.9: Floating-point, Fixed-point and Block floating-point (BFP) tensor representations. BFP tensors share an exponent across multiple elements.

Block floating-point (BFP) [47] is an alternative number representation that combines the best aspects of floating-point and fixed-point representations. In BFP, floating-point exponents are shared across blocks of N fixed-point numbers, providing a wide dynamic range to the block while ensuring that the majority of multiply-add operations can be implemented with dense fixed-point logic. Figure 2.9 provides an illustration.

Let $\mathbf{a} = [a_0, a_1, \ldots, a_{N-1}]$ represent a block of N elements, then the real value of an element a_i with mantissa M_i and shared exponent E_a is given by:

$$a_i = (1 + M_i) \times 2^{E_a} \tag{2.26}$$

The shared exponent E_a is usually determined by the maximum exponent in the block to prevent large overflow errors. As such, quantization errors arise when there is not enough precision (i.e. not enough mantissa bits) to adequately capture the value distribution of the block. The number of mantissa bits and block size are thus crucial design decisions for BFP representations.

2.4 Summary

A brief background on machine learning, spatial hardware architectures, and computer arithmetic was presented in this chapter.

Two classes of machine learning algorithms were introduced for solving non-linear problems, namely kernel methods and deep neural networks. Kernel methods extract features based on properties of a kernel function while DNNs are free to learn any data representation. In their standard form, it was shown that both techniques resolve to computations requiring predominantly matrix multiplication.

Important properties of systolic arrays and FPGAs were outlined and discussed. Systolic arrays can be designed to effectively exploit parallelism in workloads involving matrix multiplication, providing substantial speedups over conventional processors. Meanwhile, FPGAs can be reconfigured to support different types of hardware architectures. Both concepts are leveraged in Chapter 3, where a highly specialized systolic array architecture is presented for accelerating kernel methods on FPGAs.

Lastly, fixed-point, floating-point and block floating-point arithmetic schemes were described and compared in terms of numerical precision, representable range and hardware cost. In Chapter 5, a specialized new number representation is introduced called Block Minifloat, which combines the best aspects of all three types of arithmetic, enabling DNN training with fewer bits and higher accuracy.

Chapter 3

Kernel Methods: FPGA Fastfood

This chapter describes a Field Programmable Gate Array (FPGA) implementation of the Fastfood algorithm for simultaneous prediction and training. A highly specialized systolic array architecture is designed which can operate at a high frequency and supports problems that are 10^3 times larger than previously reported kernel methods.

The presentation of this chapter is based on background given on kernel methods and systolic arrays in Chapter 2, and expands on work previously published in [48].

3.1 Introduction

Kernel methods are a popular class of machine learning algorithm capable of solving many problems, ranging from classification and regression to novelty detection and feature extraction. However, they are often limited to small datasets because their memory and computation requirements scale linearly with the number of input examples. To address this problem, Le et al. [33] proposed an efficient algorithm, called Fastfood, that builds an approximation using combinations of random diagonal matrices and Hadamard transforms. This is advantageous because diagonal matrices require little storage and only involve elementwise multiplications, and the Hadamard transform can be computed in log-linear time via the Fast Walsh Hadamard Transform (FWHT). These properties reduce computation and storage requirements by several orders of magnitude, allowing the use of large-scale kernel methods in embedded applications.

This chapter describes the first known implementation of Fastfood using Field Programmable Gate Arrays (FPGA). The design, given in Section 3.3, is a systolic array architecture that creates local structure by grouping processing elements into designated blocks. This gives the flexibility of compiling different configurations of Fastfood, and achieve a 500 MHz operating frequency. In fact, simply preserving module partitions dur-

ing synthesis is enough to sustain high clock rates with the design. This chapter highlights that Fastfood is particularly well suited for a hardware implementation because:

- the computation requires simple arithmetic operations involving mainly addition and subtraction
- memory requirements are minimal allowing large parameter spaces to be stored in embedded memory
- the algorithm can be translated into a datapath that has a regular structure, is highly parallelisable, and benefits from high-speed local interconnections
- the butterfly structure for computing the FWHT has an efficient hardware implementation

The Fastfood algorithm is a type of model compression for kernel methods, however, unlike techniques based on inducing reduced precision and sparsity [19, 49], Fastfood works during the training phase too. Hence, this chapter focuses on streaming applications and adaptive learning environments, and performs simultaneous prediction and training for regression in real-time. One of the key outcomes of this chapter is a machine learning architecture that supports modelling capacity similar to several layers of a deep neural network (DNN) but can also perform online training.

3.1.1 Related Work

Efficient FPGA implementations of kernel methods have previously been studied and several architectures for performing simultaneous prediction and training have been reported.

Fraser et al. [50] describe a floating-point implementation of the KNLMS algorithm. They achieve very high efficiency using a fully-pipelined architecture and time-multiplexing independent parameter sets. This removes the data dependency in the update equation which significantly improves overall throughput. However, multiple parameter sets and deep pipelining increase the latency on a per input basis.

Tridgell et al. [51] address this issue using a technique called “Braiding”, which resolves data dependencies in pipelined architectures. Their design (in fixed-point) operates at well over 10GbE line rates and achieve latencies around 10 cycles for small models, this coming at the cost of frequency and DSP resources.

Both of the above architectures are fully parallel and don’t explore the re-utilisation of resources for improving scalability. Pang et al. [52] describe a micro-coded vector processor for the SW-KRLS algorithm which does this. They also optimise for latency and support a sliding window with a software-like interface.

Each of the mentioned works targets a specific part of the FPGA design space, however, none can scale the input dimensionality and model size to support the type of big-data applications dominating the machine learning community today. A specialized implementation of Fastfood fills this void.

3.1.2 Contributions

The key contributions of this chapter are:

- The first FPGA implementation of a large-scale kernel method. The reported design can solve problems with an input dimensionality up to 3 orders of magnitude larger than previous FPGA implementations of kernel methods, and achieves 245× speed up over a single-core Central Processing Unit (CPU).
- A novel hierarchical systolic architecture for sustaining high clock rates on FPGAs. Blocks of processing elements constrain the majority of computation to small physical spaces and allow resource reuse. In particular, the design efficiently implements the computational bottleneck (FWHT) using the same processing elements that compute the rest of the Fastfood algorithm.

The chapter is organised as follows: Section 3.2 introduces the theory for Fastfood kernel methods. Section 3.3 describes the design and scalability of the architecture; results are discussed in Section 3.4; and lastly, Section 3.5 provides a summary.

3.2 Fastfood Algorithm

The Fastfood algorithm was introduced by Le et al. [33] to reduce the computational complexity of random kitchen sinks, which was previously introduced in Section 2.1.2. The main idea is that a random projection, given by $\mathbf{W}^T\mathbf{x}_i$ in Equation (2.15), can be approximated with a projection $\mathbf{V}\mathbf{x}_i$, as described by Equation (3.1), which requires much fewer operations to compute. Note that this section is based on background provided in Section 2.1.2 on kernel methods and random kitchen sinks. [1] As such, $\mathbf{W} \in \mathbb{R}^{d\times n}$ is a matrix consisting of n random basis functions, $\mathbf{x}_i \in \mathbb{R}^d$ represents an input vector with dimensionality d, and $y_i \in \mathbb{R}$ represents the corresponding output value for regression.

$$\mathbf{V}\mathbf{x}_i = [\mathbf{V}_1\mathbf{x}_i, ..., \mathbf{V}_h\mathbf{x}_i], \text{ where } \mathbf{V}_q\mathbf{x}_i = \mathbf{S}_q\mathbf{H}\mathbf{G}_q\mathbf{P}\mathbf{H}\mathbf{B}_q\mathbf{x}_i \tag{3.1}$$

To compute $\mathbf{V}\mathbf{x}_i$, we first break the computation down into $h = \lceil n/d \rceil$ separate stages, each working with d basis functions. By working from right to left, each intermediate computation of $\mathbf{V}_q\mathbf{x}_i$ can be completed by a sequence of matrix-vector operations, without ever computing or storing the matrix $\mathbf{V}_q$. Importantly, the matrix-vector operations are designed to involve minimal computation. Firstly, $\mathbf{B}$, $\mathbf{G}$ and $\mathbf{S}$ are $d \times d$ *diagonal* matrices, where $\mathbf{B}_{ii}$ and $\mathbf{G}_{ii}$ are drawn i.i.d. from $\{-1, 1\}$ and $N(0, 1)$ distributions respectively, whilst $\mathbf{S}_{ii}$ depends on the choice of kernel function. This work assumes a Radial Basis Function (RBF) kernel, also known as the Gaussian kernel, meaning $\mathbf{S}_{ii}$ is drawn from a chi-squared distribution [33]. Secondly, $\mathbf{P} \in \{0, 1\}^{d\times d}$ is a permutation matrix which can be efficiently implemented in hardware. Finally, $\mathbf{H} \in \{-1, 1\}^{d\times d}$ is a Hadamard matrix, as defined by Equation (3.2). Matrix-vector multiplication with a Hadamard matrix can be done efficiently using the Fast Walsh Hadamard Transform (FWHT). For example, using the radix-2 FWHT algorithm (given in Appendix A.2), this reduces the number of operations from d^2 to $d \log_2 d$. Note that the Hadamard matrix as described by Equation (3.2) requires the input dimension, d, to be a power of 2 ($d = 2^l$, where $l \in \mathbb{N}$). If this is not possible, the input vectors are padded with zeros to ensure this condition is met.

$$\mathbf{H} = \mathbf{H}_d = \begin{bmatrix} \mathbf{H}_{d/2} & \mathbf{H}_{d/2} \\ \mathbf{H}_{d/2} & -\mathbf{H}_{d/2} \end{bmatrix} \textit{ where } \mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.2}$$

Same as random kitchen sinks, the final feature representation of Fastfood is formed by passing $\mathbf{V}\mathbf{x}_i \in \mathbb{R}^n$ through a sinusoidal function given by Equation (3.3). From here, a pre-

[1] Only the key computation steps are presented for Fastfood. Please refer to (Le et al., 2013) [33] for full details and proofs of convergence.

Dataset	m	d	Exact RBF	Fastfood RBF
Wine	4,080	11	0.819	0.740
Insurance	5,822	85	0.231	0.264
KEGG	51,686	27	n/a	17.818
Forest	522,910	52	n/a	0.840
CIFAR-10	50,000	3,072	n/a	0.624

Table 3.1: Comparison of the test error on 4 regression benchmarks from the UCI repository [53]. The top-1 classification accuracy, as a percentage, is also given for CIFAR-10. The results were taken from Le et al. [33] where m is the number of input examples and d is the dimensionality. Results are for the Radial Basis Function (RBF) kernel.

diction is made by taking a weighted sum of Fastfood features as shown by Equation (3.4).

$$\psi(\mathbf{x}_i) = \frac{1}{\sqrt{n}} \cos(\mathbf{V}\mathbf{x}_i) \tag{3.3}$$

$$f(\mathbf{x}_i) = \sum_{j=1}^{n} \alpha_j \psi(\mathbf{x}_i) \tag{3.4}$$

In this chapter, $f(\mathbf{x})$ is trained for regression by solving the least-squares problem.

$$\min_{\boldsymbol{\alpha}} \quad \left\| \boldsymbol{\alpha}^T \psi(\mathbf{x}) - y \right\|_2^2$$

More specifically, an online stochastic gradient descent (SGD) algorithm is applied, which incrementally updates $\boldsymbol{\alpha}$ for each new $(\mathbf{x}_i, y_i)$. The exact computation required is given by Equation (3.5). Alternatively, a batch-based training method could be used, as described in Algorithm 1 (Section 2.1.1). However, this requires prior access to a larger subset of the input data which is not available in real-time applications.

$$\boldsymbol{\alpha}_{t+1} = \boldsymbol{\alpha}_t - \eta(\boldsymbol{\alpha}_t^T \psi(\mathbf{x}_t) - y_t)\psi(\mathbf{x}_t) \tag{3.5}$$

To summarise the benefits: Fastfood reduces $O(nd)$ storage and computation requirements of random kitchen sinks to $O(n)$ and $O(n \log_2 d)$ respectively. The statistical properties of the kernel function are preserved, there is minimal degradation in prediction accuracy, and Fastfood can now solve problems that were previously intractable for exact kernel methods, as shown in Table 3.1. As with random kitchen sinks, when $n = 16,384$, Fastfood achieves an accuracy on CIFAR-10 [32] which is among the top two for shift-invariant kernel representations [33].

3.2.1 Comparison with DNNs

Approximate kernel methods based on random projections strongly resemble the matrix computations at the heart of DNN layers with multiple neurons. That is, they both compute a matrix multiplication and apply a non-linearity function to the result. Therefore, fast kernel methods can be used in place of DNN layers to speed up and compress DNNs [54–56]. Perhaps even more important than the computational benefits of Fastfood-type transformations, kernel methods are preferred to DNNs because they allow domain knowledge to be imparted in a statistical manner. The kernel function is used to describe the covariance in the input space, thus carrying the ability to shape and control the features extracted. In contrast, DNNs have powerful generalisation properties because they are free to learn any data representation. However, this obfuscates the training process and yields features that are less interpretable.

Parameter	Description
n	Number of Basis Functions
d	Input Dimensionality
p	Number of PEs
$k = n/p$	Data per PE
$h = n/d$	Number of HBs
$b = p/h$	PEs per HB

Table 3.2: Architecture configuration parameters

3.3 Specialized Architecture and Design

In this section, a novel systolic array architecture is described for computing Fastfood. Suitable areas for optimisation are explored and the scalability of the design is discussed.

3.3.1 High-Level Description

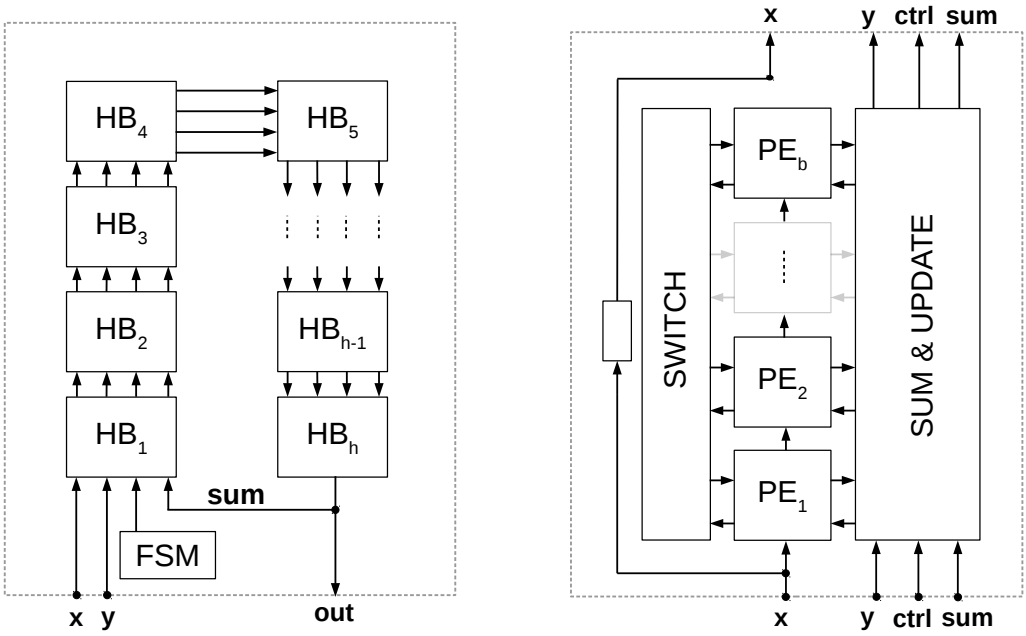
As described in the previous section, Fastfood is computed by concatenating multiple evaluations of $\mathbf{V}_q\mathbf{x}_i$, passing the result through a non-linearity, and calculating a weighted sum (Equations (3.1), (3.3), (3.4) in Section 3.2). Given that each $\mathbf{V}_q\mathbf{x}_i$ can be processed independently, the architecture separates and localises their computation using blocks of Processing Elements (PE), called Hadamard Blocks (HB). A hierarchical diagram of the Fastfood processor is shown in Figure 3.1, and the following equation shows how the computation of $f(\mathbf{x})$ (in Equation 3.4) is unrolled by this design:

$$f(x) = \sum_{j=1}^{n} \alpha_j \psi(x) = \sum^{h} \sum^{b} \sum_{j=1}^{k} \alpha_j \psi(x) \tag{3.6}$$

Here, n is the number of basis functions or expansion dimensions, k is the basis functions per PE, b is the PEs per HB, and h is the number of HBs. The other constraints are the input dimensionality, d, and total number of PEs, p. This information is summarized in Table 3.2.

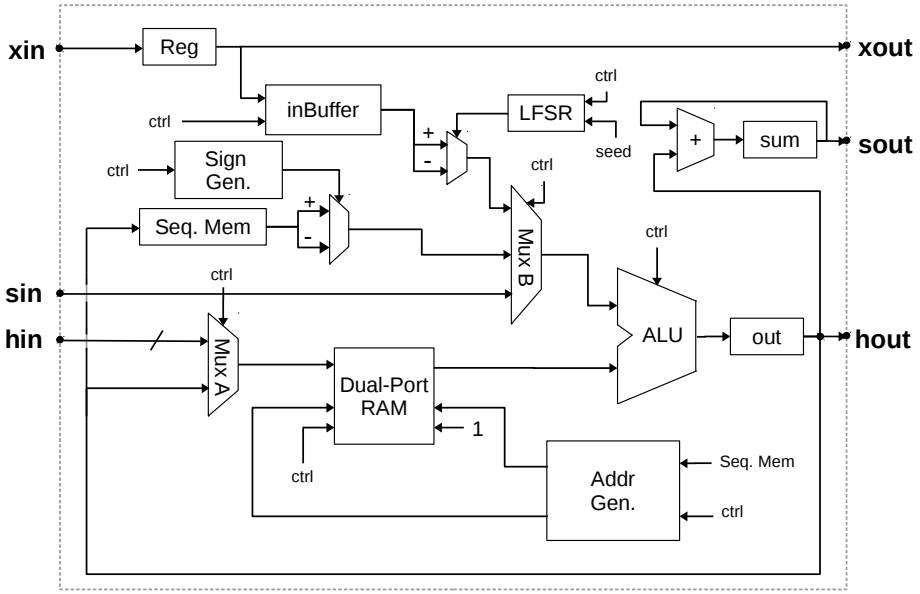
This architecture has been selected because 1) massive scalability can be achieved when PE compute resources are reused, 2) PEs with mostly local connections are more amenable to high-frequency implementations, and 3) blocks of PEs create the local connectivity needed to implement a fused and distributed Fast Walsh Hadamard Transform (FWHT). The last point gives this architecture some unique characteristics in the trade-off between resources, parallelism and frequency, and is discussed more in Section 3.3.3.

Parameters p, b and k should be chosen carefully, and depend on the FPGA device, task,



(a) Top-level

(b) Hadamard Block (HB)



(c) Processing Element (PE)

Figure 3.1: A hierarchical block diagram of the Fastfood processor

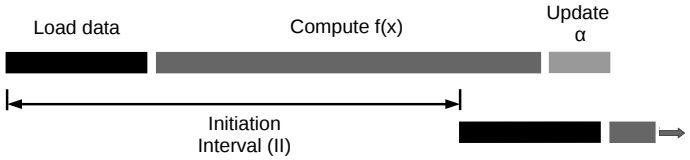


Figure 3.2: Schedule for computing Fastfood - loading and computation is overlapped

and performance requirements:

- p controls the parallelism and latency of the design
- k is a portion of the algorithm distributed evenly across the PEs. k controls the number of basis functions n and scales the memory requirements of each PE
- b is the number of PEs allocated to each HB. b controls the locality of connections between PEs, and in combination with k, also determines the input dimensionality d that can be supported

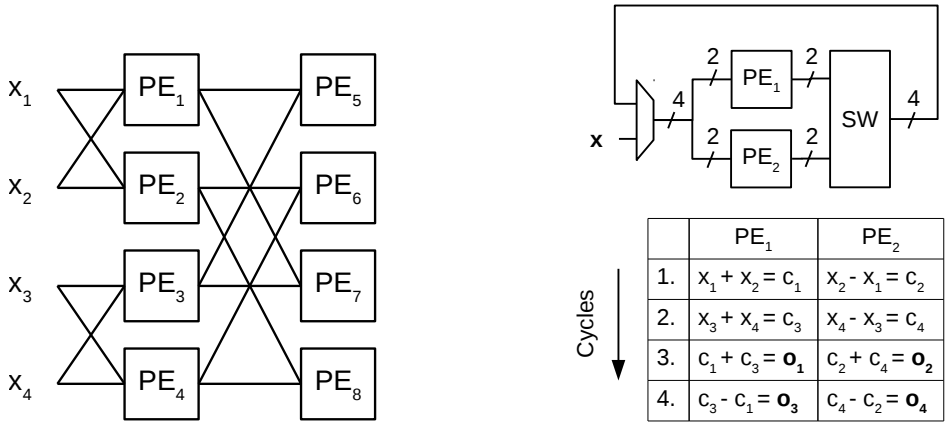
The remainder of this section provides a more detailed description and analysis for each of the above-mentioned design considerations.

3.3.2 Top-Level Module

The top-level diagram of the architecture is illustrated in Figure 3.1a (ignoring PCI-e and memory bus interfaces). It mainly consists of an h-length array of HBs connected in a ring topology. Each HB accepts control logic from a finite state machine (FSM) and a stream of $(\mathbf{x}_i, y_i)$ pairs as input data. The Fastfood processor is an iterative architecture that does not have overlapping compute stages. This means that the inputs arrive with an initiation interval (II) equal to the number of cycles taken to compute a given sized input vector. This is observed in Figure 3.2. Each HB computes a partial result for Fastfood, and an output is produced by summing each partial result as per the outer loop of Equation (3.6). The HBs are connected in a ring array for two main reasons: 1.) to minimise routing between HBs for computing the outermost accumulation, and 2.) to efficiently broadcast the input and FSM control logic across a large area of the chip.

3.3.3 Hadamard Block

Each HB contains an array of b PEs, a switch, and a module for computing the HB sum and partial update. This is shown in Figure 3.1b. HBs are created to develop local structures between PEs for implementing the FWHT at high clock rates. A special emphasis is placed



(a) no iterative reuse **(b)** full iterative reuse

Figure 3.3: Two systolic array configurations for a 4-point FWHT. This is based on the "Spiral" hardware generation framework described by Milder et al. [57].

on the FWHT because its parallel implementation has data dependencies that require communication between multiple PEs. Below, design considerations for the FWHT and *Sum & Update* unit are discussed for maximising the frequency and scalability of Fastfood.

3.3.3.1 FWHT

The standard architecture of a radix-2 4-point FWHT is the fully pipelined version given in Figure 3.3a. It is constructed by cascading $\log_2 d$ stages of d PEs and connecting them in a butterfly structure to directly implement the required switching behaviour. This design executes the FWHT in a minimum number of cycles but requires large amounts of additional hardware that are not reused. Given that the rest of Fastfood can be computed using a linear systolic array, the d-point FWHT is implemented by reusing one stage of b PEs over multiple cycles, like Figure 3.3b. This design has a compact and regular structure, which makes it well suited to processing large amounts of data at a high frequency. The only overhead is the switching network which is used to send intermediate results between $\log_2 b$ PEs. The switch is implemented using a multiplexer in each PE, and connecting the PEs directly. For increasing b, the switch size also increases. Experiments were conducted with $b = \{4, 8, 16, 32\}$, above which degradations in frequency were observed due to routing congestion.

3.3.3.2 Sum & Update

The *Sum & Update* unit routes the FSM control logic and also includes logic to implement several instructions not allocated to the PEs. Figure 3.4 gives a dataflow diagram of

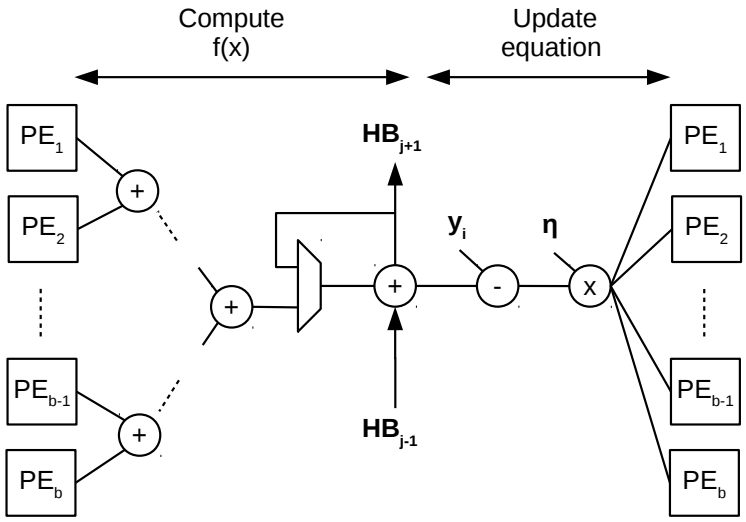


Figure 3.4: Dataflow diagram of the HB sum and update module (data moves from left to right)

the operations involved. The design uses $\log_2 b$ adders to compute the middle loop of Equation (3.6). Each HB requires the results from every other HB to know the update (outer loop of Equation 3.6). These are passed around the ring array and accumulated here. The *Sum & Update* module then includes one subtract and multiply unit for computing part of the update, Equation (3.5). The result is written back to each PE where the weights are ultimately updated. This creates a small resource inefficiency and means the PEs are idle for a short period of time. However, it only equates to an extra 1×DSP per HB and between 2-5% in LUTs and FFs, while the PEs are idle for only 1% of total processing time.

3.3.4 Processing Element

The basic structure of each PE is given in Figure 3.1c. It consists of one Arithmetic Logic Unit (ALU), two scalar operands, and a control mechanism for reading and writing to memory. Computation can only begin once a full input vector has been loaded into PE memory and the previous computation is finished. The *inBuffer* shift register is used to buffer the input and allow the loading and computation stages to be overlapped. Below, other important elements of the Fastfood PE are discussed:

3.3.4.1 Functionality

Table 3.3 summarises the compute requirements of each PE as a list of instructions. As per the table, not all the functionality is implemented with resources confined to the ALU.

Instruction	ALU	Active Cycles
Processing Element (PE)		
Mul B	N	k
Mul	Y	$3k$
Permutation	N	k
Add	Y	$2k \log_2 d + k$
Cosine	N	k
Mul-Add	Y	$k + 1$
Hadamard Block (HB)		
Adder-tree	-	$log_2 b$
Add	-	h
Sub ($y - f(\mathbf{x})$)	-	1
Mul (η)	-	1
IO Interface		
Load	-	d

Table 3.3: Fastfood instruction count for resources constrained in each PE and HB. The ALU column denotes whether a PE instruction is mapped to the ALU (Y/N)

For instance, the first Fastfood operation is a binary multiplication involving *inBuffer* and a random variable $B \in \{-1, 1\}$ (in Equation 3.1). This was implemented as a sign change using 1-bit from a 16-bit Linear Feedback Shift Register (LFSR). Test results (in Figure 3.6) showed that 16-bits were enough for the Fastfood algorithm. The Mul and Add instructions are implemented using 1×DSP slice and 1×adder in the ALU, and for Mul-Add an additional adder is used for the accumulation stage (i.e. *sum*). On a Xilinx Ultrascale device the two adders amount to approximately 8 CLBs. This overhead can be avoided by configuring the DSP slice to also do the Add and Mul-Add instructions, but this optimisation has been deferred for future work. The majority of Fastfood compute time is spent iteratively reusing the ALU adder. In fact, an average of approximately 70% of total cycles is dedicated to Adds. The cosine function is implemented using a 256-point look-up table stored in a shared *Dual-Port RAM*. The two ports are normally used for simultaneous reads/writes but for the cosine transformation, only one port is required. A result is read, i.e. $cos(x)$, using an address generated from x, where x is an intermediate result from the *Seq. Mem* shift register. Only 8-bits of x are used for the cosine look-up table since 8-bits are enough to sustain the accuracy of the function and also reduce the memory requirements.

3.3.4.2 FWHT

The FWHT involves $2k \log_2 d$ add and subtract operations per PE. To implement this, the ALU adder is reused over multiple stages and sign switching is performed on the top operand using the *Sign Generator* block. The *hin* and *hout* I/O ports connect multiple PEs, and *MuxA* switches between them. For local permutations (or switching), the *Address Generator* develops the memory access patterns required. The size of *MuxA* scales as $O(\log_2 b)$ which for $b = \{4, 8, 16, 32\}$ can be implemented efficiently with one layer of LUTs. The address and sign permutations are the same for each PE and can be generated using several counters and bit operations. This only contributes an additional 30 LUTS and 26 FFs for the *Address Generator* and *Sign Generator* modules combined. Given the small overhead, these control units are included in every PE. This reduces the compute density of each PE but removes a large fan-out problem which would likely require multiple stages of pipeline registers.

3.3.4.3 Memory

Fastfood involves a data transformation from $\mathbb{R}^d \rightarrow \mathbb{R}^n$, where $n > d$. The hardware architecture distributes this computation across p PE's so that each PE works on $k = n/p$ dimensions, also known as basis functions. This requires k-length blocks of memory for intermediate results and Fastfood parameters $\mathbf{G}$, $\mathbf{S}$ and $\boldsymbol{\alpha}$. Intermediate results are written back to *Sequential Memory* every cycle, and Fastfood parameters are generated at compile time and loaded to the *Dual-Port RAM* once at startup. The results of $\psi(x)$ (in Equation 3.3) are also saved in *Dual-Port RAM*, and for the FWHT switch, data is read from and written to a double buffer in *Dual-Port RAM* to avoid pipeline stalls. Therefore, including the *inBuffer* and 256-point cosine look-up table, each PE requires $(k + k + 3k + 2k + k + 256) \times bitwidth$ of memory. The *Dual-Port RAM* is either one or two 18Kb BRAMs depending on the size of k, and the two sequential memories are either implemented using two LUT-based shift registers (i.e. SRL) or the *Seq. Mem* shift register is mapped to a BRAM and *inBuffer* remains an SRL. In Section 3.4, results are presented for both configurations. The choice depends on whether a LUT-constrained or BRAM-constrained design is preferred.

3.3.4.4 Pipelining

The PEs are fully pipelined. There are 4 register stages for reading and writing to memory, between 3 and 6 on the ALU, 1 on the ALU operands, 1 on the output, and 1 on the *MuxB* inputs. This minimises the number of logic levels between registered signals and keeps the

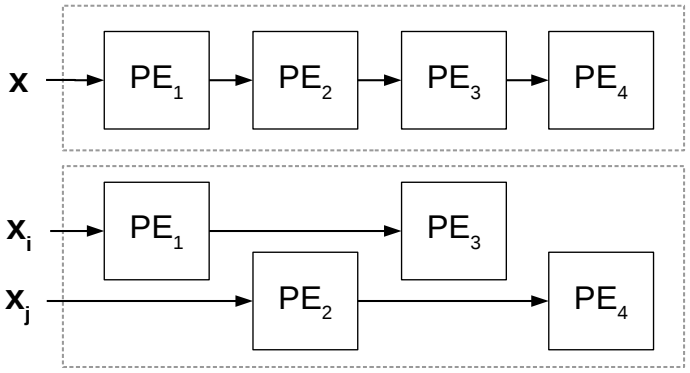


Figure 3.5: Existing input data path (top) and a proposed double data path (bottom) if greater I/O bandwidth is required

frequency high.

3.3.5 Scalability: I/O and Latency

The top-level module takes 1× 18-bit feature as input per cycle. The data is loaded through the array via a d-length shift-register, where d is the input dimensionality. Given that load and compute stages operate in parallel, one result can be retrieved every tc cycles, where tc is the total number of compute cycles given below:

$$tc = 8k + 2k \log_2 d + \log_2 b + 2h + 9 \tag{3.7}$$

The above equation is taken from Table 3.3 for prediction and training, except a few extra cycles are added because the HB control unit is not fully pipelined. The latency is thus $tc + d$ and the I/O required is $18d/8$ bytes every $max(d, tc)$ cycles.

For large d and $d > tc$, the throughput is limited by loading the input data. This equates to 18-bits per cycle, or 9Gb/s, assuming the memory bandwidth is not the bottleneck. This can be improved by broadcasting more of the input data in parallel. Figure 3.5 gives an example of how this can be achieved using two routing paths, one for even and odd-numbered PEs.

3.3.6 Trade-offs and Constraints

For a given number of basis functions, n, and input dimensionality, d, the choice of architectural parameters p, k and b is a trade-off between problem size and latency:

	LUT Total (203k)	LUT Mem. (112k)	FFs (406k)	BRAM (1080)	DSP (1700)	Max. PEs (p)
			FF-L			
k=64	323	98	524	1	1	610
k=128	398	170	531	1	1	494
k=256	547	314	540	2	1	355
			FF-B			
k=64	282	70	590	2	1	540
k=128	327	107	603	2	1	540
k=256	405	180	615	3	1	360

Table 3.4: PE resource utilisation for LUT and BRAM constrained 18-bit designs targeting a Xilinx Kintex XCU035 FPGA

3.3.6.1 Problem Size

The problem size is defined as any combination of n and d which does not exceed on-chip memory nor violate any design constraints in Table 3.2. For a large k the hardware architecture can support problems having large n and d, and when b is also large, even greater d can be supported. However, an increase in k also increases the computation and memory required for the PEs. This yields designs with higher latency or greater resource requirements for the same latency. In addition, by increasing b, the routing resources within the HBs are also increased. This allows larger switch sizes to be implemented but tends to diminish the operating frequency.

3.3.6.2 Latency/Parallelism

For a given d, the number of compute cycles in Equation (3.7) is mainly controlled by k. So for faster designs, p should be large so a small k can be distributed to each PE. Furthermore, k and b must both be powers of two, and for a fixed n, this means that p will also scale exponentially. This may result in resource utilisation which only slightly exceeds 50% in some cases. For this reason, Section 3.4 presents results for two designs. One which is LUT-constrained and the other which is BRAM-constrained. For brevity, these designs are referred to as "FF-L" and "FF-B" respectively.

3.4 Results and Evaluation

This section evaluates the specialized Fastfood architecture in terms of problem size and clock frequency. The designs were written in Chisel HDL [58] and were synthesised

	Input dim. (d)	Basis func. (n)
FF-B (p=352)	8192	90.1K
FF-L (p=320)	8192	81.9K
	Input dim. (d)	Dict. size (N)
KNLMS (Fraser '15 [50])	8	16
NORMA (Tridgell '16 [51])	8	200
KRLS (Pang '16 [52])	8	200

Table 3.5: Peak problem size for LUT and BRAM constrained 18-bit designs targeting a Xilinx Kintex XCU035 FPGA

and implemented using Xilinx Vivado 2017.2 targeting a Kintex Ultrascale XCKU035-FBVA676-2-e FPGA. Chisel is a hardware description language embedded in Scala which generates verilog. The use of Scala allows complex and parameterised structures to be expressed easily by taking advantage of the power of modern programming such as data structures and inheritance.

3.4.1 Resource Utilisation

Table 3.4 shows resource utilisation of LUT (FF-L) and BRAM (FF-B) constrained PEs for an 18-bit design. The area scales with k because of two k-length shift-registers, *inBuffer* and *Seq. Mem*. In the FF-L design, both shift registers are mapped to LUT SRL primitives, and in the FF-B design, *Seq. Mem* is implemented with a BRAM. The last column shows the maximum number of PEs that can be placed on the target device. These numbers are based on 97% of total LUTs and all available BRAMs. The remaining 3% of LUTs are attributed to the FSM and HB modules, which contribute an additional 141 and h×617 LUTs respectively. The reported designs are restricted to three BRAMs per PE, one of which is only required for $k = 256$. Larger k can be supported, but this requires doubling memory resources since k scales as a power of two. In terms of parallelism, FF-L can support more PEs for $k \leq 64$, FF-B is preferred for $k = 128$, and both designs are approximately equal for $k = 256$.

3.4.2 Problem Size

Table 3.5 gives the peak problem size of Fastfood compared with other online kernel methods. This translates to 10^3 times increase in input dimensionality and modelling capacity up to $90.1K$ basis functions. The basis functions allow kernel methods to be

		Designs achieving Freq=500 MHz			For max. resources in Table 3.4	
		LUT %	BRAM %	PEs (p)	PEs (p)	Freq. (MHz)
k=64	b=32	91.6	50.4	544	576	483
	b=64	67.9	35.6	384	576	416
k=128	b=32	92.0	41.5	448	480	487
	b=64	54.8	23.7	256	448	465
k=256	b=32	89.2	59.3	320	320	508
	b=64	53.3	35.6	192	320	476

Table 3.6: Clock frequency for an 18-bit FF-L design - *Seq. Mem* implemented using a LUT SRL primitive

		Designs achieving Freq=500 MHz			For max. resources in Table 3.4	
		LUT %	BRAM %	PEs (p)	PEs (p)	Freq. (MHz)
k=64	b=32	82.1	94.8	512	512	501
	b=64	66.0	71.1	384	512	465
k=128	b=32	88.3	88.9	480	512	487
	b=64	59.7	59.3	320	512	455
k=256	b=32	76.6	97.8	352	352	501
	b=64	43.6	53.3	192	320	468

Table 3.7: Clock frequency for an 18-bit FF-B design - *Seq. Mem* implemented using a BRAM

used in problems with many input examples, m, because they approximate the kernel function for large dictionary sizes, N. As shown in Table 3.1, exact kernel methods such as KNLMS, KRLS and NORMA are intractable on the CIFAR-10 dataset because N is required to be very large. On the other hand, Fastfood with $n = 16,384$ basis functions achieves an accuracy in the top two for RBF kernel representations. This suggests that an architecture supporting $n = 90.1K$ basis functions can solve problems even larger than CIFAR-10, which has $m = 50,000$ images and $d = 3,072$ dimensions. The achieved input dimensionality and dictionary size in Table 3.5 are based on maximum values for k and b, and Table 3.4 is used for p. More specifically, d was calculated by multiplying $k = 256$ and $b = 32$, and n was obtained by multiplying $k = 256$ and $p = 352$, where p must be a multiple of b. Although both FF-L and FF-B designs have enough resources for $p = 352$, only FF-B could place the design effectively.

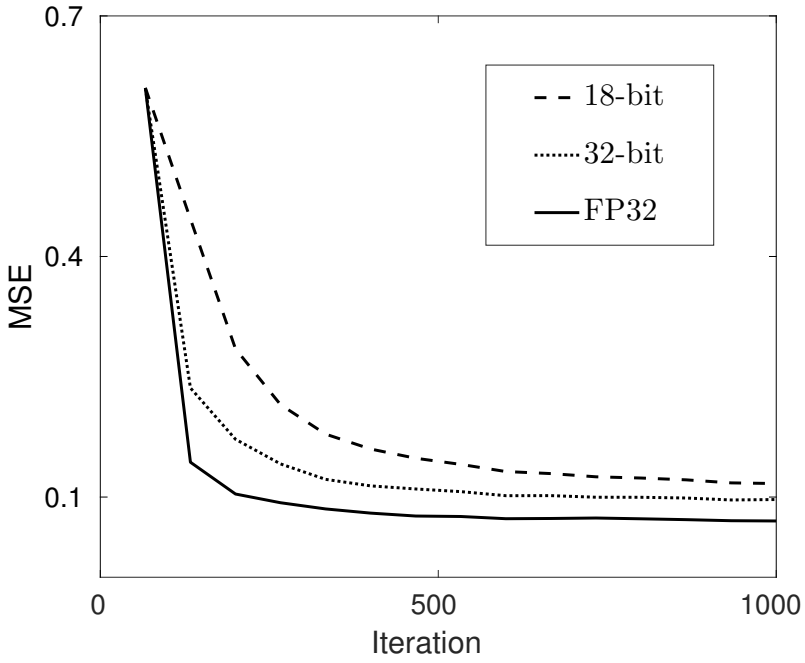


Figure 3.6: Accuracy results on a CPU: fixed-point v floating-point. A model with $n = 16,384$ and $d = 1,024$ were used for the Mackey Glass benchmark [59].

3.4.3 Clock Frequency

Table 3.6 and 3.7 show the effect that different design configurations have on the clock frequency for FF-L and FF-B designs. Importantly, both tables show that a high operating frequency can be achieved for designs very close to the resource constraints given in Table 3.4. For $b = 32$, clock rates at or above 500 MHz can be sustained right up to $p = 544$, $p = 448$ and $p = 320$ for FF-L, and up to $p = 512$, $p = 480$ and $p = 320$ for FF-B. This corresponds to utilisation rates between 89-98% of available resources. Furthermore, the frequency is only slightly reduced to 483-495 MHz for PEs which are even closer to the resource limit. Both tables also highlight the degradation in clock frequency observed for any $b > 32$. This occurs because 1.) routing congestion increases between the PEs and *Sum & Update* unit, and 2.) the size of each HB doubles in size which de-localises connections between them. Additional pipeline registers can be included to manage both these issues, but on large designs, this creates even more congestion between CLBs. Instead, an architecture is chosen where $b <= 32$, which guarantees an operating frequency close to 500 MHz for varying problem sizes. All designs were synthesised with cross-module LUT optimisation turned off. This preserves the local structure in the architecture precisely and performs up to 20% faster after place and route.

	Time (μs)	Speed-Up (×)
CPU [33]	580	1
FPGA (p=512)	2.4	245

Table 3.8: FPGA speed-up on a Xilinx Virtex XC7VX485-2 for $n = 16,384$ and $d = 1,024$

.

3.4.4 Speed-Up and Accuracy

Table 3.8 gives the speed-up of the FPGA implementation over a CPU. The execution time for the CPU version comes from the original Fastfood publication by Le et al. [33] and is only for prediction. The authors' code is written in C++ and uses the Spiral library for the FWHT. While details of the CPU are unclear from Le et al. [33], Spiral provides cache-friendly C/C++ code which is optimised using SSE vector instructions and up to four processor cores [60]. For a fair comparison, a previous generation Virtex 7 FPGA is chosen as the design target. The problem has $d = 1,024$ and $n = 16,384$, therefore, the implementation consists of 16 HBs, each of which compute a 1024-pt FWHT. For an FF-L design, the device operates at 432 MHz and has capacity for $p = 985$ PEs. However, only $p = 512$ could be used because p scales exponentially on fixed problem sizes. For $b = 32$ and $k = 32$, the number of compute cycles for prediction is $tc = 869$ (from Equation 3.7). Therefore, the design is I/O constrained (i.e. $tc < d$), and one result can only be obtained every d cycles. The final result is a 245× speed-up although only 52% of available resources are used.

The accuracy of Fastfood is given in Figure 3.6 for floating-point and fixed-point number representations. The mean square error (MSE) is plotted for the Mackey Glass regression benchmark [59] using a model consisting of $n = 16,384$ and $d = 1,024$. Results are based off one trial, where each trial refers to a new set of randomly sampled $\mathbf{G}$, $\mathbf{S}$ and $\mathbf{B}$ matrices. The graph shows that fixed-point is competitive with floating-point in terms of learning accuracy, although the convergence of 18-bit requires more training iterations. Specifically, FP32 converges after approximately 400 iterations, whereas 18-bit requires 900 iterations, corresponding to a 2.25× slowdown. Given a 245× speedup per iteration of the 18-bit FPGA implementation (over FP32 running on a CPU), the expected speedup in wall clock time of 18-bit over FP32 is 110×. Furthermore, this speedup will approach 245× as the latency associated with convergence gets hidden over time.

	d	n	Bit-width	Latency (cycles)	Fmax (MHz)	Time (ns)	Thr.put (Gb/s)
KNLMS (Virtex 7) [50]	8	16	32	207	314	3.18	80.4
NORMA (Virtex 7) [51]	8	200	18	10	127	7.87	18.3
KRLS (Stratix V) [52]	-	128	32	4396	157	28000	-
Fastfood (Kintex U)	8192	90.1 K	18	16932	508	17204	8.57
Fastfood (Virtex 7)	8192	98.3 K	18	16932	413	21162	6.97

Table 3.9: Comparison of Fastfood with other implementations of online kernel methods *Note: 1.) the input latency is included in Fastfood and KRLS but not in KNLMS and NORMA, 2.) n denotes both the number of basis functions and dictionary size, whereas, in the text, N denotes the dictionary size and is applied only in the context of KNLMS, NORMA and KRLS*

3.4.5 Analysis

The implementation of Fastfood occupies a unique part in the design space of online kernel methods. This is observed in Table 3.9 for designs targeting a Xilinx Kintex Ultrascale and Virtex XC7VX485-2. The latter is the same device used in [50] and [51]. A $p = 320$ and $p = 384$ were chosen for Kintex and Virtex devices respectively, while $b = 32$ and $k = 256$ were used for both. These configurations achieve an optimal combination of problem capacity and throughput for an 18-bit Fastfood implementation. This yields 3 orders of magnitude increase in input dimensionality, 8.57 Gb/s of throughput, and a large number of basis functions. The last point provides the ability to approximate much larger dictionary sizes than can otherwise be supported in [50–52]. Compared with random kitchen sinks that have $O(nd)$ memory complexity, Fastfood only requires storage for $3n$ parameters. This means that basis functions up to 1.54 GB (i.e. $n \times d \times 18/8$) can be approximated using only 0.58 MB, equating to a compression factor of 2655×. Future work will investigate how this result can be reinterpreted for DNNs, where state of the art compression factors are around 10× for full-precision weights [19]. The main difference being that basis functions are learned in DNNs, whereas in Fastfood, they are randomly sampled from a distribution that closely approximates a kernel function.

3.5 Summary

This chapter demonstrated the utility of employing the Fastfood algorithm for non-linear regression problems. Such an approach can be used to reduce the hardware requirements of kernel methods in applications demanding energy efficiency and real-time learning. A novel hierarchical systolic array architecture was described for minimising data transfers

between processing elements in the computation of Fastfood. This required an efficient implementation of the Fast Walsh Hadamard Transform (FWHT), the design of which is compatible with any fast transform, such as the Fast Fourier Transform (FFT). Based on synthesis, placement and routing simulations, the reported design can sustain 500 MHz clock rates while supporting problems with an input dimensionality that is 10^3 times larger than other online kernel methods. This paves the way for real-time and large-scale learning applications in control, communications and signal processing.

Chapter 4

Training Deep Neural Networks: SWALP

This chapter presents an algorithm and prototype hardware accelerator for training deep neural networks with high accuracy using low-precision arithmetic. Compared with conventional approaches, low-precision computation reduces both memory usage and computational cost, providing more scalability for Field Programmable Gate Arrays (FPGAs) with limited on-chip memory.

The presentation of this chapter is based on background given on deep neural networks and computer arithmetic in Chapter 2 and expands on work previously published in [61].

4.1 Introduction

Training deep neural networks (DNNs) requires large amounts of memory and computation, and until recently, has almost exclusively been performed with full-precision floating-point (FP32) numbers and many Graphics Processing Units (GPUs). As a result, it has proven difficult to move training into edge devices where there are stricter memory, power and computational constraints (e.g. mobile phones, radios, video cameras and satellites).

The primary workload in DNN training is general matrix multiplication (GEMM), which depending on the underlying hardware, could be either memory or compute bound. To overcome compute bound problems we want hardware with more logic and higher compute density, and for memory bound problems, it's desirable to fit at least one of the matrices in fast on-chip memory. Given that DNN model sizes are steadily increasing, and larger models produce larger matrices, more efficient accelerators are required to satisfy these demands. Low-precision computation offers arguably the best solution since fewer bits increases the compute density and reduces memory consumption.

The majority of work on low-precision DNNs has focused on the inference part only,

to speed up the run time. These low-precision network representations are obtained by quantising the weights and activations during training, and can normally achieve FP32 accuracy [19, 62]. In contrast, training with low-precision (e.g. 8-bit) generally produces worse accuracy than training with FP32. The main problem arises in computing gradients and accumulating the weight updates specifically [63, 64]. This is because the representable range of low-precision numbers is unsuitably small in most cases.

Several techniques have been developed to address this issue for 8-bit floating-point (FP8), such as wider accumulators [64], and having two different FP8 formats for forward and backward computations [65]. Fixed-point representations have also been explored [20, 22, 66, 67] since integer arithmetic offers higher performance density in hardware. However, fixed-point training suffers because of limited range and must be combined with other techniques to avoid convergence problems associated with overflow and underflow. For instance, the authors of SWALP [21] combine block floating-point quantisation with weight averaging, establishing the first training algorithm capable of achieving floating-point accuracy with predominantly fixed-point arithmetic and only 8-bit numbers. This chapter describes an FPGA implementation of SWALP, except some parts of the training algorithm are left in high-precision for greater flexibility (e.g. the weight gradients, and intermediate tensors).

The reported implementation targets the full range of Xilinx Zynq All Programmable System on Chip (APSoC) devices. Zynq comprises many different processing elements in the same package, offering real-time processing and massive integration capabilities for embedded applications. For instance, the ZCU111 board features an ARM processor and FPGA together with special multi-gigabit components for RF signal analysis. A training accelerator is designed especially for such systems, where we want to combine real-time integration with on-chip training. These applications are often referred to as *stand-alone* because they can necessarily be deployed in the field without host communication. In Section 4.5, this use case is demonstrated by training a DNN-based auto-encoder for anomaly detection in RF networks.

4.1.1 Related Work

The vast majority of work on low-precision has targeted hardware acceleration for DNN inference only [68]. Examples include FINN [69], Eyeriss [70], ESE [71], RebNet [72]. In contrast, there are relatively few examples for training DNNs on FPGAs.

Early work considered the case of backpropagation through just one neuron [73], and then one multi-layer perceptron (MLP) layer [74]. Small MLP networks are also considered in [75], except training is performed with 16-bits via weight perturbations. Such a technique can reduce the area by 10× over backpropagation when the network can

be fully pipelined. FPDeep [76] is a design framework for mapping DNN training across multiple FPGAs in a cluster. They show 3.4× better energy efficiency is possible training 16-bit AlexNet when the computation is distributed across 15 FPGAs in a pipelined manner. The authors in [22] describe an FPGA prototype which is similar in the sense that all weights and activations are stored on-chip. In general, the above-mentioned works are only relevant to single FPGA implementations if the networks are very small. F-CNN [77] is more scalable and uses run-time reconfiguration to implement the different kernels used during training. The computation is based on 32-bit floating-point arithmetic, although 8-bits is now sufficient. DarkFPGA [78] is a new accelerator for 8-bit training and is particularly optimised for convolution layers. The authors introduce several optimisations, including a new data pattern and tiling strategy, and demonstrate comparable performance with a GPU and better energy efficiency for training CIFAR-10. This chapter is similar in that an accelerator for 8-bit matrix multiplication has also been designed, but is sufficiently different in that this work focuses on techniques for high accuracy training and reports a real application of the technology.

4.1.2 Contributions

Although specialized low-precision training techniques like block floating-point and weight averaging have been described previously, to the best of the author's knowledge, this is the first demonstration of training with these techniques on Field Programmable Gate Arrays (FPGAs). Specifically, the contributions of this chapter are:

- The first FPGA implementation of a DNN training technique, which achieves the accuracy of floating-point using mostly 8-bit arithmetic.
- A modified version of SWALP which achieves improved accuracy, and a flexible hardware/software partitioning scheme.
- Quantitative analysis of the performance of the resulting system using the CIFAR-10 and MNIST benchmarks.
- The first demonstration of a real-time, radio frequency anomaly detector with FPGA-accelerated training.

The chapter is organised as follows: Section 4.2 introduces the low-precision training algorithm; Section 4.2 describes the design and implementation of the FPGA accelerator; training accuracy and hardware performance are evaluated in Section 4.4, and lastly, the chapter is summarized in Section 4.6.

4.2 Low-Precision Training Algorithm

In this section, a low-precision stochastic gradient descent (LP-SGD) algorithm is presented for training DNNs with high accuracy. Forward and backward computations are described as matrix multiplication for fully-connected and convolution layers, based on background provided in Section 2.1.3, and quantisation and averaging techniques are introduced for improving the stability of LP-SGD. Pseudocode for the algorithm is given in Algorithm 2, which crucially enables training with all weights, activations and activation gradients quantised to 8 bits only. Algorithm 2 is based on SWALP [21].

Algorithm 2: Training DNNs with LP-SGD + BFP + SWA

Define: Convolution layer k; time t; low-precision weight $\bar{W}_k$, activation $\bar{x}_k$, and activation gradient tensors $\bar{e}_k$; high-precision activation x_k and gradient tensors e_k ∇W_k; BFP quantisation functions Q_W, Q_x, Q_e; update function $U(\bar{W}_k, \nabla W_k)$; hardware accelerator function $gemm(A, B, E_A, E_B)$;learning rate η_t; softmax loss C; SWA weights W_{S_k}; SWA iterations S; SWA cycle c; Data batch sequence $\{a_t, b_t\}_{t=1}^{T}$;

{1. Initialise model:}

$\bar{W}_k^0, E_{W_k}^0 = Q_W(N(\mu, \sigma)), \forall k \in [1, L]$

$W_{S_k}^0 = N(\mu, \sigma), \forall k \in [1, L]; m = 0$

{2. Train model:}

for $t = 1, 2, \ldots, T$ **do**

{2.1 Forward propagation:}

$\bar{x}_0, E_{x_0} = Q_x(a_t)$

for $k = 1$ to L **do**

$x_k = ReLU\Big(gemm(\bar{W}_{k-1}, im2col(\bar{x}_{k-1}), E_{W_{k-1}}, E_{x_{k-1}})\Big)$

$\bar{x}_k, E_{x_k} = Q_x(x_k)$

end for

{2.2 Backward propagation:}

Compute $e_L = \frac{\partial C}{\partial \bar{x}_L}$ given $\bar{x}_L$ and b_t

$\bar{e}_L, E_{e_L} = Q_e(e_L)$

for $k = L$ to 1 **do**

$\nabla W_k = gemm\Big(im2col(\bar{x}_{k-1}), \bar{e}_k^T, E_{x_{k-1}}, E_{e_k}\Big)$

$e_{k-1} = col2im\Big(gemm(\bar{W}_k^T, \bar{e}_k, E_{W_k}, E_{e_k})\Big)$

$\bar{e}_{k-1}, E_{e_{k-1}} = Q_e(e_{k-1})$

end for

{2.3 Low-precision SGD Update:}

$\bar{W}_k^{t+1}, E_{W_k}^{t+1} = Q_W\Big(U(\bar{W}_k^t, \nabla {W_k}^t, \eta_t)\Big), \forall k \in [1, L]$

{2.4 High-precision SWA Update:}

if $t > S$ and $(t - S) \equiv 0 \pmod{c}$ **then**

$W_{S_k}^m = (W_{S_k}^m \cdot m + \bar{W}_k^t)/(m + 1), \forall k \in [1, L]$

$m = m + 1$

end if

end for

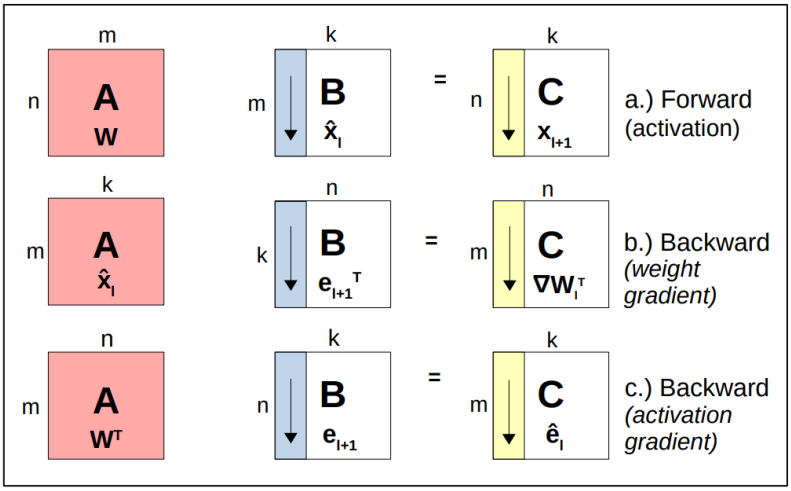


Figure 4.1: Array shapes and sizes for three matrix multiplications used for DNN training: A×B=C. Forward activation (top), backward weight gradient (middle) and backward activation gradient (bottom). A is cached in local memory, B and C are streamed in column-major order.

4.2.1 Forward and Backward Computation

Training typically requires three matrix multiplications at each layer of computation, as described in Section 2.1.3. Algorithm 2 implements general matrix multiplication as a function call, given by $C = gemm(A, B, E_A, E_B)$ in block floating-point (BFP), where A, B and C are shown in Figure 4.1. The forward computation for the activation, $\boldsymbol{x}_{l+1}$, and the backward computation for the activation gradient, $\boldsymbol{e}_l$, and weight gradient $\nabla \boldsymbol{W}_l$. The BFP shared exponents are given by E_A and E_B, for A and B respectively, and are used inside the gemm function call to convert back into floating-point. For hardware acceleration of matrix multiplication, it is desirable to store at least one of the matrices in fast on-chip memory. For this reason, matrix A (leftmost) is cached in on-chip memory while B and C are streamed in column-major order.

Table 4.1: Mapping convolution and fully-connected layers to matrix multiplication. IC, OC, k, W, H and B denote the number of input and output channels, size of convolution filter, width and height of input image, and batch size.

Layer	m	n	k
Convolution	$IC \times k_W \times k_H$	OC	$B \times W \times H$
Fully-connected	IC	OC	B

For convolution and fully-connected layers, Table 4.1 provides the mapping from the input dimensions to the matrix multiplier. In convolution layers, the inputs are typically a batch of images with dimensions (B, W, H, IC), corresponding to the batch size, width,

height and number of input channels. The outputs are computed by applying a convolution kernel to the image with typically four dimensions (k_W, k_H, IC, OC), which refer to the kernel width and height, and the number of input and output channels. The convolution kernel computes dot products over local regions of the input, moving over the input with an associated stride, and forming outputs with shape (B, W', H', OC), as described in Appendix A.3. Importantly, convolution layers can be reformulated as matrix multiplication by the *im2col* and *col2im* transforms. In Figure 4.1, $\hat{\boldsymbol{x}}_l$ and $\hat{\boldsymbol{e}}_l$ represent the data after *im2col* and before *col2im* respectively. In the next section, a prototype training accelerator is described which implements both transforms on the host CPU.

4.2.2 Block Floating-Point Quantisation

To train networks in low-precision, a quantisation function $Q(.)$ is required to convert real-valued weights, activations, and gradients into their rounded versions. A number of quantisation methods have been suggested in the literature [63, 64, 79, 80]. This chapter focuses on block floating-point quantisation with stochastic rounding, as implemented in [21].

Block floating point (BFP) is a hybrid number representation for balancing dynamic range and logic density requirements. In BFP, all numbers within a block share the same exponent, which is allowed to vary during training. The exponents are calculated at every iteration during training and are set to the largest exponent in a block to avoid overflow [22, 81]. Let $\mathbf{a} = [a_1, a_2, \ldots, a_N]$ represent a block of N real numbers, then the shared exponent, E_a, and the real value of a number a_i with mantissa x_i is given by Equations (4.1) and (4.2) respectively.

$$E_a = \max\left(\lfloor \log_2 |\mathbf{a}| \rfloor\right) \tag{4.1}$$

$$a_i = (1 + x_i) \times 2^{E_a} \tag{4.2}$$

For convolution layers, where the inputs are multi-dimensional arrays, a block could represent a batch, an image, or even just one channel. The finer the granularity, the more exponents that must be saved [21]. In this chapter, block assignments vary depending on whether the tensor is a weight or activation. For weights, one exponent is assigned to the tensor, and for activations, B exponents, are assigned to the layer, corresponding to the batch size. This is necessary because unlike weight tensors, the distribution of numbers within an activation tensor may not be well-calibrated across the entire batch of inputs.

Finally, stochastic rounding is used to quantize the mantissas, x_i, to a low-precision representation. In stochastic rounding, numbers are rounded up or down at random such

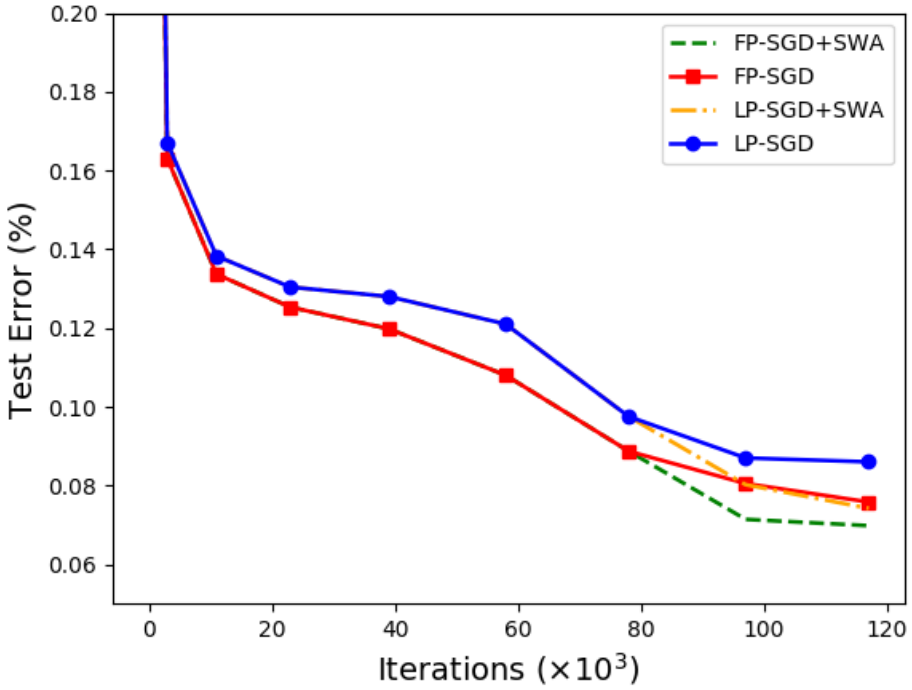


Figure 4.2: Empirical results of stochastic weight averaging (SWA) with full-precision (FP) and low-precision (LP) SGD

that the expected rounding error is zero, i.e. $\mathbb{E}[Q(x)] = x$. If each mantissa is represented by F bits, then the smallest representable mantissa is $\epsilon = 2^{-F}$ [79]. Let $\lfloor x \rfloor$ be the largest integer multiple of ϵ which is less than or equal to the real value of the mantissa x, then the quantisation function for x is:

$$Q(x) = \begin{cases} \lfloor x \rfloor & \text{with prob.} \quad 1 - \dfrac{(x - \lfloor x \rfloor)}{\epsilon} \\ \\ \lfloor x \rfloor + \epsilon & \text{with prob.} \quad \dfrac{x - \lfloor x \rfloor}{\epsilon} \end{cases}, \tag{4.3}$$

4.2.3 Stochastic Weight Averaging for LP-SGD (SWALP)

In SWALP [21], the authors showed that averaging SGD iterates of the weights with a higher learning rate can recover quantisation errors and produce better low-precision training models, even when all numbers are 8 bits. This technique has been incorporated into Algorithm 2 and is known as stochastic weight averaging (SWA) [82]. Figure 4.2 illustrates an example on the CIFAR-10 image classification dataset [32]. For the first $80k$ iterations standard SGD and a decaying learning rate is applied, and in the last $40k$ iterations the learning rate is modified, held constant, and the weights are averaged every epoch

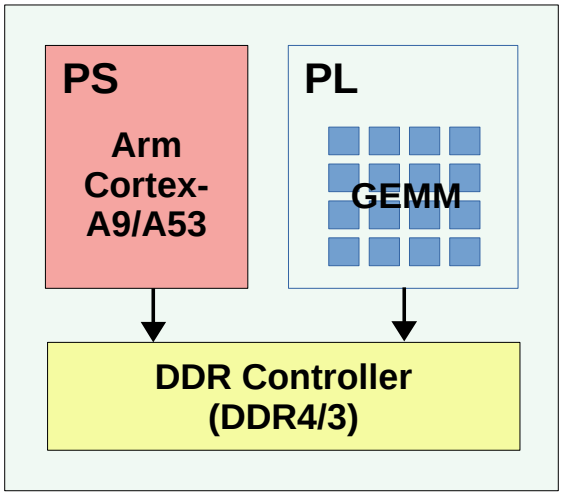


Figure 4.3: High-level system overview

(approximately 400 iterations). The averaged weights are not used directly in training but rather represent the final trained model. This technique is observed to work relatively better for low-precision models, as averaging cancels out errors in weights rounded up with those rounded down during quantisation. The main limitation from an implementation perspective is that a copy of the low-precision weights must be stored in high-precision for the moving average. For PCI-e based accelerator cards, the high-precision copy could be stored in slower host memory and the moving average calculated on the host every few hundred iterations by transferring the low-precision weights over the PCI-e bus. This way, the high-precision copy doesn't consume any of the fast DDR memory close to the accelerator. In embedded devices like Zynq, there is no host and the fast DDR memory is shared between the FPGA and ARM processor. So even though the high-precision copy is not used regularly, it will be saved with memory that could otherwise be used for more low-precision weights and supporting larger models.

4.3 Low-precision Training Accelerator

In this section, a low-precision DNN training accelerator is introduced for Zynq SoC and MPSoC devices. Figure 4.3 illustrates the three main components of the Zynq-based system, namely the programmable logic (PL) or FPGA, processing system (PS), and high bandwidth DRAM interface. A hardware/software partitioning scheme is proposed that offloads all GEMM operations to an 8-bit matrix multiplier on the FPGA, while the rest of the LP-SGD algorithm is computed in floating-point on an ARM processor. Since the majority of computation in any DNN involves matrix multiplication, significant speedups in training times can be achieved over processor only systems, without losing any generality

in the type of networks that can be supported. The only limitation is memory, both on the FPGA and in DRAM. The prototype is designed and implemented using Xilinx SDSoC 2018.3, a predominantly software-based development flow that abstracts away much of the low-level complexities associated with moving data between the PL and PS subsystems.

4.3.1 Software Overview

The software implementation is based on Darknet [83] with added support for stochastic weight averaging, low-precision convolution and fully-connected layers, contiguous memory allocation on Zynq, and compilation directives for SDSoC-based FPGA acceleration. Darknet is an open-source framework for DNNs written in C. It provides the backend for TinyYolo [84] and contains fast implementations for different layers, non-linearity functions, and DNN utility functions. A PYNQ v2.4 image is used to boot the target Zynq boards [85]. PYNQ runs desktop Ubuntu and offers a highly productive development environment, which includes C/C++ and Python libraries for interacting with custom IP on the FPGA. The use of these libraries means that the hardware accelerator is accessible from Python, which is the same abstraction level adopted by most DNN training frameworks, e.g. Tensorflow and PyTorch. All software (including bitstreams) can be installed from an online repository and compiled on the board. The repository is available at www.github.com/sfox14/darknet-zynq.

4.3.1.1 Hardware/Software Partitioning

Algorithm 2 is partitioned across FPGA and ARM processors as follows. All three *gemm* function calls are offloaded to the FPGA, corresponding to three 8-bit matrix multiplications given in Figure 4.1. Furthermore, the FPGA performs run-time transpose on matrix A (for computing the activation gradient) and handles integer to floating-point data conversions. All other operations are implemented with software on the ARM processor. This includes BFP quantization, stochastic weight averaging, matrix transpose, ReLU, and FP32 accumulators for weight gradients and weight updates. For convolution layers, *im2col* and *col2im* transforms are also performed in software. This comes at the cost of increased memory usage, as explained in Appendix A.3. Fully-connected layers are implemented similarly to convolution layers except *im2col* and *col2im* are not required.

4.3.1.2 Memory Usage

Each layer must store it's own set of 8-bit weights $\bar{\boldsymbol{W}}_k$, 8-bit input activations $\bar{\boldsymbol{x}}_k$, and BFP shared exponents (E_W, E_x). Given that LP-SGD operates over minibatches of the input

Table 4.2: VGG16 [86] Network Architecture (Convolution layers only). *Note, 2× refers to the number of times a layer is repeated. Each convolution layer has stride=1, and the input is downsampled using 2×2 maxpool layers.

#L.	Filter size ($k \times k \times IC \times OC$)	Input size ($W \times H \times IC$)
1	$3 \times 3 \times 3 \times 64$	$32 \times 32 \times 3$
2	$3 \times 3 \times 64 \times 64$	$32 \times 32 \times 64$
3	$3 \times 3 \times 64 \times 128$	$16 \times 16 \times 64$
4	$3 \times 3 \times 128 \times 128$	$16 \times 16 \times 128$
5	$3 \times 3 \times 128 \times 256$	$8 \times 8 \times 128$
6 (2×)	$3 \times 3 \times 256 \times 256$	$8 \times 8 \times 256$
7	$3 \times 3 \times 256 \times 512$	$4 \times 4 \times 256$
8 (2×)	$3 \times 3 \times 512 \times 512$	$4 \times 4 \times 512$
9 (3×)	$3 \times 3 \times 512 \times 512$	$2 \times 2 \times 512$
10 (2×)	$1 \times 1 \times 512 \times 512$	$1 \times 1 \times 512$

Table 4.3: Memory usage in megabytes (MB) for varying batch sizes on VGG16

Batch	Darknet (float)	8-bit			8-bit + SWA		
		malloc	cma	total	malloc	cma	total
1	141	4	34	38	65	34	99
32	276	110	38	148	171	38	209
128	722	438	51	489	499	51	550

data, the batch size scales the size of $\bar{\boldsymbol{x}}_k$ and subsequently the memory required for the entire network.

For training a VGG16 network on the CIFAR-10 dataset, Table 4.3 shows a comparison of memory usage for different batch sizes. The *malloc* columns refer to non-contiguous memory while *cma* columns denote the contiguous memory allocations. Contiguous buffers are allocated for the inputs and outputs of the GEMM and sized for the largest layers of the network. The weights are saved in contiguous memory which avoids the overhead of copying data. The input activations are stored in non-contiguous memory but are written into contiguous buffers each iteration. Stochastic weight averaging requires an additional 61 MB for a floating-point copy of the low-precision weights. The table assumes that one image from the batch is processed at a time, denoted by $K_B = 1$. If a larger tiling factor is used then the size of each direct memory access (DMA) transfer will be larger and more contiguous memory must be allocated. A larger tile size could lead to improved performance as shown in Figure 4.5, but this has not been implemented.

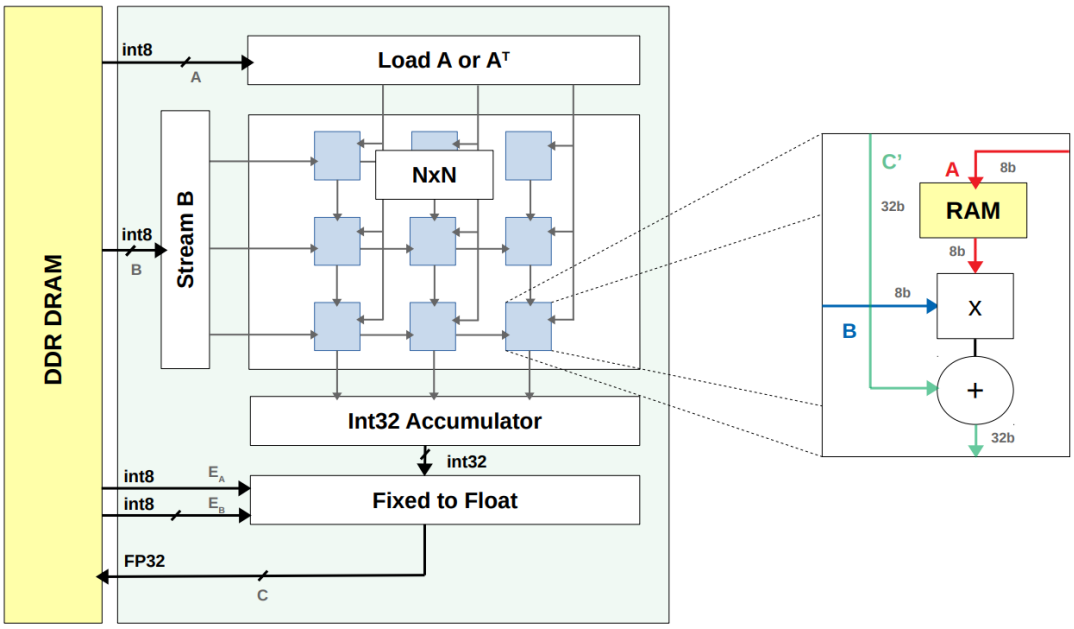


Figure 4.4: Low-precision GEMM FPGA Accelerator

4.3.2 Hardware Architecture

Figure 4.4 shows the high-level design of the accelerator architecture. A systolic array computes 8-bit matrix multiplication in fixed-point, and additional modules are included for buffering and transposing the input and output data streams, as well as implementing the fixed to floating-point data converter. The core computes matrix multiplication $AB = C$ or $A^T B = C$. The option is configurable at run-time.

4.3.2.1 Systolic Array

Matrix multiplication (in Figure 4.1) is mapped to a spatial $N \times N$ array of processing elements (PEs) as follows. First, the elements of matrix A are pre-loaded into PE block rams (BRAMs) over a high bandwidth on-chip network (NoC). B is streamed from DRAM in column-major order, forming N-length vectors which are broadcast across N columns in a pipeline. Each column computes one N-length dot product in fixed point, between a column of B and a row of A, forming partial results which are accumulated in 32-bit integer accumulators. The array can operate at high frequency since all connections between PEs are localised and have short wire lengths. If A represents the weights of a DNN then this describes a weight-stationary dataflow [70].

4.3.2.2 Processing Element

Each PE computes K_B multiply-accumulate (MAC) operations in parallel, where K_B=1 denotes the amount of batch (or instruction) parallelism in operand B. In terms of FPGA resources, this maps to one DSP block and up to four BRAM resources, depending on how much of A is cached on-chip. To achieve higher performance and better hardware efficiency, K_B should be chosen to maximize the utilisation of DSP and BRAM resources. This is shown via roofline analysis in Figure 4.5.

4.3.2.3 Fixed to Floating-Point Converter

The integer result C' is cast back to floating-point and re-scaled by successively multiplying the block floating-point exponents for matrix A and B. There is one exponent for the weights, and b exponents for the input activations and gradient activations, where b equals the batch size. The converter is fully pipelined and consumes moderate to low FPGA resources. Other preprocessing for matrix A, B and C is performed by the PS.

4.3.2.4 Memory and Array Partitioning

The accelerator assumes the entire matrix A can be pre-loaded into on-chip memory. Therefore, the maximum DNN size that can be train is limited by the availability of on-chip memory resources, i.e. the number of BRAMs and LUT RAMs on the FPGA. Figure 4.1 shows the array shapes the accelerator expects for A, B and C, for forward (top) and backward paths (middle and bottom). Since a batch is processed one example at a time, the on-chip memory needed for A is $A_{\text{FPGA}} \geq \max(oc \times ic \times k^2, oc \times w \times h)$. For VGG16 (see Table 4.2), this means the FPGA must have at least $512 \times 512 \times 9 \times 1 = 2.35MB$ of available memory. This is mapped to PE BRAMs. The PE memory is statically partitioned at compile time along both row and column axes, but the memory access pattern is run-time configurable allowing data to be fetched in either row-major or column-major order. This is critically important for training workloads specifically, where the weights W and the transposed weights W^T are both mapped to A and calculate the output activation and activation gradients in the forward and backward path respectively (as shown in Figure 4.1). Without run-time configuration the design would be highly inefficient and require approximately 9× more memory resources.

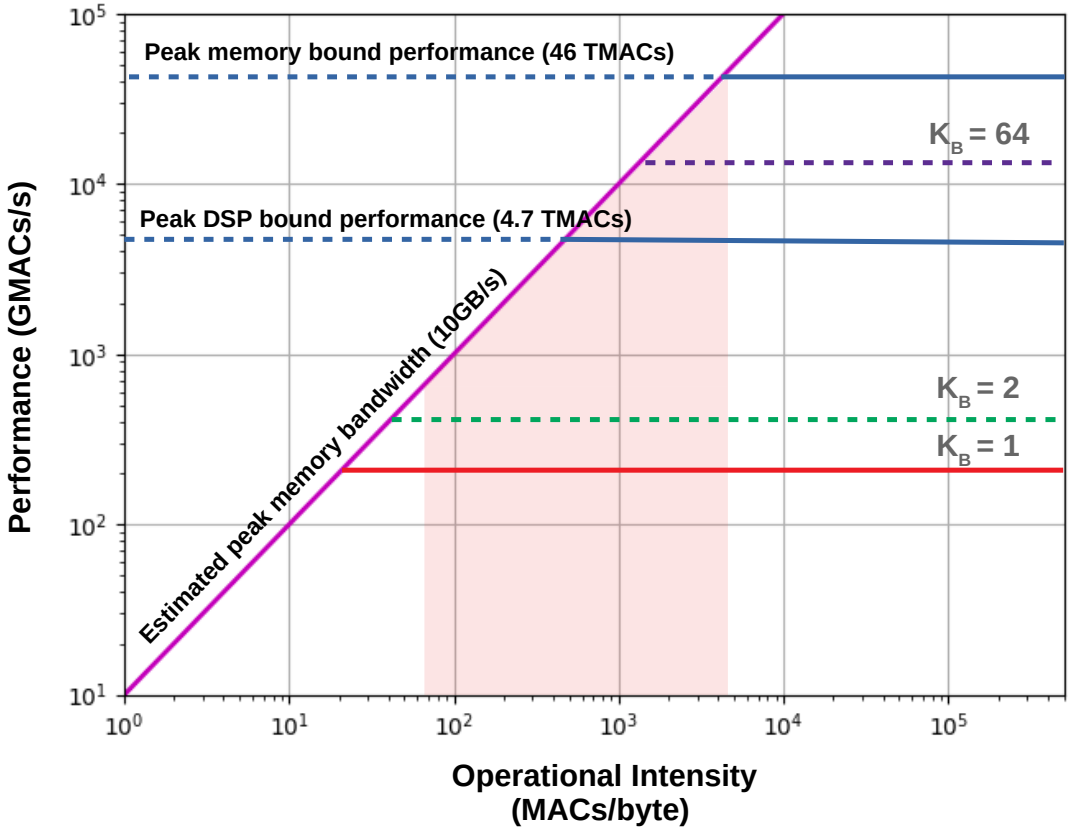


Figure 4.5: Roofline model: 32x32 PE array at 200 MHz

4.3.2.5 Performance Roofline

A roofline model [87] is used in Figure 4.5 to visually relate peak performance and off-chip memory bandwidth with the operational intensity of the full range of VGG16 layers. The red shaded area captures the full range of different VGG layers and their respective operational intensities. The largest matrix multiplication occurs in computing the backward path for layers with $oc = 512$ and $ic = 512$, where 512×9 MACs are required per input (or per byte). In contrast, the smallest VGG layer has an operational intensity of 64 MACs/byte. A conservative estimate of 10 GB/s is assumed for the memory bandwidth of the ZCU111 board. Therefore peak memory bound performance equals to $10 \times 512 \times 9 = 46,080$ giga-operations per second (GMACs/s), or 46 TMACs/s. Additionally, peak DSP performance is based on 4272 DSPs doing two 8-bit MAC at 550 MHz, therefore $4272 \times 2 \times 550 = 4.70$ TMACs/s. It can be said that designs that do not achieve 4.70 TMACs/s of performance are simply not utilising the available DSP resources efficiently. The red line is the reported implementation and is based on a 32×32 PE array doing K_B=1 MAC per cycle at 200 MHz. Therefore $32 \times 32 \times 200 = 205$ GMACs/s. Clearly, this design massively underutilizes available compute resources, and thus all VGG layers are compute bound. To utilise more of the available memory bandwidth and DSP resources, it would be advantageous to

consider designs that better exploit batch (or instruction) level parallelism, by computing multiple rows of B in parallel. This can be thought of as blocking or tiling, and the green and purple dotted lines show performance points of larger tile factors. The authors in [78] describe tiling strategies for optimising the GEMM for training, and there are many other highly optimised GEMMs that could replace the reported core.

4.4 Analysis and Implementation

The architecture and training algorithms were tested on two generations of Zynq SoC and MPSoC devices from Xilinx Inc. The Pynq-Z1 board, which integrates an ARM Cortex-A9 processor and small 7-series FPGA, and the ZCU111 development board (also known as RFSoC) which couples four ARM Cortex-A53 processors and a large Ultrascale+ FPGA. For comparative purposes, the training accuracy and run times are validated on standard well-known networks and datasets, such as VGG16 [86] and CIFAR-10 [32]. These problems are perhaps ill-suited to the low power embedded application domain, which is the main focus, but they are familiar to the research community in general. In the final section, the training accelerator is evaluated on the targeted problem with real-world constraints.

4.4.1 FPGA Resource Utilisation

Tables 4.4 and 4.5 show FPGA resource utilisation for different PE array configurations on the ZCU111 and Pynq Z1 boards. As mentioned previously, the accelerator is designed to fit the entire A matrix in on-chip memory, and more specifically in block rams (BRAMs) located in PEs. If this is not possible then larger layers can be handled by breaking A into smaller blocks and calling the accelerator multiple times. Arbitrarily large layers can be solved as long as they fit in ARM memory. However, let's assume for now that the accelerator can only be called once and therefore A_{max} refers to the largest supported matrix for a specific PE and BRAM configuration. Notably, A can be at most 672×6080 and 168×1520 on the ZCU111 and Pynq-Z1 boards respectively. This refers to designs that consume 97% and 55% of available BRAMs. Notably, further improvements to 55% BRAM usage are not possible on the Pynq-Z1 board due to power-of-two scaling associated with high-level synthesis (HLS) tools. Specifically, arrays in HLS must ordinarily be partitioned by power-of-two factors otherwise large amounts of additional logic is inferred leading to synthesis problems.

Table 4.4: Resource utilisation for ZCU111 board

Config.	A_{max} (row/col)	BRAM 36k (1080)	LUTs/DSPs/FFs (4.2M)/(4.2k)/(850k)	Freq. (Mhz)
16x16	240/2198	149	32.3k/269/20.7k	262
16x16	336/3040	277	32.5k/269/20.8k	253
32x32	480/4160	533	69.3k/1037/25.8k	246
32x32	672/6080	1045	73.1k/1037/25.6k	181

Table 4.5: Resource utilisation for Pynq-Z1 board

Config.	A_{max} (row/col)	BRAM 36k (140)	LUTs/DSPs/FFs (53k)/(220)/(106k)	Freq. (Mhz)
8x8	120/1064	45	13.9k/81/17.4k	114
8x8	168/1520	77	14.1k/81/17.4k	112

4.4.2 Training Accuracy

The key results from SWALP [21] have been reproduced in Table 4.6 for training VGG16, PreResNet-20 [88] and logistic regression networks on CIFAR-10 and MNIST datasets.

Table 4.6: Test accuracy (%) on CIFAR-10 and MNIST for VGG16, PreResNet-20 and Logistic Regression, trained with different quantisation schemes. The number of epochs taken to reach 92% accuracy for batch size 128 is also recorded.

Dataset	Model	Float		SWALP [21]		Ours (8-bit)	
		Acc.	Ep.	Acc.	Ep.	Acc.	Ep.
CIFAR-10	VGG16	93.02	205	92.47	195	92.93	177
	PreResNet-20	93.29	223	93.29	225	93.72	218
MNIST	Logistic Regression	92.6	104	92.06	66	92.7	108

All networks were trained with batch size 128, and without bias and batch normalisation. The training algorithm was run for 250, 350, and 150 epochs, for VGG16, PreResNet-20 and logistic regression respectively, using starting learning rates of 0.05, 0.1 and 0.1 which are decayed linearly. Stochastic weight averaging is applied for the last 25% of epochs

Table 4.7: Time per iteration (*secs*) on CIFAR-10 and MNIST for VGG16 and Logistic Regression, trained on Pynq-Z1 and ZCU111 boards

Dataset	Model	Batch	Pynq-Z1		ZCU111	
			A9	FPGA	A53	FPGA
CIFAR-10	VGG16	1	-	-	5.6	1.58
		32	-	-	172	9.89
		128	-	-	680	38.6
MNIST	LogReg	128	0.0401	0.0084	0.0121	0.0031
		256	0.0824	0.0149	0.0247	0.0057

with a modified constant learning rate of 0.01. Importantly, Table 4.6 shows that the BFP quantisation scheme achieves virtually the same test accuracy as full-precision floating-point networks and the prior work of SWALP [21]. This is expected since this work uses a combination of 8-bit fixed-point and full-precision floating-point (for the weight gradients and updates), whereas SWALP is entirely 8-bit. The choice for a mixed precision algorithm is predicated on the fact that Zynq has fast DDR memory shared between the PL and floating-point units in the PS. Basically, the weight updates can be computed efficiently in low-precision on the FPGA, and accumulated in high-precision on the ARM.

4.4.3 Performance

Training times per iteration are given in Table 4.7 for varying batch sizes. As mentioned previously, the accelerator does not exploit batch level parallelism, and therefore training times increase linearly with batch size. Designs with 8×8 and 32×32 array configurations were placed on the Pynq-Z1 and ZCU111 boards running at 100Mhz and 180Mhz respectively. Using low-precision and offloading all GEMMs to the FPGA, this achieves a 17× speed-up for CIFAR-10 and VGG16 on the ZCU111, and up to 5.5× and 4.3× speed-ups for MNIST on the Pynq-Z1 and ZCU111 boards respectively. This is compared to software running on an A53 (or A9) Arm processor and all computation done in full-precision floating-point. For an additional point of reference, a mid-range Tesla M40 GPU performs at roughly 16 iterations per second for batch 128 on CIFAR-10 and VGG16. This corresponds to a massive 600× speedup over the ZCU111 implementation. However, the work by Luo et al. [78] has shown that this performance gap can be significantly reduced, through GEMM optimisations and also by implementing more of the training on-chip. In applications that must be deployed stand-alone (without host communication), and which need the energy efficiency and low latency of FPGA inference engines, then

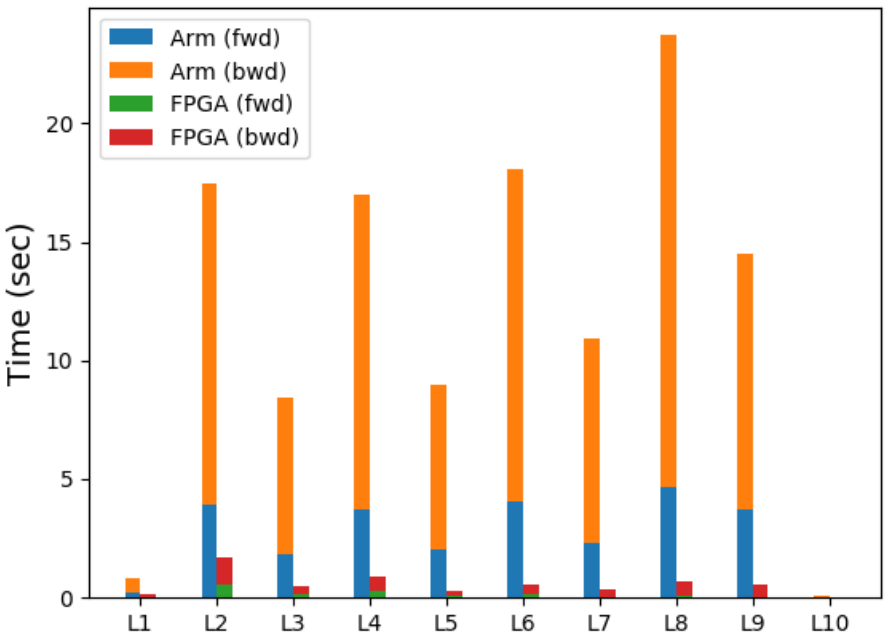


Figure 4.6: VGG16 per-layer forward and backward times on ZCU111

system designers are likely to tolerate slower training times as long as that training can be computed with high accuracy on-chip. The low-precision techniques presented in this work do not compromise the training accuracy. Figure 4.6 profiles the execution time and relative speed-up for forward and backward paths, and for each convolution layer in the VGG16 network (details provided in Table 4.2). The backward path requires an extra GEMM and takes observably longer than the forward path, and layer 8 and layer 2 take the most time on the ARM and FPGA respectively. The input and output activations are particularly large in layer 2 (i.e. $32 \times 32 \times 64$), and therefore proportionately more time is spent in software on the *im2col* and quantisation functions.

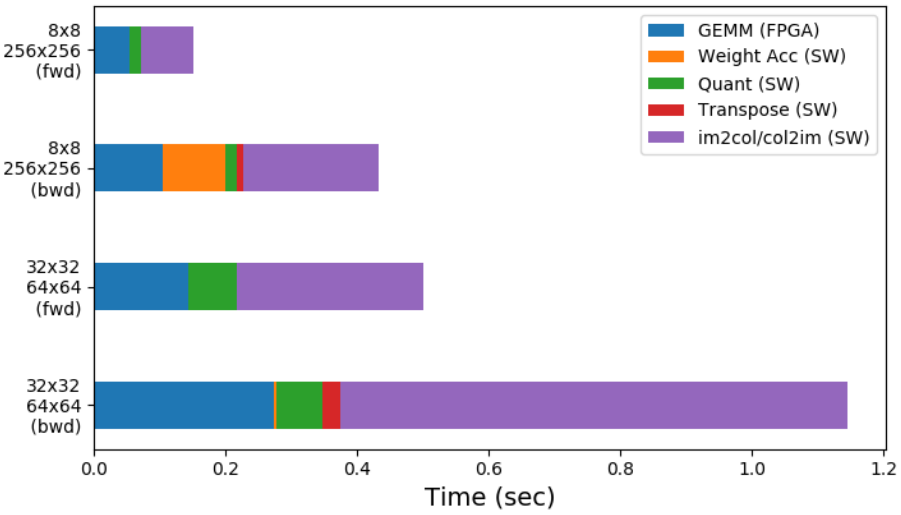


Figure 4.7: Software overhead of convolution layers

4.4.4 Software Overhead

Figure 4.7 profiles the run time and software overhead on two different convolution layers, for both the forward and backward paths. Clearly, *im2col* and *col2im* are the main performance bottlenecks and are especially damaging on layers with a large number of input and output activations (like VGG16 layer 2, $32 \times 32 \times 64$ pixels). Other significant overheads exist for quantisation and accumulating large weight matrices (like VGG16 layer 6, $256 \times 256 \times 9$). Future work should aim to move these functions into hardware, thus eliminating memory transfers by a factor of $k_w \times k_h$, where k_w and k_h denote the width and height of the convolution kernel (more details provided in Appendix A.3).

4.5 Case Study: Anomaly Detection in RF Networks

The application of neural networks to physical layer radio signals is extremely challenging due to the high data rates involved. In anomaly detection, one would like to perform training and prediction in an online manner to obtain a model of normal data, while simultaneously making normal/abnormal classifications. In this section, a low-precision variant of a small and simple DNN is trained for detecting anomalies in frequency modulated radio signals.

Figure 4.8 shows the network architecture for the RF anomaly detector. The main component is the autoencoder multi-layer perceptron (MLP) network. The first two layers encode and compress the input signal by learning a dimensionality reduction transform, and the final two layers reconstruct the encoding back to the original input signal. The mean-squared error loss is computed at the output layer using the original input signal as

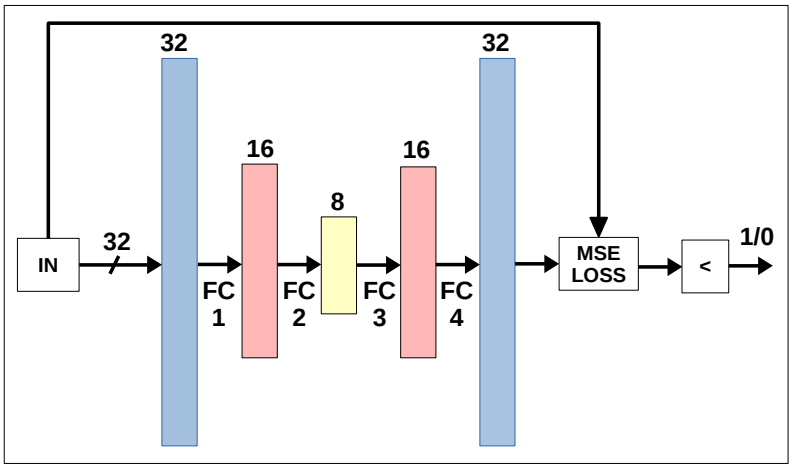


Figure 4.8: Auto-encoder DNN architecture

the truth values. An anomaly is detected when the loss is higher than a preset threshold value. This is a form of unsupervised learning.

Training is still performed by offloading each GEMM to the FPGA accelerator on the ZCU111 board. This experiment uses a 16×16 PE array configuration running at 200Mhz. The training data consists of sliding windows of complex I/Q samples. Each window has a length of 32 which represents the dimensionality of the input vectors. Training is performed by collecting and then iterating low-precision SGD over a batch of 1000 inputs. To validate the performance of the anomaly detector during training, a test set was created with three different types of noise (i.e. bandpass, chirp and complex sine), and an F1-score computed for correctly detecting the known anomalies. Figure 4.9 shows a plot of the F1-score on the test set for floating-point and 8-bit training algorithms. The anomaly detector shows convergence to an F1-score of approximately 0.87 after about 200 iterations for both curves. The 8-bit version is accelerated by the FPGA and is observably faster to converge. This reflects a 1.75× speed-up over training with software-only. Much larger speed-ups are expected for wider networks that require more computation.

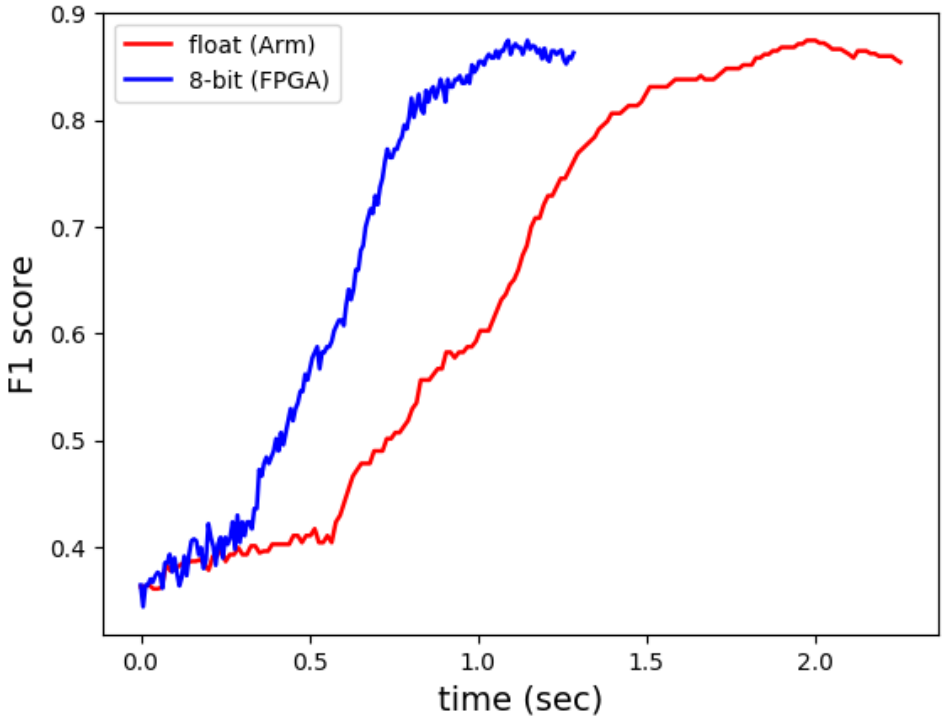


Figure 4.9: Comparison of F1-scores and training times for anomaly detection

4.6 Summary

In this chapter, new techniques were introduced for efficiently training DNNs with high accuracy on Zynq devices. The convolution and fully-connected layers are computed on the FPGA using only 8-bit block floating-point numbers for weights, activations and activation gradients, while the rest of the network is computed in full-precision on an ARM processor. A prototype training accelerator was designed for portability across multiple Zynq boards and integration with high-level software frameworks. Results of an implementation on multiple benchmarks show up to 17× speed-ups over processor only systems without any degradation in training accuracy.

Chapter 5

Training Deep Neural Networks: Block Minifloat

This chapter presents Block Minifloat, a specialized number representation for training deep neural networks efficiently and with high accuracy.

The presentation of this chapter is based on background given on deep neural networks and computer arithmetic in Chapter 2 and expands on work previously published in [89].

5.1 Introduction

The energy consumption and execution time associated with training deep neural networks (DNNs) are directly related to the precision of the underlying numerical representation. Most commercial accelerators, such as NVIDIA Graphics Processing Units (GPUs), employ conventional floating-point representations due to their standard of use and wide dynamic range. However, double-precision (FP64) and single-precision (FP32) formats have relatively high memory bandwidth requirements and incur significant hardware overhead for general matrix multiplication (GEMM). To reduce these costs and deliver training at increased speed and scale, representations have moved to 16-bit formats, with NVIDIA and Google providing FP16 [44] and Bfloat16 [45] respectively. With computational requirements for DNNs likely to increase, further performance gains are necessary for both datacenter and edge devices, where there are stricter physical constraints.

New number representations must be easy to use and lead to high accuracy results. Recent 8-bit floating-point representations have shown particular promise, achieving equivalent FP32 accuracy over different tasks and datasets [64, 65]. Such representations are referred to as *minifloats*. Minifloats are ideal candidates for optimization. By varying the number of exponent and mantissa bits, many formats can be explored for different trade-offs of dynamic range and precision. These include logarithmic and fixed-point

representations which provide substantial gains in speed and hardware density compared to their floating-point counterparts. For instance, 32-bit integer adders are approximately 10× smaller and 4× more energy-efficient than comparative FP16 units [18]. That said, fixed-point representations still lack the dynamic range necessary to represent small gradients for backpropagation, and must be combined with other techniques for training convergence.

Block floating-point (BFP) (in Chapter 4) share exponents across blocks of 8-bit integer numbers and provide a type of coarse-grained dynamic range for training. This approach will typically incur some accuracy loss on more challenging datasets [21, 22], however, all dot products within the block can be computed with dense fixed-point logic. In comparison, HFP8 [65] minifloats require larger floating-point units (expensive FP16 adders in particular) but have at least 5 exponent bits dedicated to each gradient and suffer zero degradation in training accuracy. It would seem that an ideal representation should bridge the gap between each of these approaches. This chapter introduces techniques for doing this with 8-bit and sub 8-bit precision schemes, overcoming two key challenges in the process. These are listed below and discussed with related works.

5.1.1 Challenges and Related Work

Minimising data loss with fewer bits: While several works have demonstrated training with fewer than 8 bits of precision, they typically lead to loss of accuracy on more complex problems and have performance bottlenecks because parts of the algorithm are left in high-precision [62, 90, 91]. Therefore, training end-to-end with reduced precision representations that are persistent remains a key challenge. In this regard, 8-bit tensors with 16-bit updates can be trained effectively [66]. Data loss arises when formats do not have enough range to capture variations in tensor distributions during training. BFloat [45] adds two extra exponent bits for a custom 16-bit representation, and the Apex library is used in [64, 65, 92] for scaling the loss function into a numerically representable range. Block floating-point and other variants apply similar functionality for fixed-point numbers but at a finer granularity. WAGE [93] uses layer-wise scaling factors, SWALP [21] shares exponents across feature maps and convolution channels, and HBFP [22] does the same for dot products, though their implementation requires caching of intermediate activations in FP32 and wide weight storage for better accuracy. S2FP8 [94] replaces loss-scaling in FP8 [64] with squeeze and shift factors that centre 8-bit minifloats over the mean exponent of the value distribution. Shift factors operate similarly to BFP shared exponents, whereas squeeze factors can divert precision away from high-value regions leading to errors in dot-product calculations. Empirical evidence of this effect is provided in Section 5.4.5. Finally, HFP8 [65] defines two minifloat formats that are optimized for range and precision requirements of forward and backward paths separately. This work seeks minifloat formats

that are also optimized for arithmetic density.

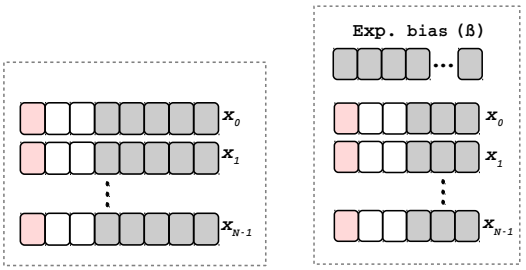
Increasing the performance density of floating-point: Most DNN training frameworks are developed with GEMM accumulation in FP32. The authors in [64] reduced the accumulation width to FP16 with chunk-based computations and stochastic rounding. However, training minifloats with even denser dot products has not been demonstrated. For DNN inference, ELMA [95] and posit number systems [96] describe arithmetic that accumulates minifloat-like numbers as integers. Such work is applicable when the number of exponent bits is small, however training under such regimes can lead to data loss due to limited dynamic range.

5.1.2 Contributions

The Block Minifloat (BM) representation is presented which addresses both of the aforementioned challenges. BM is a modification of block floating-point that replaces the fixed-point values with minifloats, whilst maintaining shared exponents across blocks of numbers. BM formats generalise a far wider spectrum of reduced precision representations and produce better outcomes than previous 8-bit regimes. Specific contributions of this chapter include:

- Block Minifloat (BM), a more efficient alternative to INT8 and FP8 for end-to-end DNN training. Shared exponent biases provide dynamic range and accuracy, while small exponent encodings provide fine-grained dynamic range and reduce the hardware cost of GEMM accumulation.
- A new 8-bit floating-point format that uses no more than **4 exponent bits**, achieving equivalent accuracy to floating-point with denser hardware via efficient Kulisch accumulation.
- An exploration of the BM design space showing high accuracy DNN training with sub 8-bit representations for all weights, activations and gradients. This includes two techniques for minimising data loss of a practical implementation, namely gradual underflow and cost-aware block designs.

The chapter is organised as follows: Section 5.2 introduces the Block Minifloat representation; Section 5.3 describes loss minimization techniques and implementation details; Section 5.4 provides training results on a wide variety of DNN models and datasets; the hardware is evaluated in Section 5.5, and lastly, Section 5.6 summarizes the chapter.



(a) Minifloat **(b)** BM (Shared exp. bias)

Figure 5.1: Description of Minifloat and Block Minifloat (BM) tensor representations

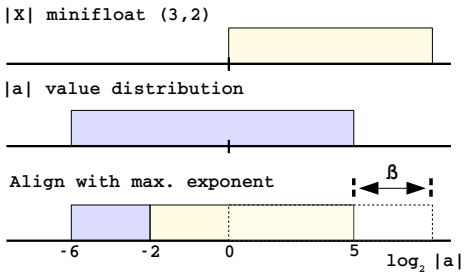


Figure 5.2: Exponent bias shifts the minifloat distribution to align with the maximum exponent of the value distribution

5.2 Block Minifloat Representation

5.2.1 Minifloat Number Format

Equation (5.1) computes the real value of a minifloat number, where (e, m) denote the number of exponent and mantissa bits in the representation.

$$X\langle e,m\rangle = \begin{cases} E=0, & (-1)^s \times 2^{1-\beta} & \times(0+F\times 2^{-m}) & \text{(denormal)} \\ \text{otherwise}, & (-1)^s \times 2^{E-\beta} & \times(1+F\times 2^{-m}) & \text{(normal)} \end{cases} \tag{5.1}$$

The decimal expansions of the exponent and mantissa are both unsigned integers, given by E and F respectively, s refers to the sign bit and $\beta = 2^{e-1} - 1$ is the exponent bias for the binary-offset encoding scheme. This is consistent with IEEE-754 floating-point standards, except that these minifloats are considerably smaller (4-8 bits only), can generalise to multiple (e, m) configurations, and do not handle nan/infinity bit patterns. Instead, arithmetic is allowed to saturate at the limits of the representable range $[X^+_{min}, X^+_{max}]$. For example, a minifloat representation with $X\langle 4, 3\rangle$ have exponent and mantissas that range between $[0, 15]$ and $[0, 7]$ respectively. Therefore, the largest normal and smallest denormal positive numbers are $X^+_{max} = 480$ and $X^+_{min} = 2^{-9}$. This corresponds to a dynamic range of 108 decibels (dB).

5.2.2 Shared Exponent Bias

The main difference between minifloat and BM representations are highlighted in Figure 5.1. Minifloats have one exponent per element, but that exponent must be wide enough to tolerate changes in DNN tensor distributions during training (i.e. 5 bits for gradients in FP8 [64]). In contrast, BM share exponent biases across blocks of N minifloat numbers. This provides the equivalent dynamic range with respect to the block, but with fewer

exponent bits than the original minifloat format. BFP operates similarly, but all numbers within the block are integers [22]. BM can generalise for this case, i.e. when $e = 0$.

The real value of the i^{th} element from BM tensor a is given in Equation (5.2), where X is an unbiased minifloat tensor, represented by (e,m) exponent and mantissa bits, and β_a is the shared exponent bias.

$$a_i = X_i\langle e, m\rangle \times 2^{-\beta_a} \tag{5.2}$$

In this example, a_i can only be represented accurately when the shared exponent bias β_a (calculated for the entire tensor) and the distribution of X jointly captures the value distribution of a. For example, large and small values in a could saturate or be lost altogether if β_a is too large or too small. However, some leeway exists when exponents are shared across dot products. This is because dot products are reduce operations, meaning their sum is dominated by the largest values in the inputs. For this reason, β_a is calculated to specifically guard against overflow, and unlike [94] there is no scaling that could divert precision away from larger value regions. The method of updating β during training is illustrated in Figure 5.2 and formalized in Equation (5.3) below.

$$\beta_a = \max\left(\lfloor \log_2 |a| \rfloor\right) - \left(2^e - 1\right) \tag{5.3}$$

The first term denotes the maximum exponent for the tensor a, which changes and must be updated during training, while the second term is fixed and refers to the maximum exponent of X.

In terms of hardware, shared biases ensure that all dot products can be computed with denser minifloat arithmetic. This is shown in Equation (5.4) for BM tensors a and b, each with N elements.

$$a \cdot b = \sum_{i=1}^{N}\left((X_i^a \times 2^{-\beta_a}) \times (X_i^b \times 2^{-\beta_b})\right) = 2^{-(\beta_a+\beta_b)} \times (X^a \cdot X^b) \tag{5.4}$$

The dot product, $X^a \cdot X^b$, have minifloat formats with smaller exponents, while the cost of calculating, storing and aligning the exponent biases during training is amortized over the length of the dot-product. Next, it is shown how minifloat formats with fewer exponent bits can lead to faster and more compact hardware.

5.2.3 Kulisch Accumulation

A Kulisch accumulator [97] is a fixed point accumulator that is wide enough to compute an error-free sum of scalar floating-point products, over the entire range of possible values. Kulisch accumulators operate by shifting the mantissa of the floating-point product into an

Table 5.1: Kulisch accumulator examples

Format		Kulisch Acc.	
(e_a, m_a)	(e_b, m_b)	kadd	kshift
(8, 23)	(8, 23)	561	512
(5, 2)	(6, 1)	102	96
(4, 3)	(5, 2)	56	48
(3, 4)	(4, 3)	34	24
(2, 3)	(3, 2)	20	12
INT8	INT8	32	-

internal register according to the exponent of the product. The sum proceeds as integer addition which is $4-10\times$ more efficient in terms of area and power compared to FP16 [18]. The number of bits required for the internal register (i.e. the addend) and shifter, scale the size and complexity of the accumulator and are provided as formulas in Equation (5.5) for BM operands $a = (e_a, m_a)$ and $b = (e_b, m_b)$.

$$kadd = 1 + (2^{e_a} + m_a + 1) + (2^{e_b} + m_b + 1) \tag{5.5}$$

$$kshift = 2^{e_a} + 2^{e_b} \tag{5.6}$$

In the above equations, *kadd* calculates the number of bits required for the largest product of two numbers, plus one extra bit for the addition, and *kshift* determines the maximum number of bits the mantissa product must be shifted to align with the addend. Crucially, by considering the size of *kadd* and $kshift$, BM formats can be designed to trade-off fine-grained dynamic range (i.e. exponent bits) for more precision and smaller hardware. In fact, formats with exponents up to 4 bits may yield *kadd* of approximately the same size as INT8/INT32 arithmetic units, while *kadd* becomes prohibitively wider and more expensive for larger exponents. This is clearly shown via example in Table 5.1 above, but more importantly, it is supported by hardware synthesis results given in Section 5.5 and Appendix B.3. For example, an 8-bit minifloat format having 4 exponent bits achieves a $1.6\times$ area reduction compared to HFP8 [65] with 5 exponent bits. Furthermore, through an extensive set of experiments, it was discovered that such representations also achieve high training accuracy which is a key contribution of this work.

5.3 Training with Block Minifloat

5.3.1 Minimizing Data Loss

BM arithmetic will incur data loss when the value distribution is too wide or requires more precision than can be captured by the underlying minifloat representation within a block. Below, steps are described for mitigating this problem without substantially increasing implementation overheads.

Gradual underflow: The reported minifloats support denormal numbers as defined in Equation (5.1). Denormal numbers have precision close to zero and ensure that consecutively smaller quantized numbers approach zero gradually. The alternative is flush-to-zero which discards the mantissa bits, producing a zero value, when $E = 0$. This equates to approximately 12.5% of the exponent encoding when $e = 3$; this is highly inefficient. Overhead for denormal numbers in hardware is minimal, and only requires detection of $E = 0$ and a single bit flip in the multiplier. Experiments show that gradual underflow is crucial for BM formats with less than four exponent bits.

Block Size: Matrix multiplication with BM is computed by dividing tensors into $N \times N$ blocks that bound the number of exponent biases and reduce data loss since each block shares one exponent bias. Square blocks are chosen so that biases are contiguous in memory regardless of whether the block is operating in the forward path or after transposition in the backward path. As such, BM can be stored with a persistent data structure, that doesn't require recasting or extra memory transfers during training. This makes BM easy to use at the software level but does mean that biases are shared across N independent dot products. In terms of hardware cost, Equation (5.7) formalizes the relationship between the size of N and three overheads, where α refer to relative area costs for each overhead.

$$\text{cost} = \overbrace{\alpha_1\Big(1 + \frac{\log_2 N}{\text{kadd}}\Big)}^{\text{Kulisch}} + \overbrace{\alpha_2 \frac{1}{N}}^{\text{FP}} + \overbrace{\alpha_3 \frac{8 + N^2}{N^2}}^{\text{Memory}} \tag{5.7}$$

For the first term, the width of the Kulisch accumulator must increase by $\log_2 N$ bits to prevent overflow in the BM dot products. In the second term, floating-point hardware (including quantization and conversion modules) are required to accumulate, align and convert BM partial results, but the cost is amortized over N fixed-point operations. Finally, in the last term, additional memory is required to compute and store one 8-bit bias for every $N \times N$ values. For large block sizes, the extra silicon area from Equation (5.7) is negligible compared to the GEMM but data loss from sharing biases can still be significant.

In Section 5.4.5, it was determined that a block size of $N = 48$ offers a good balance for both objectives and is used for the rest of this paper.

Hybrid representation: Different minifloat representations for forward and backward paths have been shown to produce better accuracy for FP8 training [65]. The same idea is used for BM, where the best balance of precision and range is determined for each path separately. Full details for all BM formats are provided in Table 5.2 where forward and backward configurations are given by $(e_f, m_f)/(e_b, m_b)$ notation. BM formats cover each precision level between 4 and 8 bits and are denoted by BM4, BM5, BM6, BM7 and BM8. For example, BM6 $(2, 3)/(3, 2)$ refers to 6-bit BM training with weight and activation tensors represented by $(2, 3)$ and activation gradient tensors represented by $(3, 2)$ minifloat formats.

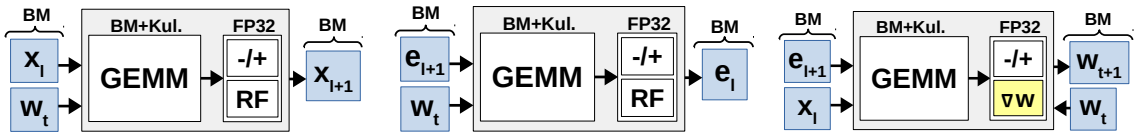
Table 5.2: Comparison of Block Minifloat number formats

Scheme	Format[1] (e, m)	Range[2] (dB)	Precision[3] (ϵ)	Kulisch Acc. kadd	kshift
FP32	(8,23)	1668	2^{-24}	561	512
Bfloat16 [45]	(8,7)	1529	2^{-8}	529	512
FP16	(5,10)	241	2^{-11}	87	64
FP8 [64]	(5,2)	185	2^{-3}	71	64
S2FP8 [94]	(5,2)	185	2^{-3}	71	64
HFP8 [65]	(4,3)/(5,2)	108/185	$2^{-4}/2^{-3}$	56	48
SWALP [21]	(0,7)	42.1	2^{-8}	32	-
INT8 [93]	(0,7)	42.1	2^{-8}	32	-
BM8 (ours)	(2,5)/(4,3)	48.0/108	$2^{-6}/2^{-4}$	31	20
BM7 (ours)	(2,4)/(4,2)	41.9/101	$2^{-5}/2^{-3}$	29	20
BM6 (ours)	(2,3)/(3,2)	35.6/53.0	$2^{-4}/2^{-3}$	20	12
BM5 (ours)	(2,2)/(3,1)	28.9/45.7	$2^{-3}/2^{-2}$	18	12
BM5-log (ours)	(4,0)/(4,0)	28.9/45.7	$2^{-3}/2^{-2}$	33	32
BM4 (ours)	(2,1)/(3,0)	21.6/42.1	$2^{-3}/2^{-1}$	16	12
BM4-log (ours)	(3,0)/(3,0)	28.9/45.7	$2^{-3}/2^{-2}$	17	16

[1] hybrid formats, i.e. forward/backward
[2] dynamic range in decibels $20\log_{10}(X^+_{max}/X^+_{min})$
[3] relative round-off error, i.e. $2^{-m} \times 2^{-1}$

5.3.2 Training Details and GPU Simulation

BM offers an alternative to standard FP32 for the computationally intensive parts of training, which is typically mapped to general matrix multiplication (GEMM). However, specialized hardware is required to realise its potential gains in speed and energy efficiency.



(a) Fwd activation **(b)** Bwd activation grad. **(c)** Bwd weight grad. and update

Figure 5.3: End-to-end Training with Block Minifloat (BM). All off-chip memory transfers are low-precision BM tensors. BM alignments, weight updates, quantization, batch normalization and ReLU are executed in on-chip scalar FP32 units. The register file (RF) stores a block of ∇W.

For the purposes of this work, the behaviour of BM hardware is simulated using GPUs and PyTorch [98]. Given that dot products are computed exactly via Kulisch accumulators, existing CUDA libraries for GEMM can be used without modification, and all data loss is attributed to quantization only. Figure 5.3 provides an illustration of the setup for each GEMM in forward and backward paths. In a practical implementation, BM does not require any costly movement or storage of high-precision tensors. This is enabled by scalar processors after the GEMM (for FP32 operations, Kulisch to floating-point conversion, block minifloat alignments, quantization etc.) and a weight update scheme that can compute and cache high-precision gradients on-chip [65]. Weight, activation and gradient tensors are quantized to BM numbers with stochastic rounding as described in [64]. For the software simulation, quantization is applied before each GEMM in forward and backward paths and contributes significant performance overhead compared to standard PyTorch layers. An approximate 5× slow-down is realised on most networks and datasets, with support for denormal numbers the main implementation bottleneck. The realisation of the same function is comparatively cheap in custom hardware however and can be fully pipelined for fast training times.

5.4 Experiments

The training accuracy of BM was evaluated on a subset of image, language and object detection modelling tasks. The entire spectrum of representations was explored on ImageNet [99] and CIFAR [32] image recognition benchmarks, with results compared against well-calibrated INT8, FP8 and FP32 baselines. On other tasks, BM8 is compared with an FP32 baseline.

5.4.1 CIFAR-10 and CIFAR-100

CIFAR experiments are run using SGD with momentum of 0.9 for 200 epochs in batches of 128 images and an initial learning rate of 0.1 which is decayed by a factor of 5 at the 60th,

Table 5.3: Final Validation Accuracy (%) on CIFAR datasets for ResNet-18

Scheme	CIFAR-10	CIFAR-100
FP32 (ours)	94.9	77.5
BM6 (2,3)/(3,2)	95.1	77.2
BM5 (2,2)/(3,1)	94.7	76.1
BM4 (2,1)/(3,0)	94.2	73.7

Table 5.4: Training Accuracy (%) on CIFAR-10 for VGG16 and log quantization

CIFAR-10	FP32	Log	∇	kshift (bits)
log-5b[1]	94.1	93.8	-0.3	32
log-BM5 (ours)	93.8	93.4	-0.4	32
log-BM4 (ours)	93.8	93.1	-0.7	16

[1] [91]
[2] results achieved with base $\sqrt{2}$

120th and 160th epochs. Table 5.3 presents results for training ResNet-18 [16] with only small BM6, BM5 and BM4 representations. These offer the highest reduction in memory usage while still reaching very close to the FP32 baseline. For example, 6-bit BM training only records a 0.3% loss in accuracy compared to FP32 on CIFAR-100 while theoretically saving 25% of memory read and write overheads compared to FP8. Logarithmic BM formats were also tested on CIFAR-10 and VGG16 network. Log representations arise when $m = 0$, and require only adds and shifts for multiply-add arithmetic. Experiments for log-BM use the same training parameters as before and are shown in Table 5.4. Results are compared against the only previously known result for log training, i.e. *log-5b* [91] and achieve similar results with respect to FP32 for 5-bit and 4-bit. BM representations have exponent biases that shift tensor distributions dynamically during training, whereas *log-5b* define offset parameters at each layer that are fixed. Allowing biases to vary during training gives BM an advantage, and results in similar validation accuracy with only 4-bit words. This corresponds to approximately half the cost for multiplication in the linear domain (by exponent add and Kulisch shift).

5.4.2 ImageNet

The ImageNet dataset has 1000 class labels, and consists of 256x256 images split into a training set with 1.28 million images and validation set with 50,000 images. ResNet-18 [16]

Table 5.5: Top-1 accuracy (%) of reduced precision (RP) training on ImageNet for ResNet-18 models

Scheme	Numerical representation (e, m)					ResNet-18	
	w	x	dw	dx	acc	FP32	RP
SWALP [21]	8^1	8^1	8^1	8^1	32^1	70.3	65.8
S2FP8 [94]	$(5,2)/(8,23)$	$(5,2)$	$(5,2)$	$(5,2)$	$(8,23)$	70.3	69.6
HFP8 [65]	$(4,3)$	$(4,3)$	$(5,10)$	$(5,2)$	$(5,10)$	69.4	69.4
BM8 (2,5)/(4,3)	$(2,5)$	$(2,5)$	$(6,9)$	$(4,3)$	31^1	69.7	69.8
BM7 (2,4)/(4,2)	$(2,4)$	$(2,4)$	$(6,9)$	$(4,2)$	29^1	69.7	69.6
BM6 (2,3)/(3,2)	$(2,3)$	$(2,3)$	$(6,9)$	$(3,2)$	20^1	69.7	69.0
BM5 (2,2)/(3,1)	$(2,2)$	$(2,2)$	$(6,9)$	$(3,1)$	18^1	69.7	66.8

[1] Fixed point

and AlexNet [24] architectures are used from the official PyTorch implementation [1], and trained on one GPU with standard settings; SGD with momentum of 0.9, batches of 256 images, and an initial learning rate of 0.1 (0.01 for AlexNet) which is decayed by a factor of 10 at epoch 30 and 60. ResNet-18 has been widely tested upon in previous work and offers the most suitable benchmark for exploring the full spectrum of BM representations, especially given the size of the network as well as the cost of BM quantization on training times (approx. 5× slow-down). Results are presented in Table 5.5 where columns w, x, dw, dx and acc refer to the numerical representation for weight, activation, weight gradient, activation gradient and on-chip GEMM accumulator. Equivalent FP32 accuracy is achieved for BM8 and BM7, slight degradation for BM6, while BM5 exceeds the reported accuracy for 8-bit SWALP [21]. Compared to S2FP8 [94], BM6 representation reaches similar levels of relative accuracy, but with two fewer bits and without a high-precision master copy of the weights. Insights are provided into potential reasons for this in Section 5.4.5 by considering the possibility of diminishing returns in accuracy from scaling minifloat representations. Compared with HFP8 [65], which offers robust 8-bit training results, BM8 produces the same accuracy on ImageNet while improving upon HFP8 in hardware density and performance. BM8 tensors can be represented with fewer exponent bits, and thus perform dot products via Kulisch accumulators that are smaller and faster than FP16 units. Furthermore, BM offers tradeoffs for even denser arithmetic and lower memory usage. In these regimes, BM hardware is more comparable to SWALP [21] which performs the GEMM in fixed-point. Proof of BM design efficiencies is provided with RTL synthesis results in Section 5.5, Figure 5.6.

[1] Implementation available at https://github.com/pytorch/examples/tree/master/imagenet

Table 5.6: Comparison of Block Minifloat (BM) and Block Floating Point (BFP) number formats trained on ImageNet with ResNet-18 model.

Scheme	BFP (ours)	BM (ours)	∇
6-bit	67.0	69.0	+2.0
8-bit	69.2	69.8	+0.6

5.4.2.1 Comparison with Block Floating Point (BFP) on ImageNet

As described in Section 5.2, BM formats bridge the gap between narrow floating-point and block floating point (BFP) representations. The main idea is that better outcomes in terms of accuracy and hardware efficiency can be achieved by exploring the spectrum between the two representations. While BFP ensures that the majority of computation involves dense integer arithmetic, the lack of fine-grained dynamic range typically leads to accuracy loss on larger models and more complex datasets. In Table 5.6, BM is shown to recover accuracy loss for 6-bit and 8-bit formats on ImageNet training. Crucially, the hardware overhead of BM is minimal thanks to narrow exponents and Kulisch accumulators.

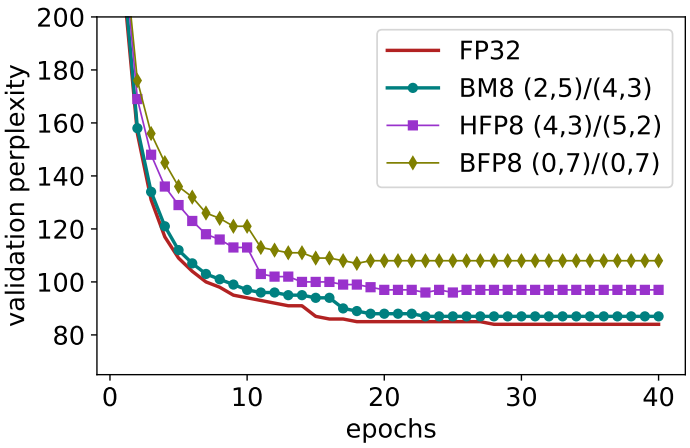


Figure 5.4: Validation perplexity of LSTM model on Penn Treebank

5.4.3 Language Modelling with LSTM

This section compares 8-bit formats for language modelling on the Penn Treebank dataset [100]. The 2-layer Long Short Term Memory (LSTM) network is adapted from PyTorch Examples [2] with all GEMM operations performed using BM8 arithmetic. The batch size is 20, the initial learning rate is 20 with 0.25 decay, the embedding and hidden dimensions are 650 and the sequence length is 35. Results in Figure 5.4 show BM8 with $(2, 5)/(4, 3)$ hybrid

[2] Implementation available at https://github.com/pytorch/examples/tree/master/word_language_model

configuration achieving better accuracy than BFP8 and HFP8 variants. The proposed BM8 representation has more fine-grained dynamic range and fewer mantissa bits than BFP8, and more precision and fewer exponent bits than HFP8 formats. This design point achieves better outcomes in terms of accuracy and hardware density than either representation separately (see Figure 5.6, Section 5.5). Validation perplexity of 87.33 is also comparable to 84.70 obtained with full-precision floating-point.

Table 5.7: Baseline FP32 v BM8 training on Image, Language and Object Detection models

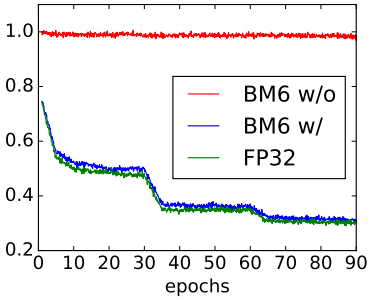
Model (Dataset) [Metric]	FP32	BM8
AlexNet (ImageNet)	56.0	56.2
EfficientNet-b0 (small ImageNet)	62.6	61.8
LSTM (PTB)[Val ppl.]	84.7	87.33
Transformer-base (IWSLT)[BLEU]	32.3	31.8
SSD-Lite (MbNetV2) (VOC)[mAP]	68.6	68.0

5.4.4 Additional Experiments

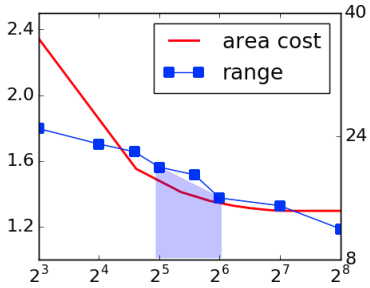
To demonstrate wider applicability of the BM number representation, BM8 was tested on several additional networks and modelling tasks. Results are summarized in Table 5.7 with experiment setup details provided in Appendix B.2. Crucially, every network tested achieved comparable accuracy with baseline FP32. This includes EfficientNet-b0 [101] image classification and SSD-lite [102] with MobileNet-V2 object detection models, both of which represent the type of network and application well suited to resource-constrained hardware devices. Furthermore, a small Transformer network was also trained for translation on the IWSLT German to English dataset [103]. Future work could involve scaling up the implementation and demonstrating training with BM representations on even larger networks and datasets. Network design with BM is another interesting research direction since the majority of network architectures have been designed and optimized while assuming an FP32 arithmetic scheme.

5.4.5 Empirical Analysis

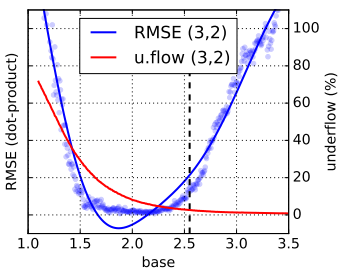
Effect of Denormal Numbers: To study the effect that denormal numbers have on training convergence in sub 8-bit networks, ResNet-18 was trained on ImageNet for BM6 with denormals (ours) and without denormals, using QPyTorch library [104]. Results are plotted against floating-point accuracy in Figure 5.5a. Without denormals, small numbers are flushed to zero and training stagnates immediately. Although not shown here, 8-bit



(a) Validation Accuracy: Training with denormal numbers on ImageNet



(b) HW (left axis) vs Range (right axis): Selecting the block size



(c) Minifloat scaling by varying the exponent base

Figure 5.5: Experiments for minimising data loss with 6-bit Block Minifloat (BM6)

representations with more than $e = 3$ bits do not suffer similar accuracy degradation without denormals. This investigation confirms the importance of denormal numbers for training BM formats with fewer exponent bits and differentiates this chapter's experiments substantially from previous 8-bit regimes.

Selecting the Block Size: Experiments were conducted on CIFAR-100 to determine suitable block sizes - those which simultaneously increase dynamic range and have low hardware overhead. Results are shown in Figure 5.5b. The largest range observed in gradient tensors at different block settings was taken as an average over the entire duration of training. Estimates of area come from Equation (5.7) with parameters; $\alpha_1 = 1$, $\alpha_2 = 10$ (relative area of fixed-point and floating-point respectively), $kadd = 21$ and $\alpha_3 = 0$. The area cost is saturated at $N = 256$, which is consistent with the length of dot products supported by the GEMM architecture in commercial hardware [38]. Finally, $N = 48$ emerged as a good selection, corresponding to one floating-point unit for every 48 multiply-accumulate operations and one 8-bit exponent bias every 2304 minifloat numbers.

Scaling the Minifloat Representation: In Figure 5.2, which was discussed previously, minifloats have exponent biases that shift the representation to align with the maximum of the underlying value distribution. Additionally, the minifloat representation could be scaled (or stretched) over a wider or narrower part of the value distribution. This effect is investigated by varying the base of the exponent, and inspecting the underflow and root mean square error (RMSE) of dot products after quantization; results are shown in Figure 5.5c. The tensor under test is a gradient tensor with maximum exponent of -17 and mean exponent of -21. Mean scaling was proposed in S2FP8 [94] for 8-bit training and works by centring the minifloat over the mean of the exponent value distribution. For the (3,2) format, mean scaling requires a base of 2.52, calculated as $b = 2^{\frac{-17+21}{7-4}}$. This is akin to

redirecting precision from high-value regions into smaller underflow regions, the result of which observably leads to increased error in the tested 6-bit regime. Better approaches could be designed to detect underflow and use higher precision arithmetic where necessary.

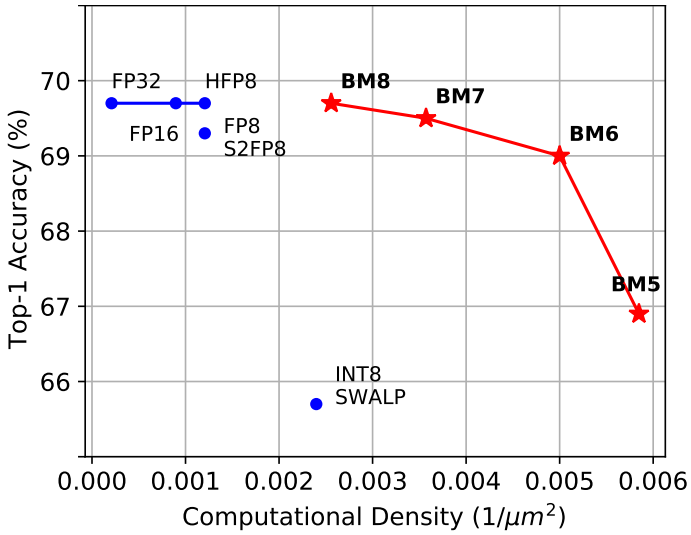


Figure 5.6: Computational density v ResNet-18 accuracy on ImageNet

5.5 Hardware Evaluation

In this section, the proposed BM representation is evaluated in hardware and compared against competitive integer and floating-point arithmetic types. Figure 5.6 summarizes the main results with a plot of computational density (measured as operations per unit silicon area) and ResNet-18 training accuracy on ImageNet. Computational density was obtained from an register-transfer level (RTL) design of single-cycle fused multiply-add (FMA) units and 4x4 systolic array multipliers. RTL synthesis was performed at 750MHz for 28nm silicon technology with area and power measurements recorded for each number representation. Table 5.8 provides a subset of these results, with coverage of all BM formats supplied in Appendix B.3.

To summarize, BM8 and BM6 arithmetic units are 2.1×(12.2×) and 4.1×(23.9×) smaller and consume 1.25 × (8.8×) and 2.3 × (16.1×) less power than competitive FP8/(FP32) representations. Such arithmetic, which has similar hardware complexity to INT8, may be especially useful in embedded applications where there are stricter area and power constraints but training still needs to achieve normal levels of accuracy and relatively high performance. With high computational density, BM arithmetic can achieve higher training throughput on compute-intensive problems, while sub 8-bit BM formats have lower bandwidth requirements leading to faster training times in memory bound applications.

Table 5.8: Logic area and power of single-cycle fused multiply-Add (FMA) and 4x4 array multipliers. Synthesized at 750 MHz with Cadence RTL Compiler 14.11 and 28nm cell library

Component	Area (μm^2)	Power (μW)
FP32	4782	10051
FP8 (w/ FP16 add)	829	1429
INT8 (w/ INT32 add)	417	1269
BM8	391	1141
BM6	**200**	**624**
INT8 (4x4 systolic)	7005	20253
FP8 (4x4 systolic)	18201	56202
BM8 (4x4 systolic)	**6976**	**18765**

Finally, overheads related to conversion from Kulisch to floating-point and BM quantization are expected to contribute little logic area relative to GEMM. This includes modules for leading-one detection, barrel shifter, maximum exponent calculation, pre-quantization buffering and stochastic rounding, each of which has an efficient implementation. Further support of these claims and other system-level effects are the subject of future work.

5.6 Summary

A new representation called Block Minifloat (BM) was presented for training DNNs effectively with reduced precision. The representation allows the implicit exponent bias within IEEE-754 floating-point specifications to vary for a block of numbers and can be trained with high accuracy using narrow exponent encodings. This chapter describes how few exponent bits lead to significantly smaller hardware, while smaller representations reduce memory bandwidth requirements, leading to potentially faster training than previous 8-bit approaches.

Chapter 6

Conclusion

This thesis aimed to demonstrate the importance of hardware specialization for improving the speed and energy efficiency of training machine learning models. Based on quantitative and qualitative analyses of kernel methods and DNN training algorithms, including performance modelling and hardware simulations using FPGAs, it can be concluded that both the underlying hardware architecture (Chapter 3) and computer arithmetic (Chapter 4 and Chapter 5) are important factors to consider when designing performant machine learning systems. Results indicate that specialization in these two areas can significantly improve the scalability and accuracy of on-chip training:

First, a high-speed systolic array architecture was described for real-time training of approximate kernel methods in fixed-point, and it was shown that a design based on minimising off-chip memory transfers and maximising data reuse is effective at increasing the on-chip model capacity by several orders of magnitude.

The limitations of DNN training in fixed-point were then discussed, and instead, a specialized representation called block floating-point (BFP) was proposed. It was then shown through the implementation of a prototype that high accuracy could be achieved using mostly low-precision BFP arithmetic. However, performance of the system could be improved with greater hardware specialization to increase parallelism and minimise off-chip memory transfers.

Finally, in order to train DNNs with further reduced precision and higher compute density, a highly specialized number representation called Block Minifloat (BM) was introduced, and effective training was demonstrated on a wide variety of benchmark models and datasets.

Putting the main results together, specialized arithmetic is useful for training large models and achieving the same accuracy with less memory, whereas specialized architectures are needed for keeping the model on-chip, reducing expensive memory transfers, and

implementing denser compute units. With the growing adoption of machine learning and the growing need for hardware acceleration, these insights should encourage computer architects and machine learning experts to work on hardware specializations together. It is likely that such collaborations will produce significant performance gains for on-chip training, and pave a path towards greater intelligence in computationally constrained devices.

6.1 Future Work

Moving forward, there are still a number of research directions for hardware specialization and on-chip training. While this thesis considered the hardware architecture and computer arithmetic as separate targets for optimization, not much work has been done to combine both specializations into the same FPGA design. An implementation here would substantially improve the performance of the prototype given in Chapter 4, which can benefit from extra parallelism in the systolic array, greater flexibility in the supported dataflow, and more functionality on-chip.

The algorithmic techniques used to efficiently approximate large-scale kernel methods in Chapter 3 could have potential for practical application in training DNNs. While the hardware architecture presented in this thesis was specialized for the Fastfood algorithm, similar design strategies, as well as other more specialized techniques, could be applied for keeping the entire DNN model on-chip during training. Further investigation would need to focus on preserving the accuracy of standard SGD training, while in terms of an implementation, massive performance gains could be expected over a system reliant on off-chip DRAM transfers.

Training with Block Minifloat representations can be expanded and further improved by considering the problem of multiple minifloat training, which selects the BM format for each input dynamically. While BM arithmetic units can already be modified to support multiple formats in hardware, further work is required to decide how and when to switch representations during training. An approach based on heuristics is the likely first step, and could potentially yield a more robust and well-tuned reduced precision training algorithm.

Abbreviations

ASIC	Application Specific Integrated Circuit
BFP	Block Floating Point
BM	Block Minifloat
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
CPU	Central Processing Unit
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
FF	Flip-Flop
FMA	Fused Multiply-Add
FP	Floating-Point
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FWHT	Fast Walsh Hadamard Transform
GEMM	Generalised Matrix Multiplication
GPU	Graphics Processing Unit
HB	Hadamard Block
IEEE	Institute of Electrical and Electronics Engineers
LUT	Look-Up Table
MAC	Multiply-Accumulate
MSE	Mean Squared Error
PCI-e	Peripheral Component Interconnect express
PE	Processing Element
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RF	Radio Frequency
RTL	Register-Transfer Level
SGD	Stochastic Gradient Descent
SoC	System on Chip
SWALP	Stochastic Weight Averaging for Low-Precision Gradient Descent

Appendix A

Basic Operations

A.1 Generalised Matrix Multiplication

The dominant computation in machine learning is generalised matrix multiplication (GEMM). Consider the example below for $C_{n\times k} = A_{n\times m} \times B_{m\times k}$, where $n = 2$, $m = 4$ and $k = 2$.

$$C = A \times B$$

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix}$$

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} + a_{24}b_{41} \\ c_{22} &= a_{21}b_{12} + a_{21}b_{22} + a_{23}b_{32} + a_{24}b_{42} \end{aligned}$$

For output C_{ij}, the result is calculated by computing the dot-product of row i in A and column j in B. When $n = m = k = N$, the number of multiply-accumulate operations scales as $O(N^3)$.

A.2 Fast Walsh Hadamard Transform

The Fast Walsh Hadamard Transform (FWHT) is an efficient algorithm for computing the Hadamard transform and is used to speed up the implementation of approximate kernel methods in Chapter 3. For an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_d]$, the Hadamard transform is calculated as the matrix product of $\mathbf{x}$ and the Hadamard matrix $\mathbf{H}$, defined below:

$$\mathbf{H} = \mathbf{H}_d = \begin{bmatrix} \mathbf{H}_{d/2} & \mathbf{H}_{d/2} \\ \mathbf{H}_{d/2} & -\mathbf{H}_{d/2} \end{bmatrix} \textit{ where } \mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{A.1}$$

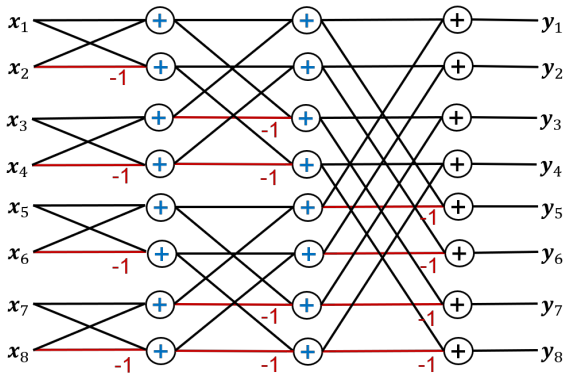


Figure A.1: The Fast Walsh Hadamard Transform applied to a vector of length d=8

The radix-2 FWHT algorithm works by breaking a size d Hadamard transform (i.e. $\mathbf{H}_d\mathbf{x}$) into sets of $d/2$ Hadamard transforms recursively. This creates a butterfly shaped dataflow diagram, as shown in Figure A.1, and reduces the number of operations from d^2 to $d \log_2 d$ but requires that d is a power of 2. An example is provided below for $d = 4$:

$$\mathbf{y} = \mathbf{H}_d\mathbf{x}$$

$$\mathbf{y} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

$$y_1 = (x_1 + x_2) + (x_3 + x_4)$$

$$y_2 = (x_1 - x_2) + (x_3 - x_4)$$

$$y_3 = (x_1 + x_2) - (x_3 + x_4)$$

$$y_4 = (x_1 - x_2) - (x_3 - x_4)$$

A.3 Convolution

This section describes the computation of convolution layers in deep neural networks (DNNs). Convolution layers are widely used in image and video processing tasks for spatial feature extraction.

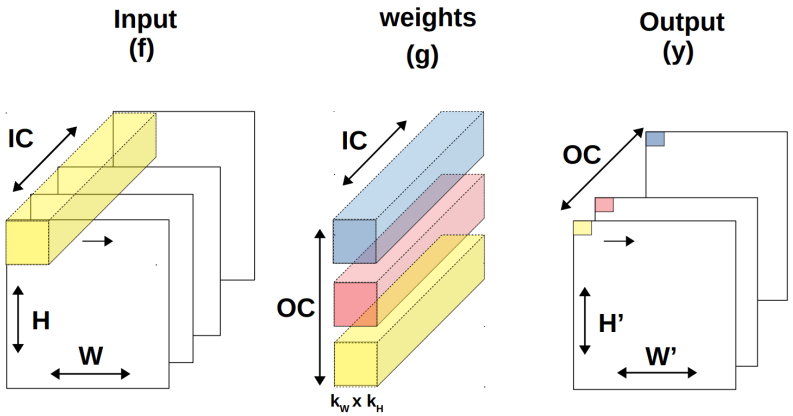
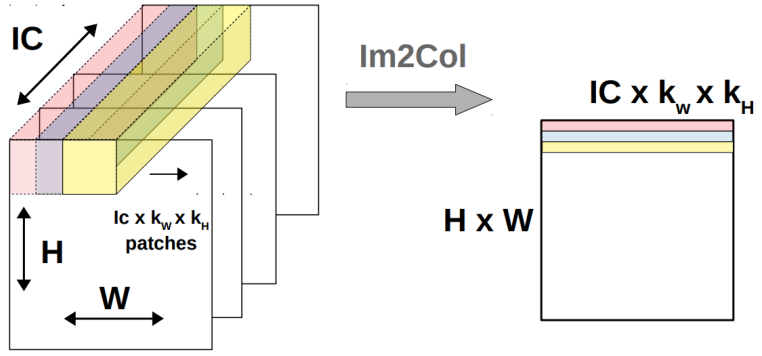


Figure A.2: Input, weight and output dimensions of convolution layer for a single example image, i.e. B=1

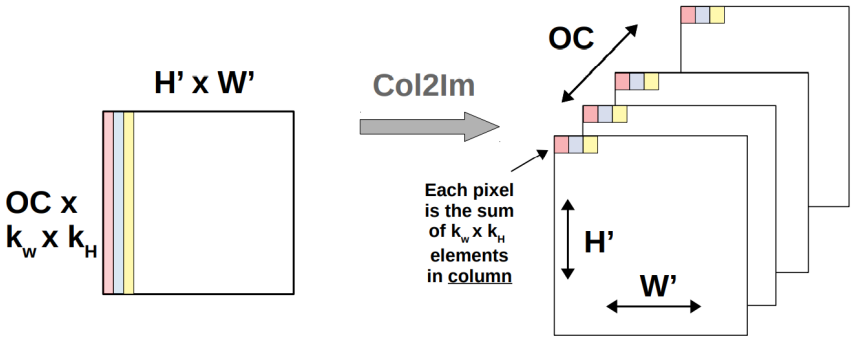
The input tensor **f** refers to a batch of images with dimensions (B, W, H, IC), corresponding to batch size, width, height and number of input channels. The outputs are computed by applying a convolution operator **g** to the input with typically four dimensions (k_W, k_H, IC, OC), which refer to the width and height of the operator and number of input and output channels. The convolution operator computes dot products over local regions of the input, moving over the image with an associated stride, and forming outputs **y** with shape (B, W', H', OC). This is shown in Figure A.2 and formalised mathematically in Equation (A.3) for an output pixel indexed by (b, i, j, o) and stride=1.

$$\mathbf{y} = \mathbf{f} * \mathbf{g} \tag{A.2}$$

$$\mathbf{y}[b][i][j][o] = \sum_{p=0}^{IC-1} \sum_{n=0}^{k_W-1} \sum_{m=0}^{k_H-1} \mathbf{f}[b][i+n-1][j+m][p] \times \mathbf{g}[n][m][p][o] \tag{A.3}$$



(a) im2Col



(b) col2Im

Figure A.3: Image-to-Column (im2col) and Column-to-Image (col2im) Transforms

To compute convolution efficiently with standard hardware architectures, the operation is usually reformatted into a matrix multiplication to improve memory utilisation and computational efficiency. Two transforms are used, called Image-to-Column (im2col) and Column-to-Image (col2im), which change the layout of the input from an image to 2D matrix, and the output from a 2D matrix back to an image. An illustration of both transforms is given in Figure A.3, where the input matrix is constructed from patches of the image sized $k_W \times k_H \times IC$, and OC pixels in the output image are formed by summing every $k_W \times k_H$ elements in a given column.

The implementation of im2col and col2im were identified as the main performance bottlenecks in Chapter 4 as both transforms increase memory traffic by $(k_W \times k_H)\times$, while col2im introduces an additional $(k_W \times k_H)$ operations for each output pixel.

Appendix B

Supplementary Material: Block Minifloat

B.1 Software Overview

Block Minifloat (BM) arithmetic requires custom hardware to achieve gains in speed and energy efficiency. QPyTorch [104], an open-source framework for low-precision training, is used to simulate the behaviour of BM hardware with existing PyTorch and CUDA libraries. QPyTorch provides a simple interface for applying quantization in the forward path (weight and activation tensors) and backward path (error, gradient and momentum tensors), ensuring that all numbers have a low-precision representation, while the actual GEMM and AXPY operations are computed in single-precision floating-point (FP32). This last point means that QPyTorch can not ordinarily be used to research low-precision accumulation strategies. BM in this thesis is different. Kulisch accumulators (as described in Section 5.2.3) compute exact dot products, and therefore GEMM operations are adequately approximated by FP32 arithmetic (which is close to exact). QPyTorch is available online [1] and supports floating-point (without denormals), fixed point and block floating-point number formats. The code implementation of BM [2] is an extension of this package.

[1] Implementation available at https://github.com/Tiiiger/QPyTorch
[2] Available at https://github.com/sfox14/block_minifloat

B.2 Experiment Details

Transformer Model (WMT)

The Transformer Base model from the FairSeq [3] repository was trained on the IWSLT'14 German to English translation task. Adam optimizer was used and the FairSeq implementation was modified with BM8 quantization. The network was trained for 25 epochs using the default parameters found in the repository. BLEU scores were calculated using the script from the repository and show similar convergence between BM8 and FP32 models.

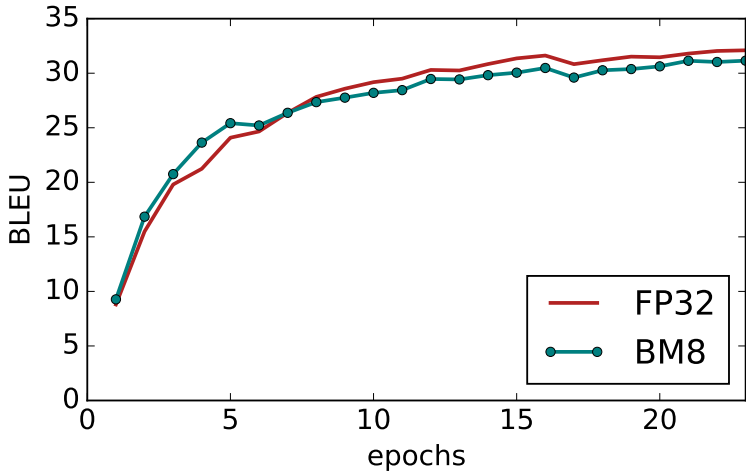


Figure B.1: Training convergence curves for Transformer on IWSLT'14 DE-En dataset

SSD-Lite (MobileNet-V2) (VOC)

The PyTorch implementation of SSD-lite was adapted from an online repository [4]. The base network is MobileNet-V2 [105] which was pre-trained on ImageNet. The entire network is trained on VOC2012 and VOC2007 trainval datasets and evaluated on the VOC2007 validation dataset. BM8 quantization was applied to all weights, activations and gradients before GEMM computations in the forward and backward paths. The network was trained with default parameter settings provided in the repository as follows: SGD with momentum of 0.9, weight decay factor 0.0005, batches of 32 images, and cosine annealing (t_{max} = 200) with an initial learning rate of 0.01. After 200 epochs, BM8 achieves an mAP of 68.0 which is sufficiently close to the reported score of 68.6.

[3] Implementation available at https://github.com/pytorch/fairseq
[4] Implementation available at https://github.com/qfgaohao/pytorch-ssd

EfficientNet-b6 (ImageNet)

The PyTorch implementation of EfficientNet [101] was adapted from an online repository [5]. The smallest EfficientNet-b0 network was trained on a reduced sized ImageNet dataset, where the images are resized from 256x256 to 128x128. This choice was made in order to reduce the training time, which is slowed down by 5× using the BM8 quantization function. The network is trained on one GPU for only 60 epochs using batch size 256 and an initial learning rate of 0.1 which is decayed exponentially with gamma of 0.90387. Figure B.2 shows the convergence of BM8 with an FP32 baseline.

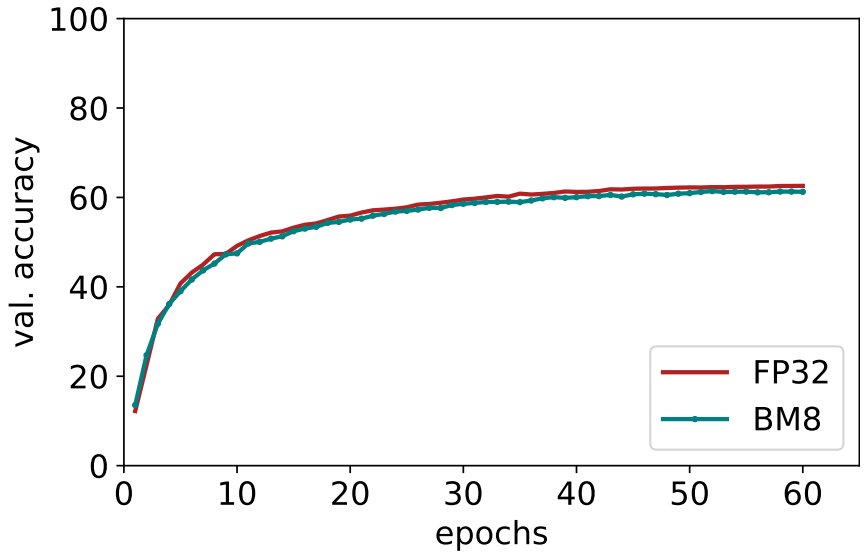


Figure B.2: Training convergence curves for EfficientNet-b0 on ImageNet

[5] Implementation available at https://github.com/narumiruna/efficientnet-pytorch

B.3 Hardware Synthesis

Fused multiply-add (FMA) units were designed in RTL for floating-point and block minifloat representations. Code was modified from the Deepfloat[6] repository for FP32, FP16 and FP8 units. The BM units with Kulisch accumulation were handwritten in Verilog following the block design given in Figure B.3. All designs were synthesised at 750 MHz using Cadence RTL compiler 14.11 and a commercial 28nm standard cell library. Since GEMM hardware is typically designed from tiles of smaller computational units, synthesis results are also provided for small 4x4 systolic array multipliers. Full coverage of results is shown in Table B.1 and Table B.2. Table B.3 is also provided and shows the component breakdown and scaling of Kulisch related costs in different 8-bit BM regimes. Given that BM relies on hybrid formats, the multiplier operands are both sized for the largest mantissa plus one bit for denormal support. Compared to (4,3)/(5,2), which is the format used in HFP8 [65], BM8 (2,5)/(4,3) is 1.6× smaller. This is because BM8 has narrower exponent encodings that reduce the width of the Kulisch accumulator.

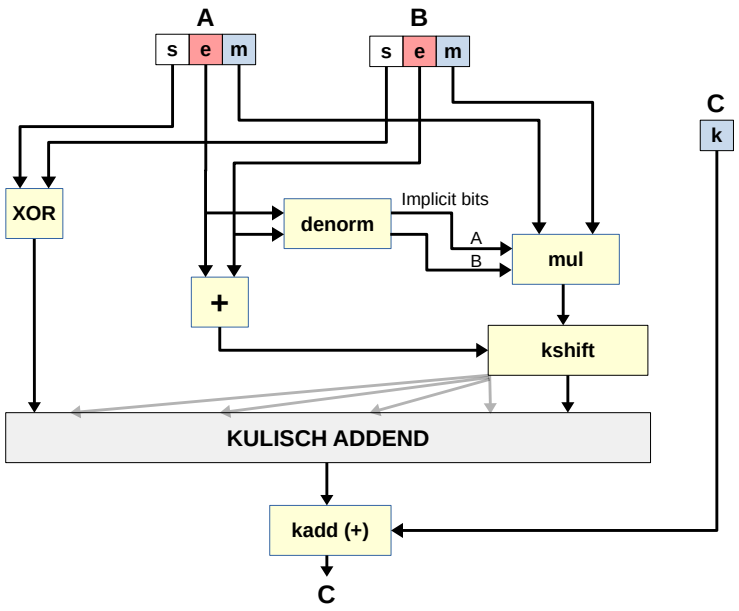


Figure B.3: Block diagram of block minifloat multiply-add; A*B + C, where A and B are minifloats and C is an integer

[6]Implementation available at https://github.com/facebookresearch/deepfloat

Table B.1: Synthesized logic area and power of single-cycle fused multiply-Add (FMA) at 750 MHz on 28nm chip.

Component	Area μm^2	Power μW
FP32	4782	10051
FP16	1116	2120
FP8 (w/ FP16 add)	829	1429
INT8 (w/ INT32 add)	417	1269
BM8	391	1141
BM7	280	840
BM6	200	624
BM5	171	546
BM5-log	231	801
BM4	115	361
BM4-log	120	426

Table B.2: Synthesized logic area and power of 4x4 systolic array multipliers at 750 MHz on 28nm chip.

Component	Area μm^2	Power μW
FP8 (w/ FP16 add)	18201	56202
INT8 (w/ INT32 add)	7005	20253
BM8	6976	18765
BM6	4083	11959

Table B.3: Component breakdown and logic area for different 8-bit BM formats

Format	Details			Area (μm^2)			
$(e,m)/(e,m)$	multiply	kadd	kshift	comb.[1]	kadd	kshift	**total**
$(3,4)/(4,3)$	$(5b \times 5b)$	34	24	210	93	74	377
$(2,5)/(4,3)$	$(6b \times 6b)$	31	20	235	79	77	391
$(3,4)/(5,2)$	$(5b \times 5b)$	49	40	253	199	95	547
$(4,3)/(5,2)$	$(4b \times 4b)$	56	48	259	276	104	639
$(5,2)/(5,2)$	$(3b \times 3b)$	71	64	300	361	113	774
$(0,7)/(0,7)$	$(8b \times 8b)$	32	NA	NA	NA	NA	418

[1] combinational logic includes multiply component

Bibliography

[1] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, vol. 37, no. 3, pp. 362–386, 2020.

[2] A. Graves, A. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*, IEEE, 2013, pp. 6645–6649.

[3] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean, "A guide to deep learning in healthcare," *Nature medicine*, vol. 25, no. 1, pp. 24–29, 2019.

[4] D. Ravì, C. Wong, F. Deligianni, M. Berthelot, J. Andreu-Perez, B. Lo, and G.-Z. Yang, "Deep learning for health informatics," *IEEE journal of biomedical and health informatics*, vol. 21, no. 1, pp. 4–21, 2016.

[5] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, *et al.*, "Wide & deep learning for recommender systems," in *Proceedings of the 1st workshop on deep learning for recommender systems*, 2016, pp. 7–10.

[6] K. Tunyasuvunakool, J. Adler, Z. Wu, T. Green, M. Zielinski, A. Žídek, A. Bridgland, A. Cowie, C. Meyer, A. Laydon, *et al.*, "Highly accurate protein structure prediction for the human proteome," *Nature*, pp. 1–9, 2021.

[7] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[8] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *CoRR*, vol. abs/1909.08053, 2019. arXiv: 1909.08053. [Online]. Available: http://arxiv.org/abs/1909.08053.

[9] R. Schaller, "Moore's law: Past, present and future," *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[10] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[11] M. S. Abdelfattah, D. Han, A. Bitar, R. DiCecco, S. O'Connell, N. Shanker, J. Chu, I. Prins, J. Fender, A. C. Ling, *et al.*, "DLA: Compiler and FPGA overlay for neural network inference acceleration," in *2018 28th International Conference on Field Programmable Logic and Applications*, IEEE, 2018, pp. 411–4117.

[12] H.-T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 01, pp. 37–46, 1982.

[13] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *CoRR*, vol. abs/2102.00554, 2021. arXiv: 2102.00554. [Online]. Available: https://arxiv.org/abs/2102.00554.

[14] T. Dao, A. Gu, M. Eichhorn, A. Rudra, and C. Ré, "Learning fast algorithms for linear transforms using butterfly factorizations," in *International conference on machine learning*, PMLR, 2019, pp. 1517–1527.

[15] Z. Yang, A. Wilson, A. Smola, and L. Song, "A la carte–learning fast kernels," in *Artificial Intelligence and Statistics*, PMLR, 2015, pp. 1098–1106.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[17] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, IEEE, 2014, pp. 10–14.

[18] W. J. Dally, "High-performance hardware for machine learning," *NIPS Tutorial*, vol. 2, 2015. [Online]. Available: https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf.

[19] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015. [Online]. Available: https://arxiv.org/abs/1510.00149.

[20] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof, *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Advances in neural information processing systems*, 2017, pp. 1742–1752.

[21] G. Yang, T. Zhang, P. Kirichenko, J. Bai, A. G. Wilson, and C. De Sa, "SWALP: Stochastic weight averaging in low precision training," in *International Conference on Machine Learning*, 2019, pp. 7015–7024.

[22] M. Drumond, L. Tao, M. Jaggi, and B. Falsafi, "Training dnns with hybrid block floating point," in *Advances in Neural Information Processing Systems*, 2018, pp. 453–463.

[23] T. M. Mitchell, *Machine learning*. McGraw-hill New York, NY, USA, 1997.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[26] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[27] J. Shawe-Taylor, N. Cristianini, *et al.*, *Kernel methods for pattern analysis*. Cambridge university press, 2004.

[28] W. Liu, J. C. Principe, and S. Haykin, *Kernel Adaptive Filtering: A Comprehensive Introduction*, 1st. Wiley Publishing, 2010.

[29] R. Collobert and S. Bengio, "SVMTorch: Support vector machines for large-scale regression problems," *The Journal of Machine Learning Research*, vol. 1, no. Feb, pp. 143–160, 2001.

[30] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[31] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in *Advances in Neural Information Processing Systems*, 2007, pp. 1177–1184.

[32] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," University of Toronto, Toronto, Ontario, Tech. Rep. 0, 2009.

[33] Q. Le, T. Sarlós, and A. Smola, "Fastfood-approximating kernel expansions in loglinear time," in *International Conference on Machine Learning*, vol. 85, 2013.

[34] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[35] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of machine learning accelerators," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2020, pp. 1–12.

[36] L. Crockett, D. Northcote, C. Ramsay, F. Robinson, and R. Stewart, *Exploring Zynq MPSoC: With PYNQ and machine learning applications*. Strathclyde Academic Media, 2019.

[37] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[38] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[39] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[40] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for deep-submicron FPGAs*. Springer Science & Business Media, 2012, vol. 497.

[41] M. J. S. Smith, *Application-specific integrated circuits*. Addison-Wesley Reading, MA, 1997, vol. 7.

[42] E. Wu, X. Zhang, D. Berman, and I. Cho, "A high-throughput reconfigurable processing array for neural networks," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2017, pp. 1–4.

[43] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2010, vol. 2.

[44] IEEE-754, *IEEE 754-2019, Standard for Floating-Point Arithmetic*. Jun. 2019, p. 82, ISBN: 1-5044-5925-3 (print), 1-5044-5924-5 (e-PDF). DOI: https://doi.org/10.1109/IEEESTD.2019.876622.

[45] D. D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A study of BFLOAT16 for deep learning training," *CoRR*, vol. abs/1905.12322, 2019. arXiv: 1905.12322. [Online]. Available: http://arxiv.org/abs/1905.12322.

[46] Xilinx Inc., *"Performance and resource utilization for floating-point v7.1."*. [Online]. Available: https://www.xilinx.com/html_docs/ip_docs/pru_files/floating-point.html (visited on 06/25/2021).

[47] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of floating-point arithmetic*. Springer Science & Business Media, 2009.

[48] S. Fox, D. Boland, and P. Leong, "FPGA fastfood - A high speed systolic implementation of a large scale online kernel method," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 279–284.

[49] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, 2015, pp. 3123–3131.

[50] N. J. Fraser, D. J. Moss, J. Lee, S. Tridgell, C. T. Jin, and P. H. Leong, "A fully pipelined kernel normalised least mean squares processor for accelerated parameter optimisation," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2015, pp. 1–6.

[51] S. Tridgell, D. J. Moss, N. J. Fraser, and P. H. Leong, "Braiding: A scheme for resolving hazards in kernel adaptive filters," in *2015 International Conference on Field Programmable Technology (FPT)*, IEEE, 2015, pp. 136–143.

[52] Y. Pang, S. Wang, Y. Peng, N. J. Fraser, and P. H. Leong, "A low latency kernel recursive least squares processor using fpga technology," in *2013 International Conference on Field-Programmable Technology (FPT)*, IEEE, 2013, pp. 144–151.

[53] M. Lichman, *UCI machine learning repository*, 2013. [Online]. Available: http://archive.ics.uci.edu/ml.

[54] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang, "An exploration of parameter redundancy in deep networks with circulant projections," in *Proc. ICCV*, 2015, pp. 2857–2865.

[55] M. Moczulski, M. Denil, J. Appleyard, and N. de Freitas, "ACDC: A structured efficient linear layer," in *4th International Conference on Learning Representations (ICLR)*, 2016. [Online]. Available: http://arxiv.org/abs/1511.05946.

[56] Z. Yang, M. Moczulski, M. Denil, N. De Freitas, A. Smola, L. Song, and Z. Wang, "Deep fried convnets," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1476–1483.

[57] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 2, pp. 1–33, 2012.

[58] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, IEEE, 2012, pp. 1212–1221.

[59] S. Van Vaerenbergh, *Kernel methods toolbox kafbox: A matlab benchmarking toolbox for kernel adaptive filtering*, 2012. [Online]. Available: http://sourceforge.net/projects/kafbox.

[60] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura, "Discrete Fourier transform on multicores: Algorithms and automatic implementation," *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, vol. 26, no. 6, pp. 90–102, 2009.

[61] S. Fox, J. Faraone, D. Boland, K. Vissers, and P. H. Leong, "Training deep neural networks in low-precision with high accuracy using fpgas," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2019, pp. 1–9.

[62] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[63] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016. arXiv: 1602.02830. [Online]. Available: http://arxiv.org/abs/1602.02830.

[64] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Advances in neural information processing systems*, 2018, pp. 7675–7684.

[65] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 4901–4910.

[66] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," in *Advances in neural information processing systems*, 2018, pp. 5145–5153.

[67] C. D. Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré, "High-accuracy low-precision training," *CoRR*, vol. abs/1803.03383, 2018. arXiv: 1803.03383. [Online]. Available: http://arxiv.org/abs/1803.03383.

[68] V. Sze, Y.-H. Chen, J. S. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," *CoRR*, vol. abs/1612.07625, 2016. arXiv: 1612.07625. [Online]. Available: http://arxiv.org/abs/1612.07625.

[69] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. 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 2017, Monterey, CA, USA, February 22-24, 2017*, 2017, pp. 65–74. [Online]. Available: http://dl.acm.org/citation.cfm?id=3021744.

[70] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, IEEE Press, vol. 44, 2016, pp. 367–379.

[71] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2017, pp. 75–84.

[72] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "Rebnet: Residual binarized neural network," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2018, pp. 57–64.

[73] S. Coric, I. Latinovic, and A. Pavasovic, "A neural network FPGA implementation," in *Proceedings of the 5th Seminar on Neural Network Applications in Electrical Engineering. NEUREL 2000 (IEEE Cat. No. 00EX287)*, IEEE, 2000, pp. 117–120.

[74] R. G. Gironés, R. C. Palero, J. C. Boluda, and A. S. Cortés, "FPGA implementation of a pipelined on-line backpropagation," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 40, no. 2, pp. 189–213, 2005.

[75] S. Siddhartha, S. Wilton, D. Boland, B. Flower, P. Blackmore, and P. Leong, "Simultaneous inference and training using on-FPGA weight perturbation techniques," in *2018 International Conference on Field-Programmable Technology (FPT)*, IEEE, 2018, pp. 306–309.

[76] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Patel, and M. Herbordt, "A framework for acceleration of cnn training on deeply-pipelined FPGA clusters with work and weight load balancing," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2018, pp. 394–3944.

[77] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-CNN: An FPGA-based framework for training convolutional neural networks," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 2016, pp. 107–114.

[78] C. Luo, M.-K. Sit, H. Fan, S. Liu, W. Luk, and C. Guo, "Towards efficient deep neural network training by FPGA-based batch-level parallelism," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2019, pp. 45–52.

[79] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015, pp. 1737–1746.

[80] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," *CoRR*, vol. abs/1805.11046, 2018. arXiv: 1805.11046. [Online]. Available: http://arxiv.org/abs/1805.11046.

[81] Z. Song, Z. Liu, and D. Wang, "Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[82] P. Izmailov, D. Podoprikhin, T. Garipov, D. P. Vetrov, and A. G. Wilson, "Averaging weights leads to wider optima and better generalization," *CoRR*, vol. abs/1803.05407, 2018. arXiv: 1803.05407. [Online]. Available: http://arxiv.org/abs/1803.05407.

[83] J. Redmon, *Darknet: Open source neural networks in C*, http://pjreddie.com/darknet/, 2016.

[84] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," *CoRR*, vol. abs/1804.02767, 2018. arXiv: 1804.02767. [Online]. Available: http://arxiv.org/abs/1804.02767.

[85] *Python productivity for zynq*. [Online]. Available: http://www.pynq.io/home.html.

[86] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. arXiv: 1409.1556. [Online]. Available: https://arxiv.org/abs/1409.1556.

[87] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[88] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European conference on computer vision*, Springer, 2016, pp. 630–645.

[89] S. Fox, S. Rasoulinezhad, J. Faraone, david boland, and P. Leong, "A block minifloat representation for training deep neural networks," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=6zaTwpNSsQ2.

[90] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016. arXiv: 1606.06160. [Online]. Available: http://arxiv.org/abs/1606.06160.

[91] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *CoRR*, vol. abs/1603.01025, 2016. arXiv: 1603.01025. [Online]. Available: http://arxiv.org/abs/1603.01025.

[92] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *CoRR*, vol. abs/1710.03740, 2017. arXiv: 1710.03740. [Online]. Available: http://arxiv.org/abs/1710.03740.

[93] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," *CoRR*, vol. abs/1802.04680, 2018. arXiv: 1802.04680. [Online]. Available: http://arxiv.org/abs/1802.04680.

[94] L. Cambier, A. Bhiwandiwalla, T. Gong, M. Nekuii, O. H. Elibol, and H. Tang, "Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks," *CoRR*, vol. abs/2001.05674, 2020. arXiv: 2001.05674. [Online]. Available: https://arxiv.org/abs/2001.05674.

[95] J. Johnson, "Rethinking floating point for deep learning," *CoRR*, vol. abs/1811.01721, 2018. arXiv: 1811.01721. [Online]. Available: http://arxiv.org/abs/1811.01721.

[96] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[97] U. W. Kulisch and W. L. Miranker, *Computer arithmetic in theory and practice*. Academic press, 2014.

[98] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[99] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, IEEE, 2009, pp. 248–255.

[100] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The Penn treebank," *Comput. Linguist.*, vol. 19, no. 2, pp. 313–330, Jun. 1993, ISSN: 0891-2017.

[101] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, PMLR, Sep. 2019, pp. 6105–6114. [Online]. Available: http://proceedings.mlr.press/v97/tan19a.html.

[102] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *European conference on computer vision*, Springer, 2016, pp. 21–37.

[103] M. Cettolo, J. Niehues, S. Stüker, L. Bentivogli, and M. Federico, "Report on the 11th IWSLT evaluation campaign, IWSLT 2014," in *Proceedings of the International Workshop on Spoken Language Translation, Hanoi, Vietnam*, vol. 57, 2014.

[104] T. Zhang, Z. Lin, G. Yang, and C. D. Sa, "QPyTorch: A low-precision arithmetic simulation framework," *CoRR*, vol. abs/1910.04540, 2019. arXiv: 1910.04540. [Online]. Available: http://arxiv.org/abs/1910.04540.

[105] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.