Block Arithmetic Techniques for the Implementation of Deep Neural Networks

WENJIE ZHOU

THE UNIVERSITY OF SYDNEY

Supervisor: Philip H.W. Leong
Associate Supervisor: David Boland

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

School of Electrical and Computer Engineering
Faculty of Engineering
The University of Sydney
Australia

5 February 2026

Abstract

The success of artificial intelligence (AI) models, especially deep neural network (DNN)s, has attracted tremendous investment in developing the latest AI models by private and government departments. Performance is crucial for the evolution of DNNs, particularly as computational requirements are surging. Along with increasing DNN model scale, the training cost is becoming a new problem for DNNs.

One of the critical techniques for energy-efficient training is low-precision arithmetic. Block arithmetic is a promising technique that reduces precision requirements and power consumption. This method further reduces the word length of the element, and the shared exponent expands their dynamic range.

This thesis aims to develop an improved block arithmetic algorithm and implementation methodology. At the arithmetic level, this work investigates the implementation of block arithmetic. At the general matrix multiplication (GEMM) kernel level, this dissertation examines kernel design under different block arithmetic implementations. For rescaling, this work further addresses the challenges associated with block arithmetic and introduces the proposed delayed scaling method called the delay update. At the application level, this work utilizes neural basis expansion analysis for interpretable time series analysis (N-BEATS) based inference and training accelerators to demonstrate the advantages of block arithmetic.

The contributions of this work are as follows: Firstly, we propose the block minifloat (BM) implementation for inference, the first implementation of an field programmable gate array (FPGA) based accelerator using BM arithmetic during publication, demonstrating hardware efficiency and accuracy benefits over integer and floating point on N-BEATS. Secondly, we propose the BM implementation for training, in the form of the first FPGA implementation of a 4-bit BM, mixed-precision neural network training of N-BEATS. Thirdly, we propose the delay update method to reduce the rescaling computation in block arithmetic. Empirical studies show that the delay update scheme achieves nearly the same accuracy as the commonly used maximum calibration method, with a significant hardware implementation advantage.

In summary, this dissertation shows how block arithmetic can benefit neural network inference and training. These research outcomes demonstrate that block arithmetic can offer the same accuracy as the contemporary approaches using integer and floating point, with lower precision leading to lower resource utilization.

Acknowledgements

I'm grateful to have finally completed the long and arduous journey of my PhD. The challenges I faced were not only academic, but also many circumstances beyond my control. The COVID-19 pandemic disrupted both research and daily life, while the rising cost of living added further difficulties. Overcoming these problems has made this accomplishment even more meaningful.

I want to express my gratitude to my supervisor, Philip Leong. In daily life, he always cares for his students and provides support. In research, he not only provided detailed, insightful academic feedback but also offered a visionary perspective that elevated the quality of my research. We often held different views but always agreed on one principle: pursuing practical, high-quality research.

I would also like to express my thanks to my father. He provided financial and emotional support throughout my doctoral studies. During these years, he also took on the responsibility of caring for my grandparents and managing the family finances, which allowed me to focus on my research without distraction.

Additionally, I would like to express my sincere gratitude to everyone who supported me throughout my PhD journey. Their encouragement and assistance have been invaluable in helping me reach this stage. Finally, this research reported in this thesis was supported by the award of a University of Sydney International Student (USYDIS) scholarship to the PhD Candidate.

Authorship Attribution Statement

This thesis includes material that has been previously published or prepared for publication during my PhD studies under the supervision of Professor Philip H.W. Leong. The ideas and development of each publication are primarily my own work, carried out under the guidance of my supervisors and with the assistance acknowledged below:

- Professor Philip H.W. Leong provided overall research direction and guidance throughout my PhD.
- **Chapter 3:** Professor Leong proposed the concept of implementing N-BEATS inference using BM. I developed the BM-based N-BEATS inference accelerator and introduced the integer-based block arithmetic implementation. Professor Leong and David Boland contributed to idea refinement, provided implementation guidance, and assisted in polishing the manuscript. Haoyan Qi supported the implementation of the host control system.
- **Chapter 4:** Professor Leong suggested the implementation of N-BEATS neural network training using BM. I designed the BM-based N-BEATS training accelerator and proposed the ideas. Professor Leong and David Boland helped refine the ideas, guided the implementation, and contributed to manuscript preparation. Haoyan Qi assisted with the host control system implementation.
- **Chapter 5:** Professor Leong introduced the concept of developing a delayed scaling algorithm. I proposed the delay update algorithm and conducted the GEMM kernel performance analysis. Professor Leong provided guidance and support throughout the development process.

Artificial Intelligence Statement

During the preparation of the thesis, the author used M365 Copilot for the purposes of text enhancement (e.g., spelling, sentence structure, grammar). The author confirms that where text was modified by generative AI, the content was reviewed for possible errors, inaccuracies, and bias. The author takes full responsibility for the submitted thesis and ensures the work is their own and has used generative AI within the parameters of use (refer to the University of Sydney generative AI guide for researchers).

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Motivation

In recent years, artificial intelligence (AI) has gained significant traction across a wide range of applications [7, 63, 95, 142]. The success of the AI models, especially deep neural network (DNN), has attracted tremendous investment in developing the latest AI models from private and government departments. In America, the Stargate project, led by private sectors such as OpenAI, Softbank, Oracle, and MGX, was announced to invest up to \$500 billion in the U.S. AI infrastructure, particularly data centers and power generation facilities [69, 136]. Simultaneously, in the Middle East, Saudi Arabia launched Humain, committing \$77 billion to build out graphics processing unit (GPU) clusters and other AI infrastructure, aimed at capturing 7% of global AI training capacity by 2030. The United Arab Emirates (UAE) is also mobilizing significant AI investments via funds such as MGX's \$100 billion asset pool, demonstrating the global scale of strategic AI infrastructure deployment [113].

Performance is crucial for the evolution of DNNs, particularly as computational requirements are also surging. For example, the number of parameters of GPT-3 in 2020 was 175 billion; now the new large language model (LLM)s have many more weight parameters [14]: DeepSeek-R1 has 671 billion, Grok has 314 billion, and Llama 3.3 has 405 billion. The most recent LLMs are estimated to have several trillion parameters (GPT-4 has 1.8 trillion and Claude-3 2 trillion) [33]. Along with increasing DNN model scale, the training cost is becoming a new problem for DNN. Reference [111] summarized the training cost of LLMs, and this is reproduced in Figure 1.1.

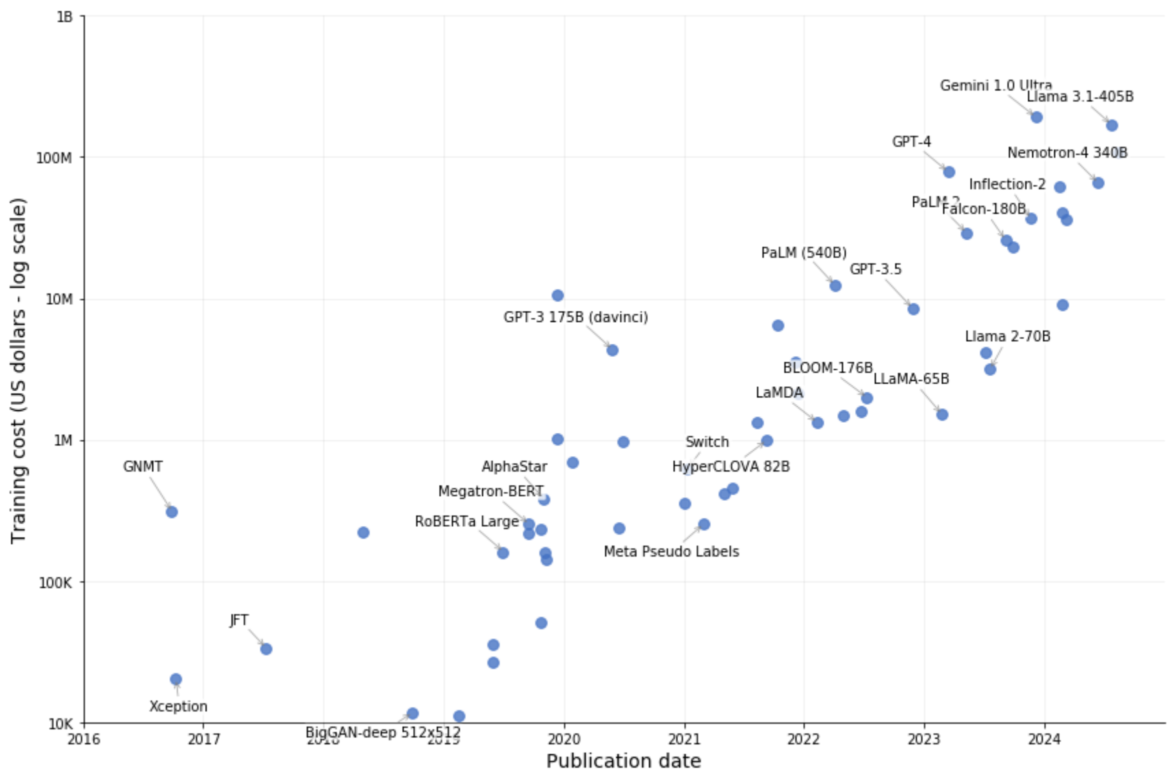


FIGURE 1.1: Estimated training cost of select AI models, 2016-24 (Source: Epoch AI, 2024) [111]

TABLE 1.1: Supported Precision of different NVIDIA GPU architectures [128, 168]

	Year	Cuda Core precisions	Tensor Core precisions
Maxwell	2014	FP64/32,INT8	No
Pascal	2016	FP64/32/16	No
Volta	2017	FP64/32/16, INT8	FP16
Ampere	2020	FP64/32/16, INT8, BF16	FP64/16, BF16, TF32, INT8/4/1
Hopper	2022	FP64/32/16, BF16, INT8	FP64/16/8, TF32, BF16, INT8
Blackwell	2024	FP64/32/16, BF16	FP64/16/8/6/4, TF32, BF16, INT8

A majority of the training cost is associated with the power consumption, including cooling [111]. The power consumption brings both economic and environmental impact, making the energy efficiency of the AI chip a critical factor [33]. One of the critical techniques for energy-efficient training is low-precision arithmetic, which can reduce the memory storage size requirements, memory I/O bandwidth, and increase the on-chip parallelism. Low-precision arithmetic is also one of the key innovations in the evolution of GPUs. As summarized in Table 1.1, the trend is to support more and more lower precision types.

Block arithmetic is a promising technique that reduces precision requirements and power consumption. Block arithmetic uses floating-point (FP) or integer format for its data, but the elements within a block have a shared exponent. This method further reduces the word length of the element, and the shared exponent expands their dynamic range. Block arithmetic includes arithmetic like block minifloat (BM), block floating-point (BFP), etc. More recently, block arithmetic has been adopted by the major computer manufacturers, including Microsoft, AMD, Intel, Meta, NVIDIA, and Qualcomm, in the form of the microscaling (MX) format [129, 144]. But some questions for block arithmetic remain:

- Block arithmetic typically operates under low-precision, which can amplify computational errors—particularly in matrix multiplication—thereby affecting training accuracy. In addition, block arithmetic can employ smaller block sizes. Accumulating partial sums across different small blocks may introduce additional rounding errors. Furthermore, to improve accuracy, different precision configurations for forward and backward propagation are often applied during training. What are the optimal arithmetic implementations under these conditions, particularly for matrix multiplication?
- Block arithmetic involves shared exponent calculation during matrix multiplication. Besides, the difference between block size and the general matrix multiplication (GEMM) kernel tile size introduces design challenges for the GEMM kernel. Furthermore, overlapping inner-product computation with rescaling or other blocks' computation latency is desirable. How can these challenges be addressed in GEMM kernel design?
- Rescaling requires computing the maximum absolute value to determine the block scale, followed by conversion to a low-precision format. This process introduces data dependencies that significantly increase latency and buffer requirements. How can these dependencies be broken at the algorithmic level to reduce latency?
- Given the advantages of block arithmetic, what minimum precision can be employed for training without compromising accuracy, and how can the block arithmetic training performance be maximized?

1.2 Aims and Contributions

This thesis aims to develop an improved block arithmetic algorithm and implementation methodology. The specific objectives are as follows:

- At the arithmetic level, propose a block arithmetic implementation that minimizes computational error and supports run-time configurable precision.
- At the GEMM kernel level, design a block arithmetic GEMM kernel and incorporate pipelining to overlap block arithmetic computation latency within the GEMM kernel operation or with other neural network computation blocks.
- For rescaling, propose a simplified and efficient algorithm that eliminates data dependencies, thereby reducing computation latency and improving hardware performance.
- At the accelerator level, use neural basis expansion analysis for interpretable time series analysis (N-BEATS) as an example, investigate the lower precision limits for training, and develop hardware–arithmetic co-optimization strategies to enhance training accelerator design.

The hardware implementation platform utilized in this thesis is field programmable gate array (FPGA), a reconfigurable architecture that offers significant flexibility. The arithmetic operations in other platforms—such as central processing unit (CPU), GPU, or application specific integrated circuit (ASIC)s—are fixed in the datapath post-fabrication. The flexibility of FPGAs enables the implementation of accelerators, making it an ideal platform for this research. The contributions of this thesis are summarized as follows:

BM for inference: This work presents the first implementation of an FPGA-based accelerator utilizing BM arithmetic during the period of publication. We introduce an integer-based BM arithmetic design to avoid rounding in normalization for inference acceleration. A novel accelerator architecture for N-BEATS is proposed based on a BM systolic array. Experimental results demonstrate that 8-bit BM achieves area and performance comparable to an 8-bit signed

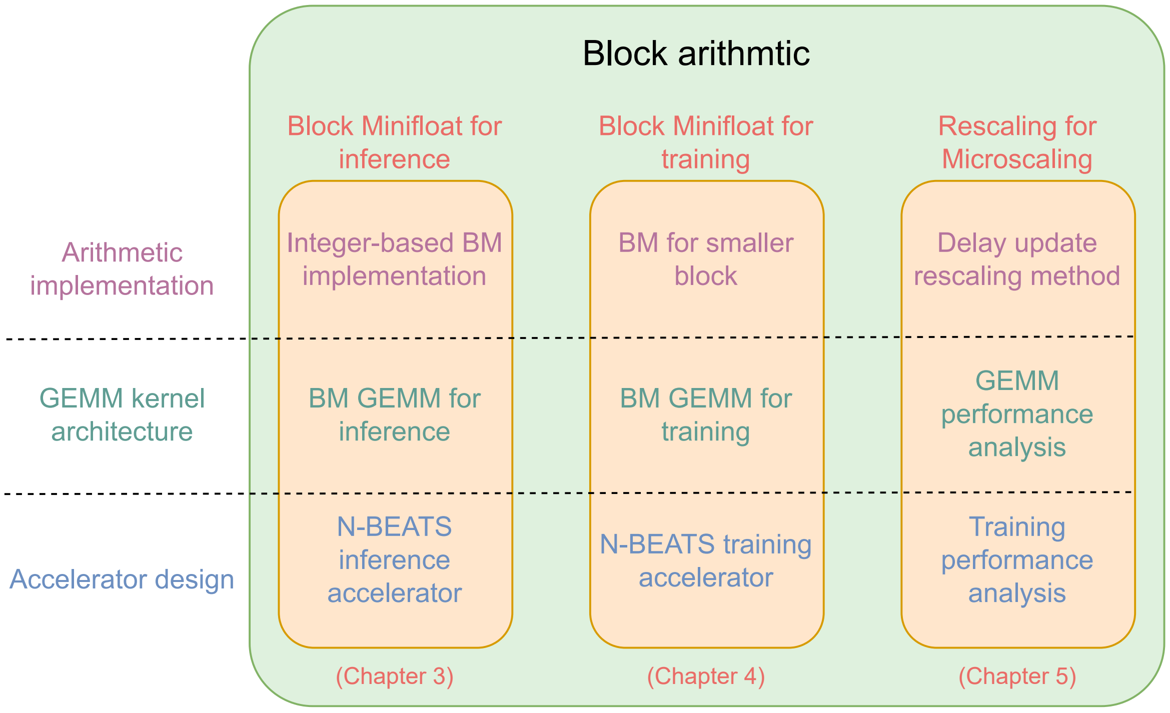

Block arithmtic
Block Minifloat for inference
Block Minifloat for training
Rescaling for Microscaling
Arithmetic implementation
Integer-based BM implementation
BM for smaller block
Delay update rescaling method
GEMM kernel architecture
BM GEMM for inference
BM GEMM for training
GEMM performance analysis
Accelerator design
N-BEATS inference accelerator
N-BEATS training accelerator
Training performance analysis
(Chapter 3)
(Chapter 4)
(Chapter 5)

FIGURE 1.2: The contribution and structure of the thesis.

integer (INT8) datapath, while maintaining accuracy levels similar to 16-bit floating-point (FP16) on N-BEATS [199].

BM for training: We propose a novel cross-block BM multiply accumulate (MAC) unit that supports independent block and tile sizes, and incorporates a high-precision buffer to enhance accuracy. In addition, we present a new BM GEMM kernel capable of run-time precision configuration, enabling optimal precision selection for forward and backward computations. Together, this is the first FPGA-based mixed-precision neural network training accelerator using 4-bit BM for N-BEATS.

Rescaling for MX: We analyze the impact of delayed scaling on GEMM performance and propose a delay update scheme. This method estimates the block scale using the maximum absolute value from the previous minibatch iteration. Empirical evaluations indicate that the delay update scheme achieves accuracy comparable to the maximum calibration method. Quantitative analysis of the performance of GEMM reveals significant hardware advantages, with training latency reductions by up to 40%.

1.3 Thesis Structure

The structure of the thesis is illustrated in Figure 1.2, and each chapter is outlined as follows:

- Chapter 2 provides background information on machine learning and DNN training, a review of relevant literature, and an overview of low-precision arithmetic techniques used in training.
- Chapter 3 presents the implementation of integer-based BM arithmetic, the design of BM GEMM for inference, and the development of an N-BEATS inference accelerator.
- Chapter 4 discusses the implementation of BM arithmetic for smaller blocks in training, the design of BM GEMM for training, and the construction of an N-BEATS training accelerator.
- Chapter 5 analyzes the performance of GEMM, introduces the delay update rescaling method, and evaluates training performance.
- Chapter 6 concludes the thesis and outlines potential directions for future research.

CHAPTER 2

Background

This chapter provides an introduction to deep learning and its hardware implementation. It includes a concise overview of foundational concepts in machine learning, DNN, and DNN training, as well as a discussion on low-precision arithmetic for training and hardware platforms for DNN acceleration.

2.1 Machine Learning

AI refers to the capability of computer systems to emulate human cognitive functions such as learning and problem-solving. Through AI, a system applies mathematical and logical reasoning to learn from new data and make informed decisions [107].

Machine learning is a subfield of AI that focuses on the development of algorithms capable of improving automatically through experience. A more precise definition of machine learning is provided in [119]:

DEFINITION 2.1.1. 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.

Task T refers to the specific learning objective that a machine learning algorithm aims to solve, such as image classification, speech recognition, or text generation. Experience E denotes the dataset that provides informative examples from which the algorithm can learn patterns or rules. The performance measure P defines the criteria used to evaluate the effectiveness and generalization capability of the algorithm.

TABLE 2.1: Examples of Machine Learning Tasks

Task (T)	**Experience (E)**	**Performance Measure (P)**
Image Classification	Labeled image datasets	Classification accuracy
Speech Recognition	Audio recordings with corresponding text labels	Accuracy of correctly recognized words
Time Series Prediction	Historical time series data from a specific domain	Forecast bias measured by metrics such as mean square error (MSE)
Text Translation	Parallel corpora in two languages	Phrase-level overlap between generated and reference text
Language Modeling	Text corpora	Predictive likelihood (e.g., perplexity)

For instance, in an image classification task, labeled datasets such as CIFAR-10 [94] and IMAGENET [32] serve as the experience E, while classification accuracy or the cross-entropy loss function are commonly used as performance measures P. A summary of representative machine learning tasks is provided in Table 2.1.

2.1.1 Types of Machine Learning

Machine learning algorithms can be categorized based on various criteria. One commonly used classification is based on the type of supervision involved during training [44]. According to this criterion, machine learning algorithms are typically divided into four categories: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning.

- **Supervised learning.** In supervised learning, the training data includes labeled examples that guide the algorithm toward generating correct outputs. The presence of labels enables the system to learn mappings between inputs and desired outputs. Common supervised learning algorithms include k-Nearest Neighbors [24], linear regression [60], logistic regression [60], support vector machine (SVM) [23], decision trees [140], random forests [67], and DNN [47]. A typical example is the image classification task [32], where labeled images are used during training. A *loss function* is employed to evaluate the model's performance; poor predictions result in higher loss values, prompting the algorithm to adjust its parameters accordingly.

- **Unsupervised learning.** In unsupervised learning, the training data is unlabeled. The algorithm attempts to discover hidden patterns or structures within the data. Representative unsupervised learning algorithms include clustering methods (e.g., K-means [109], hierarchical cluster analysis (HCA) [82]), dimensionality reduction techniques (e.g., principal component analysis (PCA) [83], Kernel PCA [147], locally linear embedding (LLE) [145], t-distributed stochastic neighbor embedding (t-SNE) [106]), and association rule learning algorithms (e.g., Apriori [6], Eclat [188]). A common example is clustering, where data points are grouped into clusters based on similarity, and new data is assigned to the most appropriate cluster.
- **Semi-supervised learning.** Supervised learning requires a large volume of labeled data, which can be costly and time-consuming to obtain. In contrast, unsupervised learning uses unlabeled data but often suffers from lower accuracy. Semi-supervised learning combines both approaches to mitigate their limitations. It typically involves a small amount of labeled data and a larger pool of unlabeled data. The algorithm initially learns from the labeled data and gradually incorporates the unlabeled data into the training process [149].
- **Reinforcement learning.** Reinforcement learning focuses on decision-making by autonomous agents [174]. During training, the agent interacts with an environment, performs actions, and receives feedback in the form of rewards or penalties. Through repeated interactions, the agent learns an optimal strategy—known as a *policy*—for selecting actions that maximize cumulative rewards. Unlike supervised learning, reinforcement learning does not rely on explicit human guidance or labeled data.

2.1.2 Supervised Learning System

A supervised learning system operates in two distinct phases: training and inference. The training phase involves learning generalizable patterns from data, while the inference phase uses the learned parameters to make predictions on previously unseen data.

The training process comprises several key components: the dataset, the model, and the learning algorithm. Typically, the dataset is partitioned into three subsets: training data, validation data, and test data. The training and validation sets are used during the learning phase, while the test set is reserved for evaluating the final model performance. The machine learning model is a computational framework designed to recognize patterns or make predictions, and it is trained by learning a set of parameters from the data [141]. The learning algorithm iteratively adjusts these parameters to improve prediction accuracy. It consists of an optimization algorithm and a loss function.

To illustrate the training process, consider a linear regression model applied to a time series prediction task. The first step is data preparation, where the dataset is randomly shuffled and divided into training, validation, and test subsets. The training data is used to learn the model, typically forming the majority of the dataset. During training, the model is periodically evaluated on the validation set to monitor overfitting. Overfitting occurs when the model performs well on training data but poorly on validation data. After training, the test set is used to assess the model's generalization performance.

In this example, let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\} \in \mathbb{R}^{N \times d_1}$ represent the historical input sequence of N examples, and $\hat{\mathbf{Y}} = \{\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2, ..., \hat{\mathbf{y}}_N\} \in \mathbb{R}^{N \times d_2}$ denote the corresponding forecast output sequence. Each input-output pair $(\mathbf{x}_i, \hat{\mathbf{y}}_i)$ is extracted from a continuous time-series segment of length $(d_1 + d_2)$, where the first d_1 elements form the input and the remaining d_2 elements form the output. The resulting datasets are denoted as $\{\mathbf{X}_{train}, \hat{\mathbf{Y}}_{train}\}$, $\{\mathbf{X}_{valid}, \hat{\mathbf{Y}}_{valid}\}$, and $\{\mathbf{X}_{test}, \hat{\mathbf{Y}}_{test}\}$.

The training process is summarized in algorithm 1. The model takes an input vector $\mathbf{x}_i$ and predicts an output vector $\mathbf{y}_i$ using a learnable weight matrix $\mathbf{w} \in \mathbb{R}^{d_2 \times d_1}$. After selecting the model hyperparameters d_1 and d_2, the linear regression model is defined as:

$$\mathbf{y} = f(\mathbf{x}) = \mathbf{w}\mathbf{x}^T \tag{2.1}$$

The mean square error (MSE) is commonly used as the loss function in linear regression. For the i-th training example, the MSE loss is defined as:

$$\mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \frac{1}{d_2} \|\hat{\mathbf{y}}_i - \mathbf{y}_i\|_2^2 = \frac{1}{d_2} \|\hat{\mathbf{y}}_i - f(\mathbf{x}_i)\|_2^2 \tag{2.2}$$

The loss function [156] quantifies the discrepancy between predicted outputs and ground truth labels. For validation and test data, performance metrics may be identical to the loss function or differ depending on the task. For example, in image classification, cross-entropy loss is used during training, while classification accuracy is used for validation and testing [32]. Common loss functions include MSE [64, 143], cross-entropy [34, 61, 153, 160], and KL divergence [25, 80, 190, 195].

The goal of training is to minimize the loss over the training dataset $\hat{\mathbf{Y}}_{train}$:

$$\min_{\mathbf{w}} \sum_{\hat{\mathbf{y}}_i \in \hat{\mathbf{Y}}_{train}} \mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i) \tag{2.3}$$

Generalization ability is evaluated on the validation dataset $\hat{\mathbf{Y}}_{valid}$:

$$\min_{\mathbf{w}} \sum_{\hat{\mathbf{y}}_i \in \hat{\mathbf{Y}}_{valid}} \mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i) \tag{2.4}$$

The optimization algorithm seeks the weight matrix $\mathbf{w}$ that minimizes the loss function (Equation 2.3). A widely used method is *gradient descent*, a first-order iterative algorithm. The first step involves computing the gradient of the loss with respect to the weights, followed by *weight update*:

$$\mathbf{w}^{(s+1)} = \mathbf{w}^{(s)} - \eta \frac{\partial \mathcal{L}_s}{\partial \mathbf{w}} \tag{2.5}$$

η is the learning rate, $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$ is the weight gradient. The gradient computation follows the backpropagation rule:

$$\frac{\partial \mathcal{L}_i}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}_i}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \mathbf{w}} = \frac{2}{d_2}\left|\hat{\mathbf{y}}_i - \mathbf{y}_i\right|^T \cdot \mathbf{x}_i \tag{2.6}$$

Algorithm 1: Learning process for linear regression model. Ep is the training epoch number.

```
// Initialize the weights w, prepare training dataset
    $\mathbf{X}_{train}$ and validation dataset $\mathbf{X}_{valid}$
for $Epoch \leftarrow 0$ to $Ep$ do
    // Training
    for $\boldsymbol{x}_i, \boldsymbol{y}_i$ in $\{\boldsymbol{X}_{train}, \hat{\boldsymbol{Y}}_{train}\}$ do
        Forward Computation $\mathbf{y}_i = f(\mathbf{x}_i)$; // Equation 2.1
        Training Loss $C_i^{train} = \mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i)$; // Equation 2.2
        Gradient Computation $\frac{\partial C_i}{\partial \mathbf{w}}$; // Equation 2.6
        Weight Update // Equation 2.5
    // Validation
    for $\boldsymbol{x}_i, \boldsymbol{y}_i$ in $\{\boldsymbol{X}_{valid}, \hat{\boldsymbol{Y}}_{valid}\}$ do
        Inference $\mathbf{y}_i = f(\mathbf{x}_i)$; // Equation 2.1
        Validation Loss $C_i^{valid} = \mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i)$; // Equation 2.2
    // Early Stopping
    $\bar{\mathbf{C}}_{Epoch} = mean(C_i^{valid})$ if $\bar{\mathbf{C}}_{Epoch}$ starts diverging then
        break;// Stop training
Return weight w
```

During each epoch of the training process, the weight parameters are first updated using the training dataset $\{\mathbf{X}_{train}, \hat{\mathbf{Y}}_{train}\}$. The updated weights and the model $f(\cdot)$ are then evaluated on the validation dataset $\{\mathbf{X}_{valid}, \hat{\mathbf{Y}}_{valid}\}$. If the validation loss decreases, the training continues until the final epoch Ep is reached. However, if the average validation loss $\bar{\mathbf{C}}_{Epoch}$ increases, indicating potential overfitting, the training process is terminated.

2.1.3 Deep Neural Network Training

A DNN is a neural network architecture that contains multiple hidden layers between the input and output layers. Commonly used models such as convolutional neural network (CNN) [99], long short-term memory (LSTM) [68], and Transformer [166] are all examples of DNNs.

The training process of a DNN is conceptually similar to the time series prediction example, but it employs a layer-wise computation structure, as illustrated in Figure 2.1. The forward pass, also referred to as inference, transforms the input data through successive layers to produce the output. The backward pass involves propagating the error from the output layer back toward the input layer, allowing the model to update its parameters based on the computed gradients. The gradient of each layer is dependent on the error propagated to that layer.

The overall training procedure of a DNN resembles the process described in algorithm 1, with the forward computation (Equation 2.1) replaced by the forward pass of the DNN, and the gradient computation (Equation 2.6) replaced by the backward pass of the DNN.

A widely used optimization algorithm for DNN training is the stochastic gradient descent (SGD) algorithm [66], which extends the basic gradient descent method used in linear regression. The SGD algorithm can be viewed as a stochastic approximation of gradient descent optimization [154]. Instead of computing gradients over the entire dataset, SGD estimates the gradient using a small subset of training examples $\{\mathbf{x}_i, \mathbf{x}_{i+1}, ..., \mathbf{x}_{i+B}\}$, referred to as a *minibatch*. The number of examples in a minibatch, denoted by B, is also known as the *batch size*. Using minibatches introduces stochasticity into the training process, which can have a regularizing effect and lead to more robust gradient estimates [47].

We illustrate the forward and backward passes of a DNN using an multilayer perceptron (MLP) as an example. The MLP is one of the foundational architectures in DNN research and applications [98]. It consists of multiple fully connected (FC) layers, each followed by a nonlinear activation function [120], as depicted in Figure 2.2. The FC layer is a fundamental building block not only in MLPs but also in various widely adopted models, including N-BEATS [130], the feedforward layers in Transformer architectures [166], and other neural components involving matrix multiplication, such as the LSTM cell [68] and self-attention mechanisms in Transformers [166].

Forward Pass

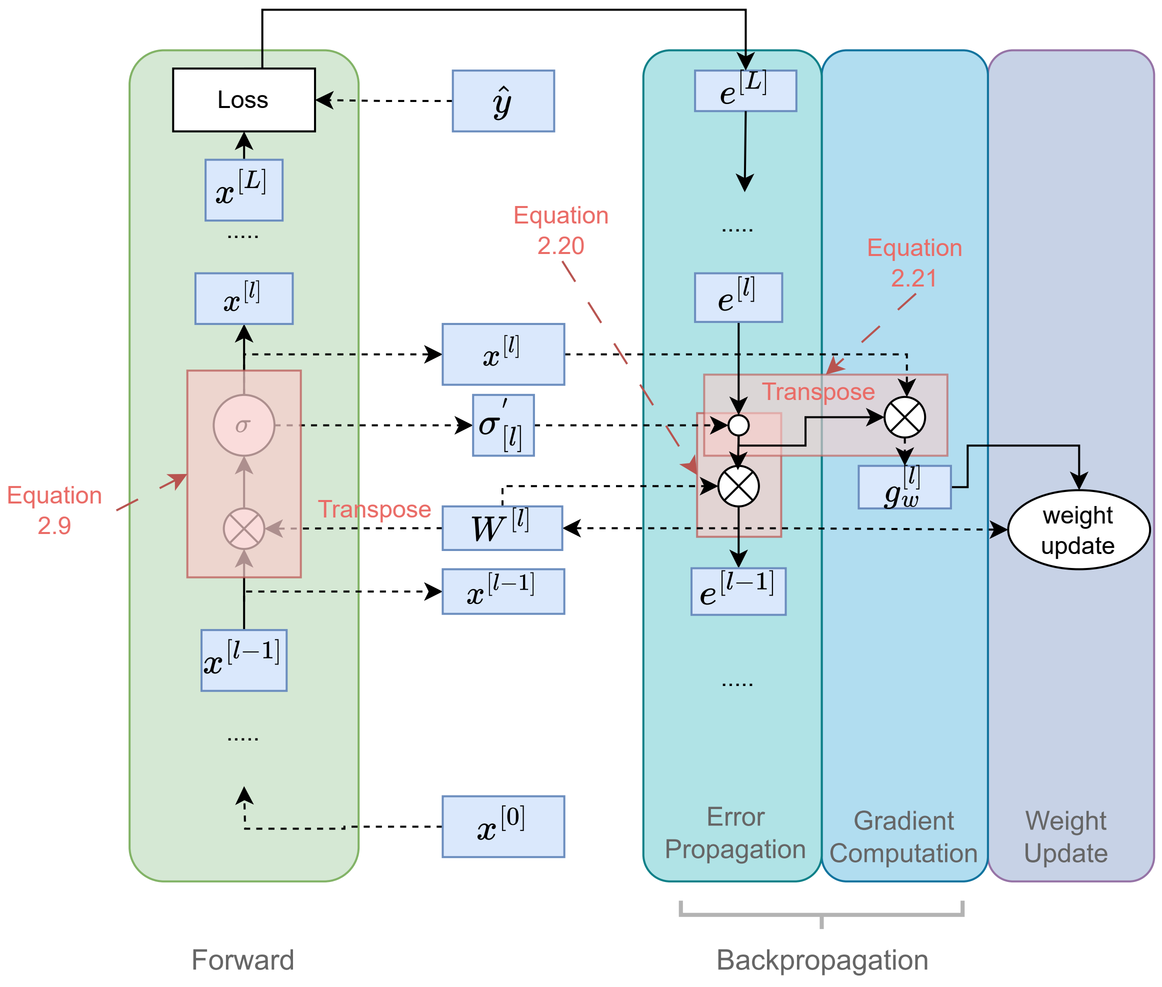


FIGURE 2.1: Forward and backward pass of the l-th fully connected (FC) layer in the multilayer perceptron (MLP) example. The left block illustrates the forward pass, while the three blocks on the right represent the backward pass. Solid lines indicate computation flows, and dashed lines denote data read/write operations to memory. Blue blocks represent data that must be stored in memory during forward and backpropagation. Computation blocks are highlighted with semi-transparent red overlays.

Figure 2.2(a) shows layer l with n input neurons. The neuron output $\mathbf{x}^{[l+1]}$ is the activation function σ output of intermediate value $\mathbf{z}^{[l]}$:

$$\mathbf{x}^{[l+1]} = \sigma(\mathbf{z}^{[l]}) \tag{2.7}$$

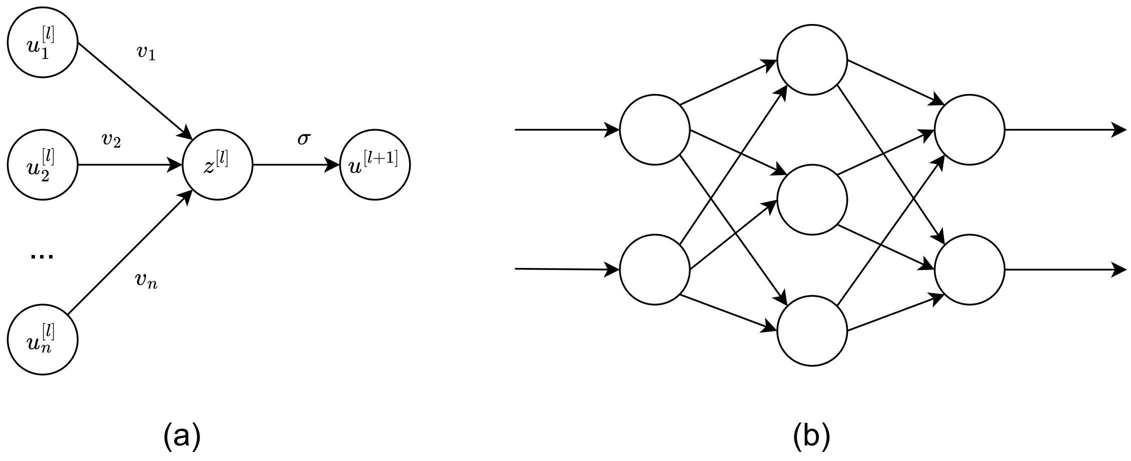


FIGURE 2.2: a MLP network example. (a) is a n input neuron. (b) is a three-layer MLP network.

where σ is the non-linear activation function, such as rectified linear unit (ReLu), Sigmoid, Tanh functions. $\mathbf{z}^{[l]}$ equals input vector $\mathbf{u} = [u_1, u_2, ..., u_n]$ dot product with the weight vector $\mathbf{v} = [v_1, v_2, ..., v_n]$:

$$\mathbf{z}^{[l]} = \mathbf{v}\mathbf{u}^T = \sum_{i=0}^{n} v_i u_i \tag{2.8}$$

A FC layer is composed of multiple neurons and produces multiple outputs for the next layer, as shown in Figure 2.2(b). The computation is a the matrix multiplication of the weight matrix $\mathbf{W}^{[l]} \in \mathbb{R}^{m \times n}$ with the $(l-1)$th layer input matrix $\mathbf{x}^{[l-1]} \in \mathbb{R}^{B \times n}$:

$$\mathbf{x}^{[l]} = f^{[l]}(\mathbf{x}^{[l-1]}) = \sigma(\mathbf{x}^{[l-1]}\mathbf{W}^{[l]^T} + \mathbf{b}_{repl}^{[l]}) \tag{2.9}$$

Here $\mathbf{b}_{repl}^{[l]} \in \mathbb{R}^{B \times m}$. $\mathbf{b}^{[l]} \in \mathbb{R}^{m}$ is the bias vector, $\mathbf{b}_{repl}^{[l]}$ is achieved by the replication of $\mathbf{b}^{[l]}$ B times in the first dimension. The input $\mathbf{x}^{[l-1]}$ is a minibatch of training examples with a batch size of B and each example length of n. And $\mathbf{z}^{[l]}$ in FC layer is:

$$\mathbf{z}^{[l]} = \mathbf{x}^{[l-1]}\mathbf{W}^{[l]^T} + \mathbf{b}_{repl}^{[l]} \tag{2.10}$$

Thus, a DNN with L layers can be written as:

$$D(\mathbf{x}^{[0]}) = f^{[L]}(...f^{[2]}(f^{[1]}(\mathbf{x}^{[0]}))) = \mathbf{x}^{[L]} \tag{2.11}$$

$\mathbf{x}^{[0]}$ is the input of the network, $\mathbf{x}^{[L]}$ is the DNN model output.

Backward Pass

Before computing the backward pass, the output $\mathbf{x}^{[L]}$ is compared to the label $\hat{\mathbf{y}}$ using the loss function $\mathcal{L}$:

$$C = \mathcal{L}(\mathbf{x}^{[L]}, \hat{\mathbf{y}}) \tag{2.12}$$

In the backward pass, the gradients of each layer are computed according to the backpropagation rule [47], and all weights are subsequently updated using SGD. Backpropagation is a gradient computation algorithm that propagates the error and the gradient of the loss function backward from the output layer toward the input layer by applying the chain rule, as illustrated in Figure 2.1.

In this context, we denote the activation gradient of the l-th layer as the error $\mathbf{e}^{[l]}$, and the gradients of the weight and bias parameters in the l-th layer as $\mathbf{g}_w^{[l]}$ and $\mathbf{g}_b^{[l]}$, respectively.

$$\mathbf{e}^{[l]} = \frac{\partial C}{\partial \mathbf{x}^{[l]}} \tag{2.13}$$

$$\mathbf{g}_w^{[l]} = \frac{\partial C}{\partial \mathbf{W}^{[l]}} \tag{2.14}$$

$$\mathbf{g}_b^{[l]} = \frac{\partial C}{\partial \mathbf{b}^{[l]}} \tag{2.15}$$

In backpropagation, the error in each layer is computed based on the error propagated from the subsequent layer, while the error in the final layer is derived from the cost function C. Specifically, we define the error at the output layer as $\frac{\partial C}{\partial \mathbf{x}^{[L]}} = \mathbf{e}^{[L]}$. Figure 2.3 illustrates the backpropagation process for a DNN layer.

Given the error $\mathbf{e}^{[l]}$ at the l-th layer, the corresponding gradients—$\mathbf{g}_w^{[l]} \in \mathbb{R}^{m \times n}$ for the weights, $\mathbf{g}_b^{[l]} \in \mathbb{R}^m$ for the biases—and the error propagated to the $(l-1)$-th layer, $\mathbf{e}^{[l-1]} \in \mathbb{R}^{B \times n}$, are computed as follows:

$$\mathbf{e}^{[l-1]} = \mathbf{e}^{[l]} \frac{\partial \mathbf{x}^{[l]}}{\partial \sigma} \frac{\partial \sigma}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{e}^{[l]}} \tag{2.16}$$

$$\mathbf{g}_w^{[l]} = \mathbf{e}^{[l]} \frac{\partial \mathbf{x}^{[l]}}{\partial \sigma} \frac{\partial \sigma}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{W}^{[l]}} \tag{2.17}$$

$$\mathbf{g}_b^{[l]} = \mathbf{e}^{[l]} \frac{\partial \mathbf{x}^{[l]}}{\partial \sigma} \frac{\partial \sigma}{\partial \mathbf{z}^{[l]}} \frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{b}^{[l]}} \tag{2.18}$$

$\frac{\partial \sigma}{\partial \mathbf{z}^{[l]}} = 1$, $\sigma'_{[l]} = \frac{\partial \mathbf{x}^{[l]}}{\partial \sigma}$ is the derivative of activation function. If the activation function is ReLu, then:

$$\sigma'_{[l]}(x_i) = \begin{cases} 1 & (x_i > 0) \\ 0 & (x_i \leq 0) \end{cases} \tag{2.19}$$

x_i is the ith element in $\mathbf{x}$. Thus, the error and gradient of lth FC layer are:

$$\mathbf{e}^{[l-1]} = (\mathbf{e}^{[l]} \circ \sigma'_{[l]})\mathbf{W}^{[l]} \tag{2.20}$$

$$\mathbf{g}_w^{[l]} = (\mathbf{e}^{[l]} \circ \sigma'_{[l]})^T \mathbf{x}^{[l]} \tag{2.21}$$

$$\mathbf{g}_b^{[l]} = sum((\mathbf{e}^{[l]} \circ \sigma'_{[l]}), 0) \tag{2.22}$$

We define $sum(X, d)$ as summing each row on the dth dimension of a tensor X. Once the gradient computation finishes, weights and bias will be updated with the SGD algorithm [66]:

$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \eta \mathbf{g}_w^{[l]} \tag{2.23}$$

$$\mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \eta \mathbf{g}_b^{[l]} \tag{2.24}$$

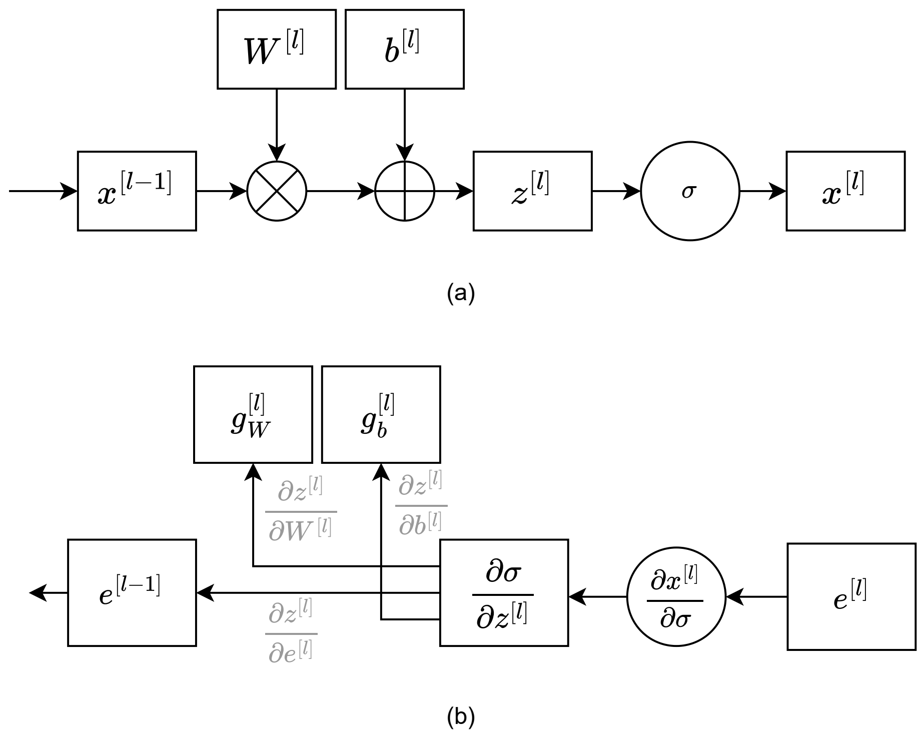

$W^{[l]}$
$b^{[l]}$
$x^{[l-1]}$
$z^{[l]}$
σ
$x^{[l]}$
(a)
$g_W^{[l]}$
$g_b^{[l]}$
$\frac{\partial z^{[l]}}{\partial W^{[l]}}$
$\frac{\partial z^{[l]}}{\partial b^{[l]}}$
$e^{[l-1]}$
$\frac{\partial \sigma}{\partial z^{[l]}}$
$\frac{\partial x^{[l]}}{\partial \sigma}$
$e^{[l]}$
$\frac{\partial z^{[l]}}{\partial e^{[l]}}$
(b)

FIGURE 2.3: The forward and backward pass of a FC layer. (a) is the forward pass (inference), (b) is the backward pass.

2.2 Literature overview

2.2.1 Challenge in Training Acceleration

Unlike inference, training involves both backpropagation and weight updates, making its acceleration fundamentally different from that of inference. First, training requires significantly more computation and memory operations. Second, these computational differences lead to distinct hardware optimization strategies for memory access and processing.

As illustrated in Figure 2.1, training differs from inference in several key aspects [52, 101]:

- **Computation.** The computational operations are highlighted in semi-transparent red blocks in Figure 2.1. In the FC layer, both forward and backward passes involve matrix multiplications. However, matrix transpositions are required in Equation 2.9

and Equation 2.20, which necessitate either non-unit stride memory access or matrix reorganization in memory. Additionally, different activation functions impose varying computational demands.

- **Data Dependency.** Beyond the layer-wise dependencies present in both forward and backward passes, training introduces additional dependencies. Within each training iteration, backpropagation can only commence after the forward pass and loss computation are complete. Similarly, the forward pass of the next iteration must wait until the current iteration's backpropagation finishes. Within the backward pass of the l-th layer, the weight update for $\mathbf{W}^{[l]}$ (Equation 2.23) depends on the completion of gradient computation (Equation 2.21), which in turn depends on the error propagation for $\mathbf{e}^{[l]}$ (Equation 2.20). These dependencies constrain the achievable parallelism.
- **Data Storage.** During the forward pass, in addition to storing the input/output activations of the l-th layer and weights $\{\mathbf{W}^{[1]}, ..., \mathbf{W}^{[L]}\}$, the intermediate activations $\{\mathbf{x}^{[0]}, ..., \mathbf{x}^{[L-1]}\}$ and activation derivatives $\{\sigma'_{[0]}, ..., \sigma'_{[L-1]}\}$ must also be stored for use in the backward pass. During backpropagation, both the error $\mathbf{e}^{[l]}$ and the gradient $\mathbf{g}_w^{[l]}$ must be stored. The memory requirements for storing intermediate results vary depending on the activation function used.
- **Data Movement.** In addition to standard data movement for layer-wise computation and error propagation, training introduces significantly more data transfer than inference. During the forward pass, $\mathbf{x}^{[l-1]}$ and $\sigma'_{[l]}$ must be written to memory for use in the backward pass. In the backward pass, the weight update step requires reading both the weight $\mathbf{W}^{[l]}$ and the gradient $\mathbf{g}_w^{[l]}$ from memory, followed by writing the updated weight back. In neural networks with FC layers, weights $\{\mathbf{W}^{[1]}, ..., \mathbf{W}^{[L]}\}$ are typically stored off-chip, necessitating high memory bandwidth to avoid bottlenecks.

Besides the challenges inherent in training, the optimization techniques used for training differ significantly from those used for inference. Methods such as low-precision arithmetic and sparsity are widely adopted in inference acceleration [16, 46, 57, 65, 70, 78, 100, 131, 151]. Many inference accelerators utilize fixed-point number systems due to their reduced hardware area and lower computation latency compared to FP. However, the limited dynamic range of fixed-point arithmetic is insufficient for training, where FP formats are often required [10, 114, 124].

In sparsity optimization, inference can benefit from weight pruning, which reduces the number of computations by creating sparse weight matrices. In contrast, training involves continuous weight updates in every iteration, making it difficult to maintain the fixed sparse weights throughout the process [101].

2.2.2 Related Works

In response to the aforementioned challenges, numerous studies have proposed solutions aimed at improving training efficiency. This section provides a brief overview of these approaches, categorized by the hardware platforms used—specifically, implementations targeting ASIC and FPGA architectures.

ASIC

Many DNN applications, particularly those deployed on edge devices, require substantial computational power while meeting strict latency constraints. To enable real-time deployment of DNNs, embedded processors must deliver high throughput with low power consumption. Consequently, the demand for domain-specific accelerators has grown significantly. ASIC architectures, designed for customized applications, offer optimal speed and energy efficiency [108].

Numerous ASIC-based solutions have been proposed to address the challenges outlined above. These approaches can be broadly categorized into two strategies for mitigating the transposition problem in computation. The first approach is data rearrangement for the GEMM kernel input. This method involves reordering matrix indices to produce a transposed matrix as input to the GEMM kernel, as illustrated in Figure 2.4(a). The latency introduced by data rearrangement can be overlapped with computation latency between matrix tiles. References [102, 104] implement this by reading tiled weights from external memory and rearranging the transposed tiles in on-chip static random-access memory (SRAM). Other

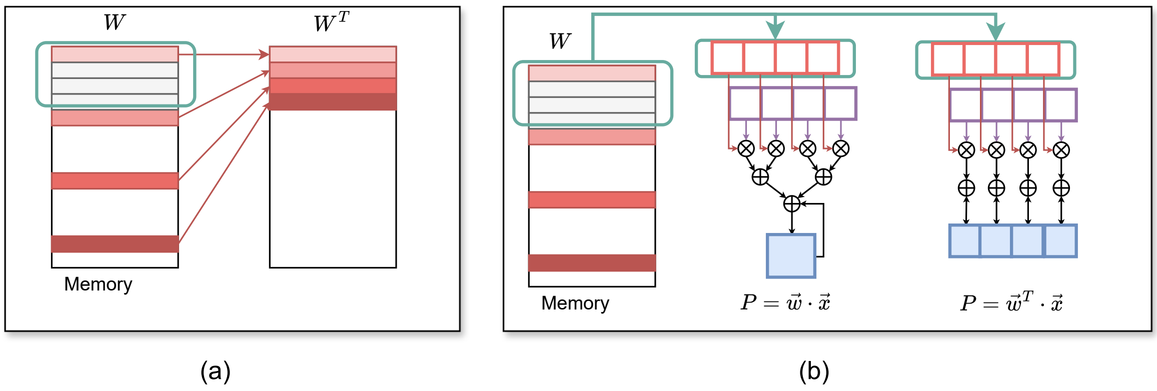


FIGURE 2.4: Techniques for addressing the transposition problem. (a) Data rearrangement. (b) Data path configuration. The uncolored/colored bars in the memory block represent elements of W, with word lengths depending on specific applications (e.g., 8-bit or 16-bit).

works [150, 155] propose customized SRAM designs capable of directly reading transposed matrices, albeit at the cost of reduced SRAM cell density. The second approach is the configurable data path inside GEMM kernel. This strategy avoids explicit transposition by designing a configurable data-path GEMM kernel, as shown in Figure 2.4(b). By adjusting the data flow between processing element (PE)s, the PE array can be reconfigured to perform transposed matrix multiplication. However, the performance of the GEMM kernel varies depending on the data path configuration. Reference [152] introduces a PE array that can be configured as weight-stationary, output-stationary, or row-stationary. Reference [102] achieves transposition by switching the weight and output dataflow configurations, while [88] enables transposed/non-transposed computation by altering the input data paths for weights and activations.

To address data dependency challenges, alternative algorithms have been proposed to replace traditional backpropagation. One such method is direct feedback alignment, illustrated in Figure 2.5. In standard backpropagation, each layer's error is computed based on the error from the subsequent layer. In contrast, direct feedback alignment propagates the final layer error $\mathbf{e}^{[L]}$ directly to all preceding layers, thereby breaking inter-layer dependencies and increasing parallelism in the backward pass. References [52, 56, 102] incorporate direct feedback alignment into ASIC designs to enhance training efficiency. However, these methods are typically effective only for specific problem domains.

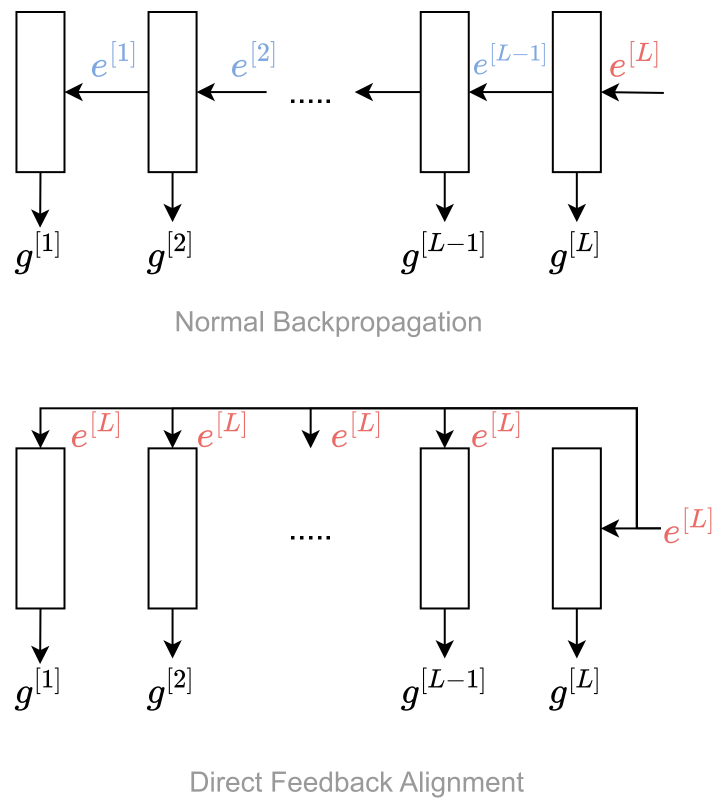


FIGURE 2.5: Illustration of direct feedback alignment.

To mitigate data storage and movement overheads, sparsity introduced by activation functions (e.g., ReLu) or max pooling layers is exploited during backpropagation to compress zero-valued elements in tensors. Techniques such as zero-value compression [86, 91], compressed sparse row encoding [79], and run-length encoding [102] have been proposed. Additional methods include JPEG-based compression and hierarchical compression schemes [38, 85, 132], which build upon run-length encoding. In sparsity optimization, MAC unit designs also leverage sparsity to skip zero-valued computations during backpropagation, thereby improving computational efficiency [52, 86, 91, 102].

FPGA

A FPGA is a reconfigurable integrated circuit that can be programmed post-fabrication to implement a wide range of custom circuit designs. The architecture of a FPGA consists of various programmable components, including logic elements, I/O modules, and specialized units. These components are interconnected via pre-fabricated routing tracks and programmable switches. Circuit functionality and reconfiguration are primarily achieved through logic blocks and digital signal processing (DSP) blocks. Developers typically design circuits using hardware description language (HDL) languages such as Verilog or VHDL, or convert high-level C code into HDL. Once the HDL design is complete, a computer aided design (CAD) tool compiles it into a *bitstream* file used to configure the FPGA [12].

The programmability, flexibility, and customizable I/O interfaces make FPGA a compelling platform for deep learning acceleration tasks [11]. First, its fine-grained programmability allows for the implementation of computing units with low-precision or custom data formats, such as INT8, 8-bit floating-point (FP8), binary, or ternary arithmetic [29, 118]. Second, the customization capabilities enable accelerator designs optimized for specific constraints, such as low-latency and low-batch inference scenarios. Third, the reconfigurability of FPGA allows developers to adapt to the rapid evolution of neural network architectures by modifying the HDL design and reloading the bitstream. Fourth, unlike traditional processors, FPGA lacks hierarchical memory and instruction sets, enabling direct data access from I/O interfaces. This feature allows neural network accelerators on FPGA to communicate directly with external devices supplying input data.

Several FPGA-based inference accelerator designs have demonstrated the platform's performance capabilities. For example, reference [176] achieved a 741 MHz operating frequency and over 7 TOPS for CNN inference on a Xilinx Ultrascale device, while reference [35] reported 17.2 TOPS on a Xilinx Alveo U280 at 300 MHz. Furthermore, FPGA can integrate multiple IP cores to support full-system implementations. These high-performance results also suggest the feasibility of on-chip training using FPGA.

Recent research has explored FPGA implementations for training. Early works investigated architectural and parallelism strategies for CNN training [105, 198]. References [123, 193] focused on implementing BFP-based training for CNN, while [41] demonstrated 8-bit BFP training using the stochastic weight averaging low-precision (SWALP) algorithm, achieving accuracy comparable to 32-bit floating-point (FP32). Reference [167] showed that low-batch training significantly reduces memory requirements and improves energy efficiency on FPGA compared to GPU. Reference [172] proposed a hybrid model-layer parallelism strategy for CNN training, achieving 6.4× higher energy efficiency than comparable GPU servers. These implementations highlight the energy efficiency advantages of FPGA, making it well-suited for power-constrained, edge-based training applications.

2.3 Low-precision arithmetic used for training

Low-precision arithmetic is one of the key optimization techniques for neural network training. Its primary advantage lies in reducing memory usage for both on-chip and off-chip storage, thereby lowering memory bandwidth requirements. A variety of low-precision arithmetic schemes have emerged, including Minifloat, posit [51], block arithmetic, mixed fixed-point, and logarithmic number systems [196, 197]. Among these, Minifloat and block arithmetic are the two most widely adopted approaches in training.

2.3.1 Fixed Point

Fixed-point arithmetic is a method of representing real numbers in digital systems. It encodes values using a fixed number of binary digits, with a predetermined position for the binary point. An example of an unsigned fixed-point number is shown in Figure 2.6. A fixed-point number with M total bits and F fractional bits can be represented as a bit vector:

$$U = (u_{M-1}u_{M-2}\ldots u_0), \quad u_i \in \{0,1\} \tag{2.25}$$

Its numerical value is given by:

$$U = (u_{M-1}u_{M-2}\ldots u_0)_2 = \sum_{i=0}^{M-1} u_i \times 2^i \times 2^{-F} \tag{2.26}$$

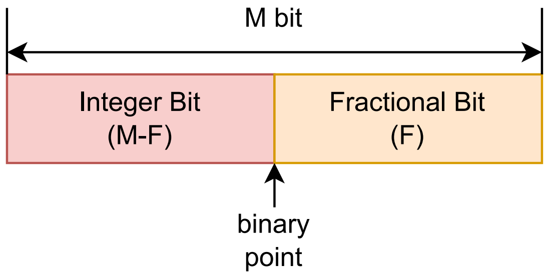


FIGURE 2.6: Unsigned fixed-point arithmetic scheme.

Fixed-point numbers can also represent signed values using two's complement notation. In this format, the most significant bit (most significant bit (MSB)) carries a negative weight, resulting in the following representation for a signed fixed-point number S:

Further, the fixed point can also represent a signed real number in 2's complement form. The representation of the 2's complement form is similar to the unsigned fixed-point, but with the MSB having weight of $-u_{M-1}2^{M-1}$. Thus, the 2's complement form of a signed fixed-point number S can be represented as:

$$S = (s_{M-1}s_{M-2}s_{M-3}...s_0)_2 = (-s_{M-1}2^{M-1} + \Sigma_{i=0}^{M-2} s_i \times 2^i) \times 2^{-F}, s_i \in \{0, 1\} \quad (2.27)$$

The precision and dynamic range of a fixed-point number depend on the configuration of integer and fractional bits. For a signed fixed-point number with m total bits and f fractional bits, the smallest representable difference between two values is 2^{-f}, and the roundoff error introduced during quantization is approximately 2^{-f-1}.

The dynamic range of such a number is given by:

$$-2^{m-f-1}, 2^{m-f-1} - 2^{-f} \quad (2.28)$$

If the magnitude of a real number exceeds this range, overflow or underflow occurs. A common mitigation strategy is saturation, where values are clipped to the maximum or minimum representable limits, introducing clipping errors.

Due to its limited dynamic range, fixed-point arithmetic has traditionally seen limited use in training [21]. However, some works have explored fixed-point-based training by incorporating

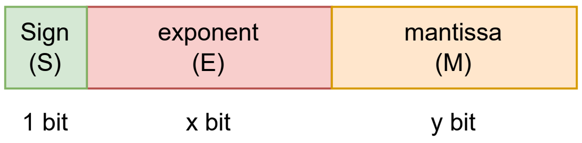


FIGURE 2.7: FP arithmetic structure

scaling factors or shared exponents to extend the dynamic range [18, 54, 55, 92, 164, 187]. These approaches are conceptually closer to block arithmetic than to conventional fixed-point representations. However, fixed-point arithmetic can be highly efficient for inference when specific techniques are applied to address quantization challenges. SmoothQuant [180] introduces a post-training quantization method for LLM, enabling INT8 quantization of both weights and activations across all matrix multiplications in LLMs. Building on SmoothQuant, FlightLLM [189] achieves an average of 3.5 bits for weights and 8 bits for activations.

2.3.2 Floating Point

FP arithmetic is widely used in neural network training due to its significantly larger dynamic range compared to fixed-point arithmetic. The IEEE 754 standard [73] defines the conventional formats for FP representation. A floating-point number can be expressed in the format FP$\langle x, y\rangle$, where the number consists of a 1-bit sign S, an x-bit exponent E, and a y-bit mantissa M, as illustrated in Figure 2.7.

The IEEE 754 standard specifies three commonly used FP formats: 64-bit floating-point (FP64), FP32, and FP16. Specifically, FP64 uses a 64-bit FP$\langle 11, 52\rangle$ format, FP32 uses a 32-bit FP$\langle 8, 23\rangle$ format, and FP16 uses a 16-bit FP$\langle 5, 10\rangle$ format. These formats are widely supported by modern GPU architectures, such as NVIDIA A100 [126] and H100 [127], and have been adopted in early training implementations [89, 115], which primarily utilize FP32 and FP16.

TABLE 2.2: The summary of floating point data format

	Format	Bit Length	Data Format
IEEE 754 Standard	FP64	64	FP$\langle 11, 52\rangle$
	FP32	32	FP$\langle 8, 23\rangle$
	FP16	16	FP$\langle 5, 10\rangle$
Minifloat	BFloat16	16	FP$\langle 8, 7\rangle$
	FP8	8	FP$\langle 5, 2\rangle$
			FP$\langle 4, 3\rangle$
	FP6	6	FP$\langle 2, 3\rangle$
			FP$\langle 3, 2\rangle$
	FP4	4	FP$\langle 2, 1\rangle$

The value of IEEE 754 standard FP number $f(s, e, m)$ is given by Equation 2.29:

$$f(s,e,m) = \begin{cases} (-1)^s \times (0 + m \times 2^{-y}) \times 2^{1-\eta} & e = 0 \text{ (denormalized)} \\ (-1)^s \times (1 + m \times 2^{-y}) \times 2^{e-\eta} & e \neq 0 \text{ (normal)} \\ \text{NaN or Infinity} & e = 2^x - 1 \end{cases} \tag{2.29}$$

η denotes the exponent bias, defined as $\eta = 2^{(x-1)} - 1$, and s, e, and m represent the sign, exponent, and mantissa values of the floating-point number $f(s, e, m)$, respectively.

As shown in Equation 2.29, normalized numbers include an implicit leading bit to reduce memory usage, represented by the term $(1 + m \times 2^{-y})$. Denormalized numbers, also referred to as subnormal numbers, are used to represent values in the gradual underflow region. In this case, the implicit bit is set to zero, resulting in the term $(0 + m \times 2^{-y})$. Special values such as NaN (Not a Number) are used to represent undefined numerical results, such as $\frac{0}{0}$, while infinity denotes values that exceed the representable range of the floating-point format.

In addition to conventional floating-point formats, minifloat is a reduced-bit floating-point representation [117]. It adheres to the principles of the IEEE 754 standard [73], including the definitions of infinity, not a number (NaN), normalized, and subnormal numbers, although minor variations may exist across different implementations. A summary of floating-point data formats is provided in Table 2.2.

brain floating-point 16 bits (BFLOAT16) is one of the earliest minifloat formats adopted for neural network training [74, 84]. It uses a 16-bit FP$\langle 8, 7\rangle$ format, offering the same dynamic range as FP32 due to its identical exponent width, while reducing the overall word length by half. BFLOAT16 was first introduced in the distributed learning framework TensorFlow [3] and supported by the hardware chip tensor processing unit (TPU) [173]. Compared to 16-bit signed integer (INT16) or FP16, BFLOAT16 does not require additional hyperparameter tuning, such as loss scaling for FP16 or fine-grained block quantization for INT16. It achieves accuracy comparable to FP32 across various applications, including computer vision, speech recognition, natural language processing, generative models, and recommendation systems [84]. Today, BFLOAT16 is supported by a wide range of mainstream hardware platforms [13], including Intel Cooper Lake [2], Ice Lake [72], IBM Power 10 [71], Arm Neoverse [8], and Habana HL [159].

FP8 further reduces the bit width of minifloat formats to 8 bits and is increasingly becoming a mainstream format in modern hardware, such as Graphcore [81] and NVIDIA H100 [127]. While FP8 generally follows the IEEE 754 standard, two commonly used formats are FP$\langle 5, 2\rangle$ and FP$\langle 4, 3\rangle$, with some variations in implementation details [10]. For example, reference [162] defines a configurable FP8 format with a 6-bit unsigned exponent bias, allowing the dynamic range to vary based on the bias value. In this format, NaN and infinity are not supported. In contrast, [114] adopts IEEE 754-compliant FP$\langle 5, 2\rangle$, while FP$\langle 4, 3\rangle$ extends its dynamic range by omitting infinity and using a single mantissa bit-pattern for NaNs. Reference [124] investigates the impact of different exponent biases and finds that selecting a bias that covers the majority of value distributions enables FP8-based matrix multiplications and convolutions to match the test performance of FP32.

In inference, INT8 is a widely used arithmetic format [77], often paired with quantization aware training (QAT) [121] to mitigate accuracy degradation. However, FP8 simplifies deployment by avoiding the need for such compensatory techniques and maintaining higher accuracy [97].

The use of FP8 in training has also gained traction. Early studies demonstrated the feasibility of training DNNs with FP8 and reported minimal accuracy loss across various models and

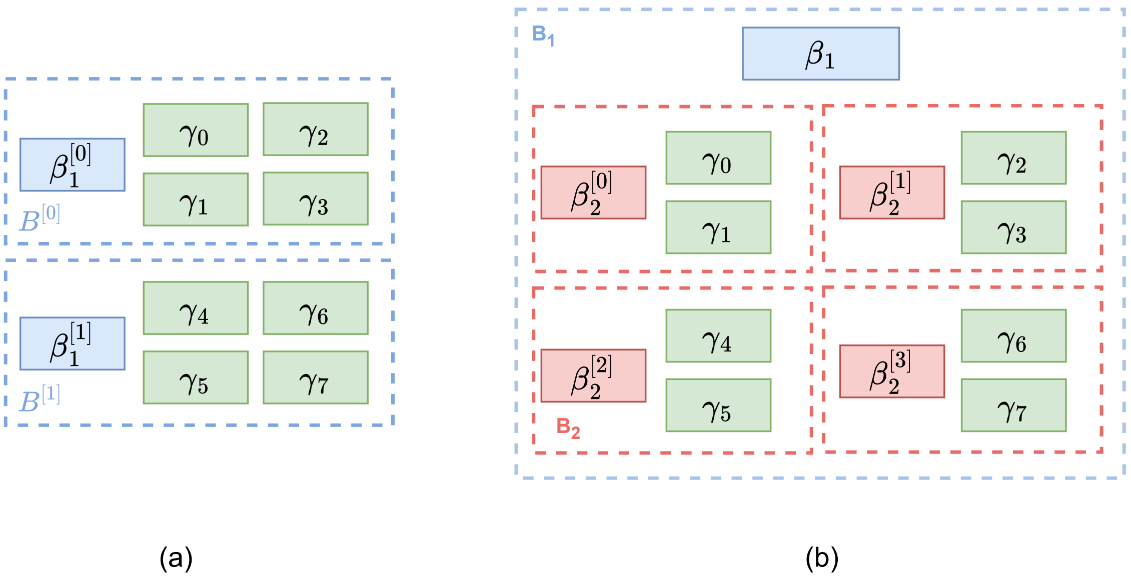

$\beta_1^{[0]}$
$B^{[0]}$
γ_0 γ_1 γ_2 γ_3
$\beta_1^{[1]}$
$B^{[1]}$
γ_4 γ_5 γ_6 γ_7
(a)
B_1
β_1
$\beta_2^{[0]}$ $\beta_2^{[1]}$ $\beta_2^{[2]}$ $\beta_2^{[3]}$
B_2
(b)

Figure 2.8: An illustrative example of block arithmetic. (a) Block arithmetic with one-level shared exponent. The block size in this example is 4. (b) Block arithmetic with two-level shared exponents β_1 and β_2. In this example, the first-level block B_1 has a block size of 4, and the second-level block B_2 has a block size of 2.

datasets [170]. Reference [157] proposed a hybrid FP8 format for end-to-end distributed training. More recently, several works have explored the application of FP8 in LLM training [39, 43, 133, 135] and inference [90, 103], aiming to reduce memory storage requirements. Furthermore, the newly released NVIDIA Blackwell architecture [1] introduces support for 4-bit floating-point (FP4) and 6-bit floating-point (FP6) formats, prompting emerging research on FP4-based LLM training [171].

2.3.3 Block Arithmetic

Block arithmetic methods are widely adopted in low-precision neural network training, including BFP [36], MX [129, 144], BM [40], and Microexponent [30]. These formats can represent tensor elements using 8 bits or fewer [144], making them suitable for training tasks such as LLM, while maintaining accuracy comparable to FP32. Block arithmetic is also applicable to low-precision inference, as seen in methods like vector scaling quantization (VSQ) [26].

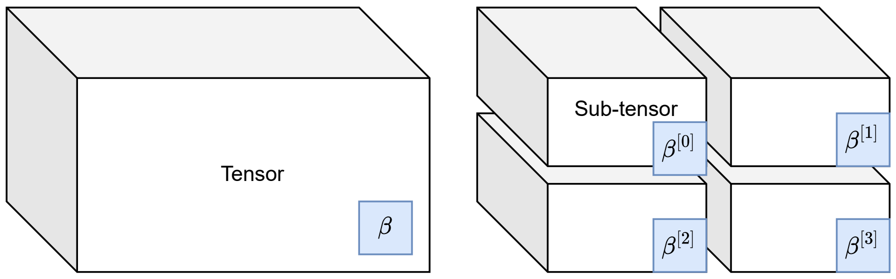


FIGURE 2.9: An example of a global block and a local block. The tensor in the example is a three-dimensional tensor. (a) Global block example. (b) Local block example.

Block arithmetic consists of data elements γ_i and shared exponents β. The elements can be represented in either floating-point or fixed-point format. The shared exponent β is a value common to all elements within a block B, allowing the representation to save exponent bits while preserving a wide dynamic range. The block size refers to the number of elements sharing the same exponent.

In deep learning, a *tensor* is a fundamental data structure representing multidimensional arrays, generalizing scalars, vectors, and matrices to higher dimensions [161]. When applying block arithmetic to tensors, two types of shared exponent schemes are commonly used: *global shared exponent* and *local shared exponent* [87]. Figure 2.9 illustrates examples of both. In the global block scheme, a single shared exponent is applied to the entire tensor, making the block size equal to the tensor size. In the local block scheme, the tensor is partitioned into smaller blocks, each with its own shared exponent. The shape of local blocks varies across network architectures. For example, ResNet assigns a shared exponent per channel [4, 146], while Transformer models divide tensors into sub-matrices, each with its own shared exponent [137].

Block arithmetic can be implemented with either one or two levels of shared exponent. Figure 2.8 illustrates both cases. One-level shared exponent formats, such as BFP, BM, and MX, express values as:

$$f(\gamma_i) = g(\gamma_i) \times 2^{\beta^{[j]}}, \quad \gamma_i \in B^{[j]} \tag{2.30}$$

γ_i is an element in the j-th block $B^{[j]}$, $g(\gamma_i)$ is its value (following the definitions in Equation 2.29 and Equation 2.27), and $\beta^{[j]}$ is the shared exponent for that block.

Two-level shared exponent formats, such as Microexponent and VSQ, use nested exponent blocks. As shown in Figure 2.8(b), the value is expressed as:

$$f(\gamma_i) = g(\gamma_i) \times 2^{\beta_2^{[j]}} \times 2^{\beta_1^{[k]}}, \quad \gamma_i \in (B_1^{[j]} \cap B_2^{[k]}) \tag{2.31}$$

In this case, γ_i belongs to the j-th first-level block $B_1^{[j]}$ and the k-th second-level block $B_2^{[k]}$. For Microexponent, the second-level block size B_2 is 2; for VSQ, it is 16.

BM Format

A BM number consists of a small floating-point value (referred to as a minifloat) together with a shared exponent bias, β, which is common to all BM numbers within the same block. The BM format supports both normal and denormalized (denorm) numbers, but employs saturating arithmetic instead of IEEE-754 overflow and underflow behavior, including Inf and NaN. The rounding scheme used is round-to-nearest, while stochastic rounding is applied during weight updates to improve convergence, particularly in low-precision training.

A minifloat representation is parameterized by the number of exponent and mantissa bits. An $\langle e, m\rangle$ minifloat format includes one sign bit, e exponent bits, and m mantissa bits. Let s, E, and M denote the unsigned integer representations of the sign, exponent, and mantissa fields, respectively. The real number represented by the minifloat is computed using:

$$f(s,E,M) = \begin{cases} (-1)^s \times g(s,M) \times 2^{1-\eta} & E = 0 \text{ (denorm)} \\ (-1)^s \times g(s,M) \times 2^{E-\eta} & E \neq 0 \text{ (normal)} \\ 0 & E = M = 0 \end{cases} \tag{2.32}$$

where $\eta = 2^{e-1} - 1$ is the exponent bias for the binary-offset encoded exponent. The significand S is defined as:

$$S = g(s, M) = \begin{cases} M \times 2^{-m} & E = 0 \text{ (denorm)} \\ 1 + M \times 2^{-m} & E \neq 0 \text{ (normal)} \end{cases} \tag{2.33}$$

We also define an inverse function that extracts the sign, exponent, and mantissa components from a BM-representable value x:

$$(s, E, M) = f^{-1}(x) \tag{2.34}$$

Similar to BFP [37], the BM format [40], denoted as BM$\langle e, m\rangle$, is used to describe a submatrix (or block) P, where each element $r_i \in P$ shares a common exponent bias β:

$$r_i = f(s_i, E_i, M_i) \times 2^{\beta} \tag{2.35}$$

For example, the BM$\langle 2, 5\rangle$ format (sometimes written as BM8$\langle 2, 5\rangle$ to indicate the total word length) includes a shared bias, 2 exponent bits, and 5 mantissa bits.

The set of elements sharing the same exponent bias is referred to as a block, and its cardinality defines the block size. Smaller block sizes are beneficial for improving training accuracy [36, 59]. The term tile size refers to the dimensions of the PE array used to construct the GEMM kernel, denoted as $T \times T$, and is typically chosen as large as possible for performance. The block size, denoted as $B_k \times B_k$, is generally distinct from the tile size and is selected to control accuracy loss.

BM can be viewed as a superset of several low-precision formats, including BFP, minifloat, and logarithmic number systems. Specifically:

- When $e = 0$, BM reduces to BFP; BM$\langle 0, m\rangle$ represents a $(m + 1)$-bit BFP value with one sign bit and m integer bits.
- When $\beta = 0$, BM behaves as a minifloat.
- When $m = 0$ and $\beta = 0$, BM corresponds to a logarithmic number system.

- When $e = 0$ and $\beta = 0$, BM is equivalent to fixed-point arithmetic.

MX Format

Block arithmetic has proven to be highly efficient for deep learning applications and was widely adopted by major hardware manufacturers in 2023—including Microsoft, AMD, Intel, Meta, NVIDIA, and Qualcomm—through the introduction of the MX format [129, 144].

MX format has the same definition as BM, but MX provides a more detailed definition for corner cases, such as NaN and Infinity. According to the specification outlined in [129], an MX-compliant format is defined by three key components: a global scale factor X, a set of private elements $P = [P_u]_{u=0}^{B-1}$, and a block size B.

The value of a block arithmetic number is defined as:

$$c_u = g(P_u) \times 2^X \tag{2.36}$$

where $g(P_u)$ denotes the value of the private element P_u, interpreted according to either fixed-point or floating-point semantics. The standard further specifies that a mechanism must be provided for converting a vector $V = [V_u]_{u=0}^{B-1}$ of scalar elements into an MX-compliant format $\{X, [P_u]_{u=0}^{B-1}\}$, by producing the block scale X and the private element vector P.

Several specific MX format definitions are presented in Table 2.3 [144]. All formats listed in Table 2.3 use a block size of 32 and an 8-bit block scale X, which is encoded using the FP$\langle 8, 0\rangle$ format. The MX elements may be represented using either minifloat or fixed-point formats. The value of an MX number r_i is given by:

$$r_i = f(s_i, e_i, m_i) \times 2^X \tag{2.37}$$

Here $f(s_i, e_i, m_i)$ is the scalar element P_i value.

Recently, publications have appeared about the MXFP4 training for LLM with comparable accuracy to that of FP32 on Vision Transformer pre-training [19] and GPT-6.7B pre-training.

TABLE 2.3: MX scalar data format

Format	Element Data Format	Element Bit-width
MXFP8	FP8(FP$\langle 4,3\rangle$/FP$\langle 5,2\rangle$)	8
MXFP6	FP6(FP$\langle 2,3\rangle$/FP$\langle 3,2\rangle$)	6
MXFP4	FP4(FP$\langle 2,1\rangle$)	4
MXINT8	INT8	8

Pre-training is commonly used in LLM training, initializing the LLM model by training it with a large and generic dataset to learn broad knowledge or features. Reference [144] evaluates MX with global blocks on various tasks, including inference, training, and fine-tuning. For training tasks, they showcase training with MXFP4 weights and MXFP6 activations and gradients on GPT models, from GPT-20M to GPT-1.5B.

Although quantization for linear layers has been widely used, its application to accelerate the attention mechanism remains limited [192]. SageAttention [191] utilizes INT8 with local blocks for Q and K matrices, FP16 for P and V matrices, and precision-enhancing methods, implementing an accurate and 2x speedup kernel compared to FlashAttention2. SageAttention2 [192] utilizes 4-bit integer (INT4) Q and K matrices and FP8 P and V matrices with local blocks, alongside additional precision-enhancing techniques, surpassing FlashAttention2 [28] and xformers [184] in operations per second (OPS) by approximately 3x and 5x on RTX4090 [163].

Rescaling

Rescaling is required not only for the initial conversion of input values but also for the transformation of intermediate block-format values. In block arithmetic, rescaling is applied after the inner-product computation for GEMM and convolution operations. This process converts the high-precision partial sums of the elements within a block into a low-precision block-arithmetic format and adjusts the shared exponent to prevent overflow during element conversion.

We denote this rescaling process using the functions $\mathcal{R}_{\mathcal{X}}$ and $\mathcal{R}_{\mathcal{P}}$:

$$X = \mathcal{R}_{\mathcal{X}}(Z), \tag{2.38}$$

$$P = \mathcal{R}_{\mathcal{P}}(Z). \tag{2.39}$$

Here, the intermediate value Z is a high-precision accumulator, typically represented in formats such as FP32, FP64, or long word-length integers to prevent overflow during inner-product computation. The variable P denotes the low-precision elements of the block after conversion, while X corresponds to the shared exponent of that block.

While other techniques can be utilized, the following minimal support is required. The scale is computed using the function:

$$X = \mathcal{R}_X(Z) = \lfloor \log_2(\max_{z_u \in Z}(|z_u|)) \rfloor \tag{2.40}$$

The element vector is then scaled according to

$$P = \mathcal{R}_P(Z) = [V_u/2^X]_{u=0}^{b-1} \tag{2.41}$$

and subsequently quantized to the element data type. Rescaling based on the maximum value is commonly referred to as maximum calibration [179]. Although Equation 2.40 provides an optimal scaling of Z to fit within the element data type, this approach can only be applied once all z_u inputs are known.

NVIDIA FP4

NVIDIA FP4 (NVFP4) is the FP4 format supported in the latest NVIDIA Blackwell GPU architecture [75]. Figure 2.10 illustrates the structure of the NVFP4. Unlike MX4 or BM4 with one-level shared exponent, NVFP4 has a two-level shared exponent, with a FP32 second-level scaling factor for the global block and FP$\langle 4, 3\rangle$ first-level scaling factors for the 16-element local blocks (micro-block).

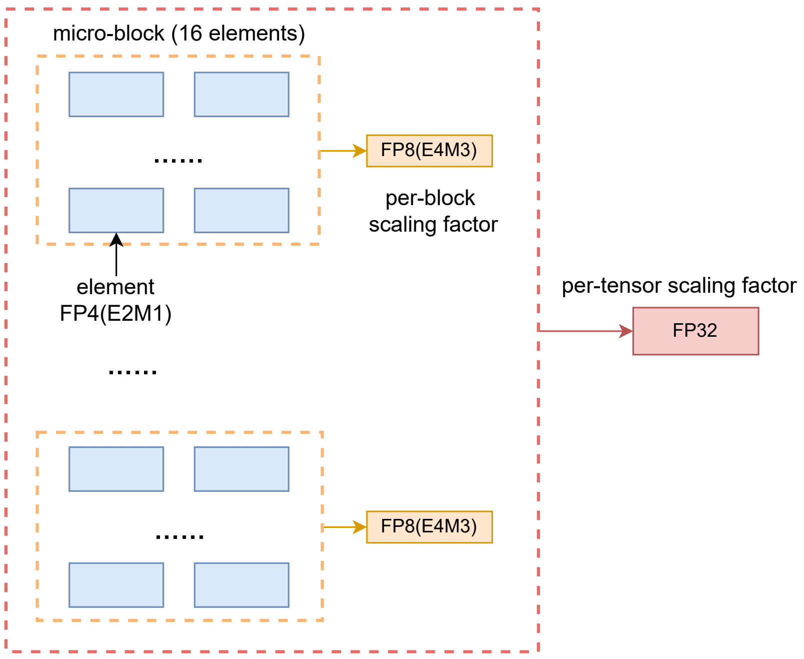


FIGURE 2.10: The structure of NVFP4 arithmetic.

NVFP4-based pretraining results achieve training loss and downstream task accuracies comparable to an FP8 or BFLOAT16 baseline [5, 20]. Reference [20] demonstrates the first fully FP4 training of LLMs on large-scale datasets, systematically analyzing key FP4 design choices such as block size, scaling format, and rounding strategy, and identifying NVFP4 (16 values sharing an FP$\langle 4, 3\rangle$ scale) as the most effective. Reference [5] compared the training efficiency between NVFP4 and MXFP4, finding that the loss of MXFP4 matches that of NVFP4 when trained on 36% more tokens, which means MXFP4 needs more training time than NVFP4.

Regarding hardware aspects, one of NVIDIA's research papers illustrates the rescaling algorithm of two-level scaling quantization [26]. For the training application, NVFP4 requires the rescaling computation to be performed twice: first for the second-level global block scalar and second for the micro-block scaling factor.

TABLE 2.4: Comparison of Arithmetic Schemes

	Advantages	Weaknesses
Fixed-point	Simple representation of real numbers; compact and efficient computation units	Limited dynamic range
Floating-point	Large dynamic range	Higher computational complexity and larger hardware units compared to fixed-point
Block arithmetic	Shared exponent provides a larger dynamic range and reduces per-element exponent storage	Rescaling introduces additional latency

2.4 Summary

This chapter provides a brief background on machine learning, an overview of relevant literature on training, and a discussion of low-precision arithmetic for training.

The machine learning section introduces the fundamental concepts and categories of machine learning. A time-series prediction example is used to illustrate the supervised learning process. The DNN training procedure is explained through the forward and backward propagation of a MLP.

We also examine the challenges associated with accelerating training by analyzing the characteristics of DNN training. Related research on ASIC- and FPGA-based training is reviewed, highlighting the advantages of FPGA, including programmability, flexibility, and configurable I/O interfaces.

Finally, we present fixed-point, floating-point, and block arithmetic schemes, including their definitions, variants, and related work. Each arithmetic scheme offers distinct advantages and limitations, which are summarized in Table 2.4.

CHAPTER 3

Block Minifloat Arithmetic for Inference

3.1 Introduction

Machine learning inference using low-precision arithmetic has been a heavily researched topic as inference applications abound, and reducing precision requirements without significantly sacrificing accuracy can enable new applications where conventional inference implementations are too computationally expensive. FPGA is an excellent platform for this type of application, particularly for edge devices where size, weight and power (SWaP) is a design consideration [176, 177].

Although many DNN training and inference accelerators use 16 or 32 bit floating point encoding (such as IEEE-754 single-precision, half-precision [73], or Google BFLOAT16 [74, 84]), lower precision weights and activations offer significant implementation benefits including reduced memory bandwidth, memory requirements, and implementation area. One popular scheme for DNN accelerator design is fixed-point arithmetic, which requires significantly fewer hardware resources than floating-point [58, 139]. However, one problem for fixed-point implementations is a small dynamic range, leading to reduced accuracy. Quantization-aware training techniques have been proposed to address this problem [17, 49, 77], and commercial quantization-aware training products are available from vendors such as advanced micro devices (AMD) and Intel (e.g. [22]).

There has been recent interest in low-precision floating-point data formats for the training of DNNs. These utilise 8, or even 6-bit, floating-point representations and can train neural networks from scratch [40, 170]. Minifloat uses a standard floating-point representation of exponent and mantissa with small word lengths, whereas BM has an additional shared block exponent that covers all the elements in a tensor to expand its dynamic range. Little research has been conducted on inference using these schemes, but compared to a conventional floating-point fused multiply accumulate (FMA), BM is computationally less expensive, particularly for floating-point formats with small exponent lengths, as large shifters for alignment are avoided. For example, 32-bit integer adders are approximately $10\times$ smaller and $4\times$ more energy efficient than FP16 units [27].

In this chapter, we explore the applicability of BM to time series prediction using the N-BEATS algorithm [130] and present a BM-based inference accelerator design. N-BEATS is a DNN for time series prediction which, at the time of publication, achieved 3$\%$ improvement over the winner of the M4 forecasting competition. The main contributions of this chapter are:

- We present a BM arithmetic implementation for the global block and the BM based systolic array for inference.
- We present a novel architecture for the acceleration of N-BEATS based on BM arithmetic. During the publication, this was the first implementation of an FPGA-based accelerator using BM arithmetic.
- The accuracy and area trade-offs between BM, floating-point, and fixed-point are explored. It is found that 8-bit BM achieves area and performance similar to an INT8 datapath with accuracy on NBEATS similar to FP16.

3.2 BM arithmetic implementation

We first describe the technique for performing BM matrix multiplications. The strategy for its acceleration is to use an efficient GEMM core which computes the inner product between each row and column of the input matrices, A and B, to produce the output matrix C. The block of matrices A and B is the global block with shared exponents β_A and β_B. Elements (s_a, M_a, E_a) and (s_b, M_b, E_b) in matrices A and B are represented in $BM\langle e_a, m_a\rangle$

and $BM\langle e_b, m_b\rangle$ format. For the BM data, each element of C is the inner product:

$$c_{ij} = \sum_k f(s_{a_{ik}}, M_{a_{ik}}, E_{a_{ik}}) f(s_{b_{kj}}, M_{b_{kj}}, E_{b_{kj}}) 2^{\beta_A+\beta_B} \tag{3.1}$$

The computation of minifloat GEMM is done in two steps: computation of the inner product and rescaling. Rescaling includes normalization of the partial sum to BM format, and alignment of the shared exponent bias. The shared exponent bias of the new matrix is:

$$\beta_{share} = \beta_A + \beta_B \tag{3.2}$$

3.2.1 Inner Product

For the inner product computation, operands a and b are multiplied as:

$$\begin{aligned} S_{mul} &= S_a \times S_b; \\ E_{mul} &= E_a + E_b \end{aligned} \tag{3.3}$$

S_{mul} and E_{mul} are the significand and exponent values of the minifloat multiplication result. These numbers are combined and accumulated with the partial sum to form the signed integer inner product P_{sum}:

$$P_{sum} = P_{sum} + S_{mul} << E_{mul}. \tag{3.4}$$

The inner product computation just described is implemented using an integer Kulisch accumulator with sufficient precision for error-free accumulation [40, 96]. Its size K_{add} and shifter range K_{shift} are given in Equation 3.5. An extra W_I bits are assigned to avoid overflow over multiple accumulations. For the BM data format, the minifloat exponent value is small, limiting the adder's required size.

$$\begin{aligned} K_{\text{shift}} &= 2^{e_a+e_b} \\ K_{\text{add}} &= 1 + 2^{e_a+e_b} + (1 + m_a + 1 + m_b) + W_I \end{aligned} \tag{3.5}$$

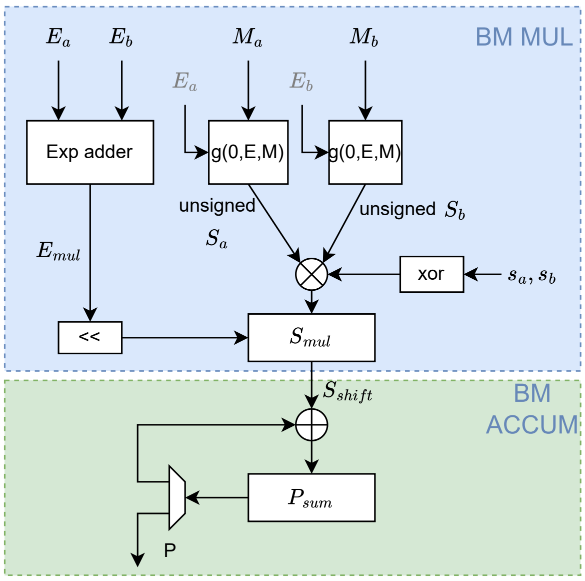


FIGURE 3.1: The BM MAC unit design for the global block.

The MAC unit design is illustrated in Figure 3.1. It can be divided into two parts: BM MUL for S_{mul} calculation and shifting; and BM ACCUM for partial sum P_{sum} accumulation. The interface between BM MUL and BM ACCUM is S_{shift}, the shifted significand value. A bit difference in the MAC unit is the significand computation in Equation 3.3. The MAC unit calculates the unsigned significands and the sign bit separately; S_{mul} is calculated with the multiplication of the sign bit and the unsigned significands S_a and S_b. After the inner-product computation finishes, the output of the MAC unit is the partial sum result P.

3.2.2 Rescaling

After inner-product finishes, the GEMM produces a block of N values, $P[N]$, which are rescaled to minifloats using the BM normalization function (algorithm 2). This involves transforming the accumulator outputs to a normalized $C\langle e_{norm}, m\rangle$ number with ***KulToBM*** and then converting into $A\langle e, m\rangle$ format with ***ExpAlign***:

$$A, \beta'_{share} = ExpAlign(KulToBM(P), \beta_{share}) \tag{3.6}$$

Algorithm 2: Rescaling from Kulisch accumulator integer to BM$\langle e, m\rangle$ output

```
Function KulToBM($P[N]$)
    // Normalization (integer to $C\langle e_{norm}, m\rangle$)
    for $i \leftarrow 0$ to $N-1$ do
        $s \leftarrow (P[i] > 0)$ ? 0 : 1; // Sign bit
        $Z \leftarrow CLZ(|P[i]|)$; // Leading zeros
        if $Z \geq Cnt_{denorm}$ and $Z < Cnt_{underflow}$ then
            $C[i] \leftarrow f(s, 0, |P[i]| << (Cnt_{\text{denorm}} - 1))$; // Denorm
        else if $Z \geq Cnt_{underflow}$ then
            $C[i] \leftarrow f(s, 0, 1)$; // Underflow
        else
            $E \leftarrow (2^{e+1} - 1 - \eta^{<e+1>}) + W_I - Z + \eta^{<e>}$;
            $C[i] \leftarrow f(s, E, |P[i]| << Z)$; // Normal
    $E_{max} \leftarrow maxexp(C)$; // Max exponent in block
    return $C$, $E_{max}$
Function ExpAlign($C[N]$, $E_{max}$, $\beta_{share}$)
    // Shared Exponent adjustment
    $\beta'_{share} \leftarrow \beta_{share} + max(E_{max} - (2^e - 1), 0)$;
    // Exponent Alignment ($C\langle e_{norm}, m\rangle \rightarrow A\langle e, m\rangle$)
    for $i \leftarrow 0$ to $N-1$ do
        $s, E, M \leftarrow f^{-1}(C[i])$;
        $E' \leftarrow E - max(E_{max} - (2^e - 1), 0)$; // Exp adj
        if $E' \leq 0$ and $E' > -m$ then
            $A[i] \leftarrow f(s, 0, M >> (1 - E'))$; // Denorm
        else if $E \leq -m$ then
            $A[i] \leftarrow f(s, 0, 1)$; // Underflow
        else
            $A[i] \leftarrow f(s, E', M)$; // Normal
    return $A$, $\beta'_{share}$;
Function Rescaling($P[N]$, $\beta_{share}$)
    $C, E_{max} \leftarrow KulToBM(P)$;
    $A, \beta'_{share} \leftarrow ExpAlign(C, E_{max}, \beta_{share})$;
    return $A$, $\beta'_{share}$;
```

where

$$e_{norm} \geq \lceil \log_2 K_{\text{add}} \rceil \tag{3.7}$$

In *KulToBM*, leading zeros are determined using the count leading zero function *CLZ*, and denormal, normal, and underflow cases are detected by comparisons with the boundary values

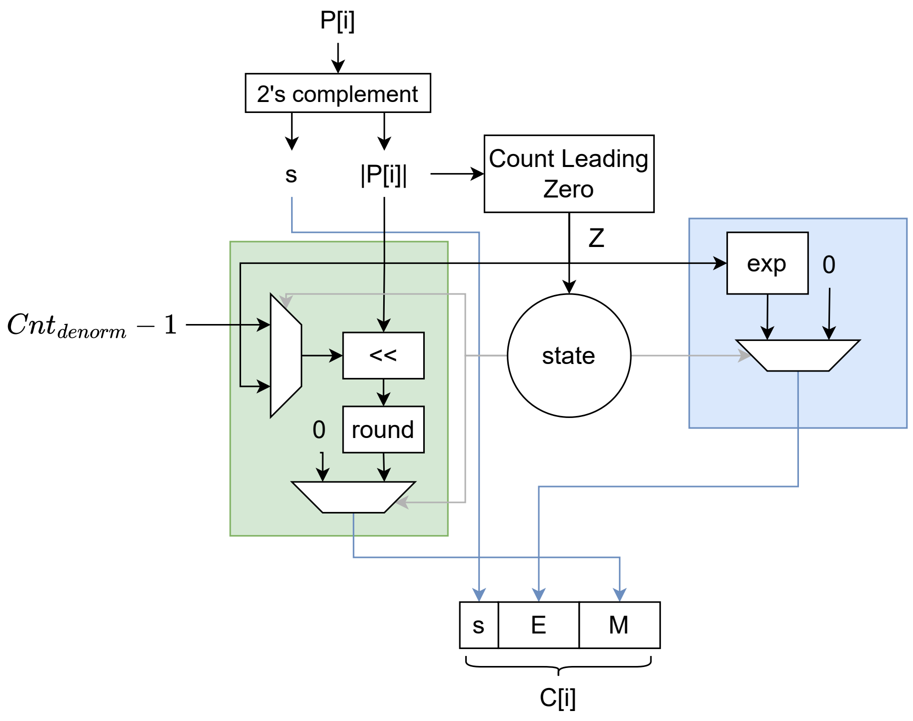


FIGURE 3.2: The normalization implementation in $KulToBM$. The left green block is responsible for mantissa computation, and the right blue block is responsible for exponent computation.

defined below:

$$\begin{aligned} Cnt_{\text{denorm}} &= W_I + (2^{e+1} - \eta^{<e+1>} + \eta^{<e>} - 1) \\ Cnt_{\text{underflow}} &= Cnt_{\text{denorm}} + m \end{aligned} \tag{3.8}$$

$\eta^{<e>} = 2^{(e-1)} - 1$ is the exponent bias for the binary-offset encoding scheme. Note that the result is in an intermediate $C\langle e_{norm}, m\rangle$ format, where e_{norm} is set via Equation 3.7. Converting $P[i]$ to an intermediate BM format $C[i]$ reduces the buffer size requirement. Exponent alignment back to the original BM format is done using *ExpAlign* in algorithm 2, where the shared exponent bias is adjusted, and the elements in C are normalized to $A\langle e, m\rangle$ format. *ExpAlign* also needs to deal with the normalised and denormalised cases to ensure $E > 0$.

The normalization in *KulToBM* implementation is illustrated in Figure 3.2 and described in algorithm 2. In normalization, $P[i]$ is first split into absolute value $|P[i]|$ and sign bit s, then the count-leading-zero block calculates the leading zeros Z, and the state block generates multiplexer control signals based on the Z value. The state generates three state signals based on the leading zero Z value, including denormal, normal, and underflow.

The left green block in Figure 3.2 is the mantissa computation block. For the denormal and underflow case, $|P[i]|$ is shifted with $Cnt_{\text{denorm}} - 1$ bits; for the normal case, $|P[i]|$ is shifted by Z bits. After shifting, if the result is normal or denormal, the relevant bits in $|P[i]|$ are split and rounded as the mantissa M; if the result is underflow, the mantissa M is set to zero. The right blue block in Figure 3.2 is the exponent computation block. For denormal and underflow cases, the exponent E is set to zero; for the normal case, the exp block is used to calculate the E value based on Z. Finally, the sign bit s, the exponent E, and the mantissa M combine to form the output $C[i]$.

The hardware implementation of *ExpAlign* follows algorithm 2, as shown in Figure 3.3. Similar to the normalization implementation, the exponent alignment computation is also reorganized for hardware efficiency. The left blue block is the exponent adjustment block. $Exp\ adj$ calculates the adjusted exponent value E'. The state generates three state signals based on E', including denormal, normal, and underflow. E' will be set to zero when it is denormal or underflow. The right green block is the mantissa adjustment block. When the state is normal or underflow, mantissa M is right-shifted with $1 - E'$ bits.

The inner product computation is error-free because the Kulisch accumulator is appropriately sized and cannot overflow. Then the integer accumulator results are converted to low-precision output by rescaling. In *KulToBM*, the minifloat values within the block are then effectively rounded to the nearest value representable, bounding this error to ϵ_1, where ϵ_1 is the machine epsilon [45]. In *ExpAlign*, a new shared exponent value is calculated based on the maximum value of C and minifloat values adjusted accordingly with a rounding error of ϵ_2. Suppose the real value of the accumulator and the rounded result are S and $\hat{S}$, and we

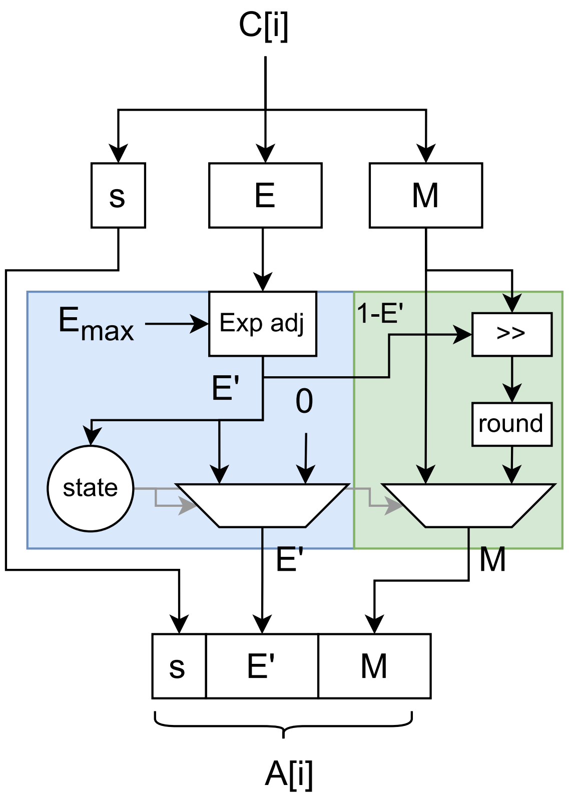


FIGURE 3.3: Exponent Alignment computation implementation in $ExpAlign$ function. The left blue block is responsible for the exponent computation, and the right green block is responsible for the mantissa computation.

define $\hat{S} = round(S) = S(1+\epsilon_1)(1+\epsilon_2)$. The error bound is:

$$\begin{aligned} \frac{|S-\hat{S}|}{|S|} &= (1+\epsilon_1)(1+\epsilon_2) - 1 \\ &= \epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2. \end{aligned} \tag{3.9}$$

3.2.3 BM Vector Addition

The BM addition is also conducted in fixed-point format and computed as follows:

$$a \pm b = \begin{cases} (S_a 2^{E_a} \pm S_b 2^{E_b} * 2^{\beta_B - \beta_A}) * 2^{\beta_A}, \beta_A \geq \beta_B \\ (S_a 2^{E_a} * 2^{\beta_A - \beta_B} \pm S_b 2^{E_b}) * 2^{\beta_B}, \beta_A < \beta_B \end{cases} \tag{3.10}$$

where S_a, S_b are the significands of a and b (Equation 2.33). The shared exponent result used is the larger of β_A and β_B:

$$\beta_{share} = \max(\beta_A, \beta_B) \tag{3.11}$$

Algorithm 3: BM Vector Addition

```
Function BMVecAdd($A[N], B[N], \beta_A, \beta_B$)
    $ebias \leftarrow |\beta_A - \beta_B|$;
    for $i \leftarrow 0$ to $N-1$ do
        $s_a, E_a, M_a \leftarrow f^{-1}(a[i])$;
        $s_b, E_b, M_b \leftarrow f^{-1}(b[i])$;
        $S_a \leftarrow g(s_a, M_a)$; // Equation 2.33
        $S_b \leftarrow g(s_b, M_b)$;
        $A_{\text{isnorm}} \leftarrow (E_a == 0)$ ? 0 : 1;
        $B_{\text{isnorm}} \leftarrow (E_b == 0)$ ? 0 : 1;
        if $\beta_A >= \beta_B$ then
            $K_a \leftarrow E_a - A_{\text{isnorm}}$;
            $K_b \leftarrow E_b - ebias - B_{\text{isnorm}}$;
        else
            $K_a \leftarrow E_a - ebias - A_{\text{isnorm}}$;
            $K_b \leftarrow E_b - B_{\text{isnorm}}$;
        $t \leftarrow S_a[i] << K_a + S_b[i] << K_b$;
        $R[i] \leftarrow KulToBM'(t)$;
    $\beta'_{share} \leftarrow \max(\beta_A, \beta_B)$;
    return $R$, $\beta'_{share}$;
```

The BM vector addition pseudocode is given in algorithm 3, with inputs and outputs being in BM format. The significands are first extracted. Next, K_a and K_b are computed. These are the shifts required for alignment, considering their exponents, denormalisation, and shared exponents. The sum is then computed in t. Finally, t is converted into BM format using *KulToBM'* (same as *KulToBM* with output format BM$\langle e, m\rangle$ instead of BM$\langle e_{norm}, m\rangle$) and returned along with the new shared exponent.

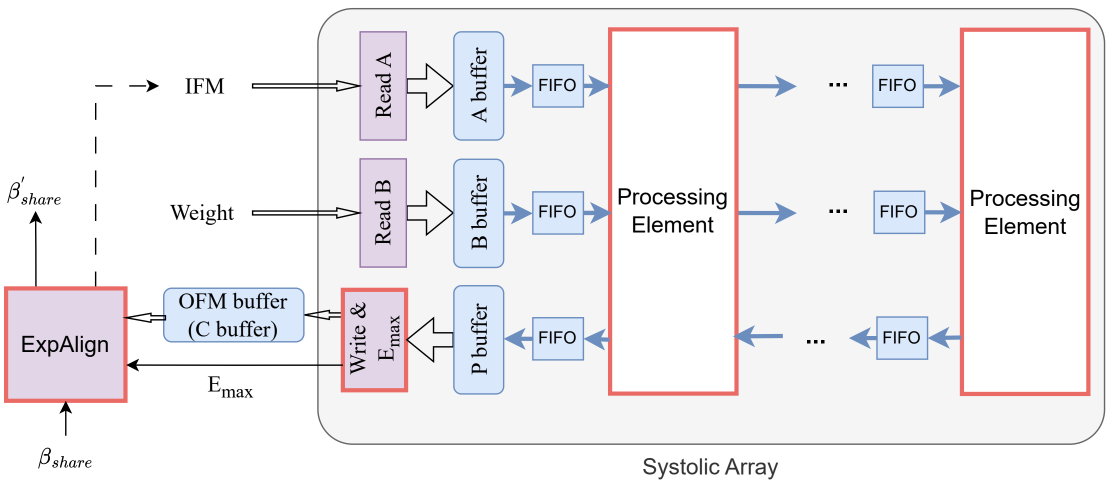


FIGURE 3.4: The BM systolic array implementation. The red block represents the BM related computation.

3.3 N-BEATS Inference Implementation

3.3.1 BM Systolic Array

The GEMM implementation is based on de Fine Licht et al. [31], modified to support BM. As illustrated in Figure 3.4, it is a 1D weight-stationary systolic array with compile-time configurable size. The configuration used is 32 PEs each with 16 parallel MAC units, and 512 memory tile size for both input matrices. The configuration tries to maximize the hardware performance and it supports matrix multiplication of $N_{\text{gemm}} \times N_{\text{gemm}}$ blocks where $N_{\text{gemm}} = 512$, to fit the matrix multiplication of the FC layer in N-BEATS. The input data are streamed through the PEs, with the width of this bus equal to the MAC per PE number (i.e. $16W$, where W is 8 or 16 bits for BM8 and FP16). Data transfers are double-buffered and will flow to the following GEMM block without interrupting computation.

As shown in Figure 3.4, the systolic array receives BM data as input (e.g., weights and input feature map (IFM) values). The inputs and outputs of each PE are connected to FIFOs. Before the systolic array computation, the BM data in the A buffer is stored into the corresponding PEs through the connected FIFO chain. During the computation, the BM data in the B buffer are pipelined and go through each PEs. After computation finishes, the result in each PE is written to the output FIFO first, and then pipelined out to the C buffer.

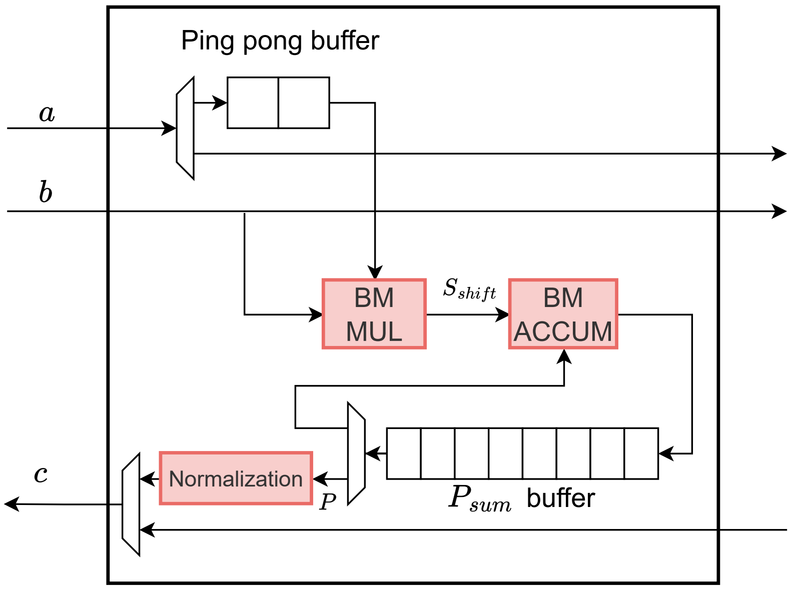


FIGURE 3.5: PE design of the BM systolic array. Red blocks represent the BM related computation.

Each PEs performs the BM inner product computation, as shown in Figure 3.5. The data from input a in each PE is weight-stationary and ping-pong buffered. The P_{sum} buffer stores the P_{sum} values, and after the inner-product computation finishes, the elements of P are converted to the $C\langle e_{norm}, m\rangle$ format through normalization block first and then are written to the FIFO connected to the PE output c. After that, the $C\langle e_{norm}, m\rangle$ format elements stored in the FIFO are pipelined out and written to the C buffer. E_{max} checking computation merges with the write-out block of the systolic array, so that the E_{max} checking latency can be overlapped.

An output buffer, such as output feature map (OFM) buffer, accepts the $C\langle e_{norm}, m\rangle$ format output from the systolic array. The $C\langle e_{norm}, m\rangle$ format data are then converted to the target $A\langle e, m\rangle$ format data through the *ExpAlign* block, and the *ExpAlign* block also generates the adjusted shared exponent values β'_{share}.

3.3.2 N-BEATS Inference Accelerator

Figure 3.6 illustrates the N-BEATS model architecture. It is composed of a number of N-BEATS blocks, each having 4 FC layers with ReLu activation, split into backcast and forecast branches. Each N-BEATS block is organised in a doubly residual manner as illustrated, where

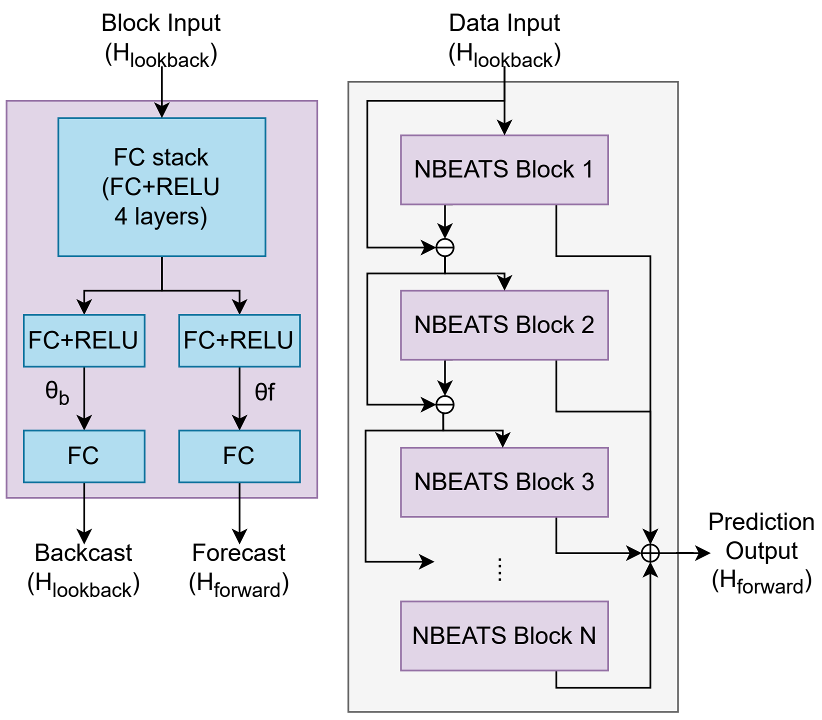


FIGURE 3.6: N-BEATS neural network model.

for Block 2-N the residual activation is the difference between the previous block's input and backcast output. All forecast outputs are summed to form the prediction output.

As shown in Figure 3.6, the left-hand side shows the N-BEATS block. The input of each block goes first to the 4-layer FC stack, and then the FC stack output activation goes to the backcast branch and forecast branch. The backcast output size is the same as the block input size, and the forecast output size is the same as the model output size. The right-hand side shows the N-BEATS model. A series of N-BEATS blocks are connected to each other. The input of each N-BEATS block is the differential value of the output and the input from the previous N-BEATS block, and the model output is the sum of all forecast values of the N-BEATS block.

The system-level architecture of our N-BEATS inference accelerator is illustrated in Figure 3.7. The weight, input dataset, and output tensors are initially stored in double data rate (DDR) memory. The IFM, OFM of each FC layer are buffers on the FPGA. The intermediate buffer (IM) buffer stores some intermediate results, including the backcast output, forecast sum, and

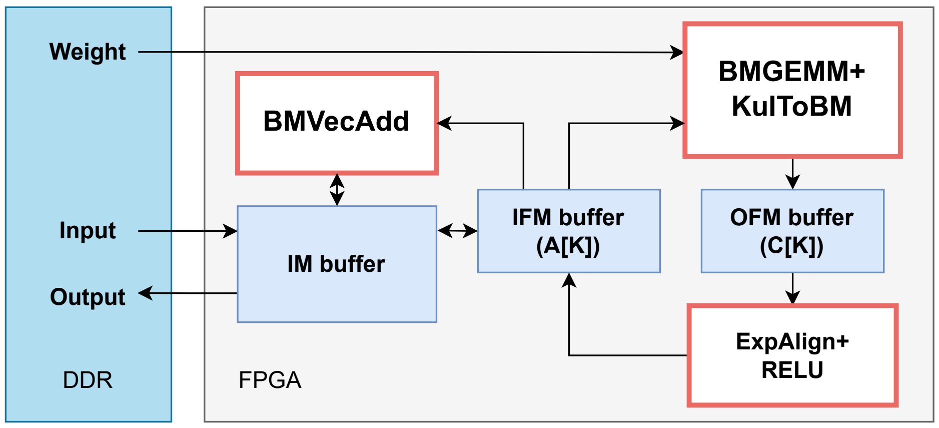


FIGURE 3.7: N-BEATS Inference Accelerator.

the FC stack output. *BMGEMM+KulToBM* is used to compute the FC layers and combines the GEMM block with integer outputs and *KulToBM* so that inputs and outputs are in BM format. The *BMVecAdd* block is used for computing residual inputs to each N-BEATS block and the final prediction output.

During inference computation, a batch of input data is fetched from DDR and written to the IM buffer. The same data is transferred to the IFM buffer to serve as input activations. Off-chip weights are passed to the *BMGEMM+KulToBM* block to compute each FC layer, and the result is placed in OFM. The OFM result is then aligned, ReLu applied, and written to IFM. The backcast residual computation and prediction output forecast sum are computed from the intermediate data in the IFM and written to IM.

The activation and weights of each layer are organised in separate $BM\langle e, m\rangle$ blocks. Each weight layer has a shared exponent bias, as does each activation layer.

3.4 Results

3.4.1 Experimental Configuration

We evaluated the performance of our implementation on the Xilinx Alveo U250 accelerator card with the Virtex UltraScale+ XCU250-L2FIGD2104E FPGA and 4 DDR memories. The inference accelerator is written in Vitis HLS 2020.2 and implemented using Vivado 2021.2

TABLE 3.1: N-BEATS Experiment Model Configurations

	Yearly	Quarterly	Monthly	Daily
$H_{forward}$	6	8	18	14
$H_{lookback}$	12	16	36	28
$dim(\theta^f)$	18	24	54	42
$dim(\theta^b)$	18	24	54	42
Samples	22850	64144	136603	187930

with a target frequency of 300 MHz. The BM data representation and computations, including Kulisch accumulation, conversion, and vector addition, are implemented using the HLS ap_int or ap_uint data types.

We use the model architecture and benchmarks from [130] with parameters detailed in Table 3.1. The model used M_{blk}=30 N-BEATS blocks, with the weight matrix $N_{\text{gemm}} \times N_{\text{gemm}}$ being 512×512 for all fully connected layers. Input data is arranged as a 2D array of dimension $B \times H_{\text{lookback}}$ where the batch size is $B = 512$ and a forecast horizon H_{forward} is 6, 8, 18, 14 for each seasonal pattern. The backcast length is $H_{\text{lookback}} = 2H_{\text{forward}}$. Experiments include training and validation on the M4 benchmark of seasonal patterns, i.e., M4-Yearly, M4-Quarterly, M4-Monthly, and M4-Daily. The other patterns, i.e., M4-Weekly, M4-Hourly, are discarded due to the limited number of samples available in the original dataset.

For our hardware implementation, we maintain the same input data size configuration as Table 3.1. To match the GEMM size, inputs, θ^b, θ^f are zero padded to N_{gemm}. Both weight and input data are stored in off-chip DDR memory encoded with BM8$\langle 2, 5 \rangle$ data type, and the IFM data type is BM8$\langle 2, 5 \rangle$, the OFM data type is BM12$\langle 6, 5 \rangle$. The Kulisch accumulator length $K_{\text{add}} = 25$, which is derived from Equation 3.5 with $W_I = 4, m_a = m_b = 5, e_a = e_b = 2$. The initial weight values and input data are generated in the conventional fashion in Pytorch (FP32 format) and converted to the BM8 format. The shared exponents are transferred to on-chip buffers.

TABLE 3.2: N-BEATS Accuracy (SMAPE LOSS) Comparison Across BM8, INT8, FP16

	Yearly	Quarterly	Monthly	Daily
FP32	13.462	11.992	11.278	3.869
FP16	13.454	11.986	11.273	3.868
INT8	19.661	18.713	15.279	9.489
BM8+round nearest	14.780	13.548	11.972	4.675

TABLE 3.3: MAC Unit Resource Utilisation and Power consumption across BM8, FP16, INT8

	LUT	DSP	FF	Power
FP16 multiply	44	2	34	1.582W
FP16 add	108	2	34	2.5W
BM8 multiply	47	0	0	0.524W
BM8 add	24	1	0	1.047W
INT8 multiply	35	0	0	0.328W
INT8 add	8	0	0	0.117W

3.4.2 Accuracy and Performance

Table 3.2 shows the symmetric mean absolute percentage error (sMAPE) for various data types using the Yearly, Quarterly, and Daily datasets from the M4 benchmark. This experiment was conducted using post-training static quantization [76]. he result shows that BM8 has a close sMAPE loss to FP16, but it gains a significant improvement on INT8.

Table 3.3 compares the hardware resources and power consumption of individual multipliers and adders for the different data types. All of the designs are synthesized from C, and power consumption is from the Vivado implementation report. It can be seen that the BM8 area and power consumption are close to INT8, and much smaller than FP16.

Table 3.4 shows the N-BEATS accelerator resource utilization, with the percentage being that of the total FPGA resources. Table 3.5 shows peak performance which is achieved when inputs, θ^b, θ^f are divisible by N_{gemm}. The BM8 and INT8 inference accelerator can operate at 300 MHz, but the FP16 accelerator can only satisfy a 200 MHz timing constraint. The BM8 design can achieve 277 GOPS performance, which is similar to INT8 and significantly better than the FP16 design. The total operations can be estimated by Equation 3.12 with

TABLE 3.4: N-BEATS inference accelerator resource consumption using BM8, INT8, FP16

	LUT	REG	BRAM	URAM	DSP
BM8	38131	50762	373	23	192
	2.45%	1.59%	15.67%	1.8%	1.56%
INT8	27112	38418	439	16	512
	1.74%	1.2%	18.45%	1.25%	4.17%
FP16	119246	157588	1019	0	2048
	7.66%	4.94%	42.82%	0	16.68%

TABLE 3.5: N-BEATS inference accelerator performance and power consumption using BM8, INT8, FP16

	Frequency	Latency	Peak Performance	Power
BM8	300MHz	0.232s	277 GOPS	21.44W
INT8	300MHz	0.228s	282 GOPS	21.97W
FP16	200MHz	0.354s	182 GOPS	22.674W

$2M_{blk}$ accounting for the number of operations per MAC and the number of N-BEATS blocks. The first term in braces accounts for the 3 FC layers; the next term for the FC layer input, backcast, and forecast branches; and the final term corresponds to the last 2 FC layers for the backcast and forecast. The computation for forecast sum and backcast residual is small and hence omitted in Equation 3.12. For the parameters in Table 3.1, performance (Op/Op_{peak}) varies from 38.7% to 41.2% of the peak, where Op_{peak} is achieved when $H_{\text{forward}} = H_{\text{loopback}} = N_{\text{gemm}}$.

$$\begin{aligned} Op \approx\ & 2M_{\text{blk}}\{3N_{\text{gemm}}^3 + N_{\text{gemm}}^2(2H_{\text{forward}} + 3H_{\text{lookback}}) \\ & + BN_{\text{gemm}}(H_{\text{forward}} + H_{\text{lookback}})^2\} \end{aligned} \tag{3.12}$$

The power consumption in Table 3.5 represents the total system power consumption, encompassing both dynamic and static power, as reported in the Vivado implementation. All three implementations are configured with two DDR banks, with one bank for input/output and the other for weights. The power consumption of BM8 and INT8 is slightly better than FP16. Using a single DDR bank results in the same performance but decreases power consumption by 11%.

3.5 Summary

In this chapter, we presented an HLS C library for the implementation of matrix multiplication using BM arithmetic. This was used to implement an FPGA-based N-BEATS accelerator, which can achieve a peak performance of 277 GOPS on an Xilinx Alveo U250 board. The performance in terms of power, area, and throughput is similar to INT8 (282 GOPS) with accuracy similar to FP16 (182 GOPS).

N-BEATS acceleration is ideally suited to acceleration via a GEMM accelerator because its computational bottleneck is fully connected layers. In future work, we will explore inference and training using BM arithmetic with other deep neural networks.

CHAPTER 4

Block Minifloat Artithmetic for Training

4.1 Introduction

FPGAs have been widely used for DNNs inference applications due to their ability to implement domain-specific data paths to optimize latency and power consumption. To date, most research has focused on inference, with training done offline using server-based equipment such as GPUs and TPUs. Edge-based training offers the potential for neural networks to adapt to local conditions, but at present, the additional computational and hardware cost is an obstacle to its adoption.

Unfortunately, it requires significantly more computation than inference, as well as high power consumption, meaning there are far fewer implementations. Custom number systems can be used to help address this challenge. Both 8-bit [15, 157], and 4-bit [158] quantization methods have been proposed, and some new number systems, such as LNS-madam [197], FP8 [114], and BFP [193] have been designed specifically for low-precision implementation.

Chapter 3 has demonstrated the advantage of BM arithmetic with the global block towards INT8 on accuracy and FP16 on hardware performance. In this chapter, we further introduce the local block and configurable precision in BM arithmetic to improve the training accuracy. Moreover, we propose an efficient and flexible implementation of a BM accelerator for DNN training. The design is tailored to an FPGA, and optimized to accelerate the N-BEATS deep neural network for time series prediction. Applications of the accelerator include anomaly detection, high-frequency trading, and sensor value prediction. The contributions of this chapter can be summarized as follows:

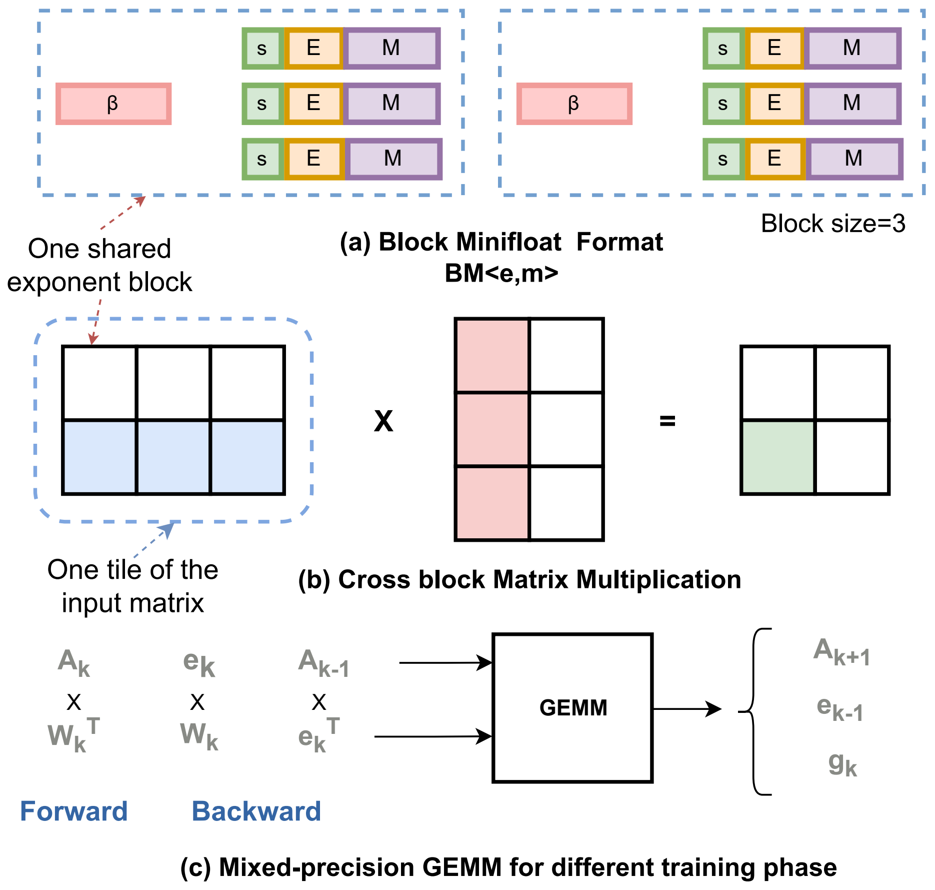

s
E
M
β
Block size=3
One shared exponent block
(a) Block Minifloat Format BM<e,m>
X
=
One tile of the input matrix
(b) Cross block Matrix Multiplication
A_k x W_k^T
e_k x W_k
A_{k-1} x e_k^T
GEMM
A_{k+1}
e_{k-1}
g_k
Forward
Backward
(c) Mixed-precision GEMM for different training phase

FIGURE 4.1: Summary of the main ideas introduced in this paper. (a) BM format with block size 3. (b) cross-block matrix multiplication. The shaded parts show a single block output obtained by multiplying 1×3 blocks by 3×1 blocks. (c) GEMM kernel. Forward and backward passes are processed by the same GEMM accelerator at different precisions.

- BM arithmetic introduces a new parameter, the local block. To achieve a balance between performance and accuracy, it should be possible to optimise block size independently of the GEMM tile size. We propose a novel cross-block BM GEMM unit that allows independent block and tile sizes and utilizes a higher precision buffer to improve accuracy (Figure 4.1(b)).
- The optimal precisions for the forward and back-propagation steps in training are different. We present a new BM GEMM kernel that supports runtime configuration of precision (Figure 4.1(c)).
- A novel, packed 4-bit BM MAC unit that can compute six independent multiplications per DSP is presented.

- We present the first FPGA implementation of 4-bit BM, mixed-precision neural network training. To the best of our knowledge, other reported FPGA implementations have used higher precision. On an Alveo U50 board, when training the N-BEATS network, we achieve a power efficiency of 42.4 Gops/W, this being a 3.12x improvement over an Nvidia GTX 1080 GPU.

4.2 Background

4.2.1 Forward and Backward Pass for an N-BEATS Block

N-BEATS was the first work to empirically demonstrate that pure deep learning could be applied to time-series analysis and achieve state-of-the-art results [130]. It improved forecast accuracy by 11% over a statistical benchmark and by 3% over the 2020 winner of the M4 competition, which used a hybrid statistical/neural residual/attention dilated LSTM stack. N-BEATS was chosen for the present study due to its accuracy and regular architecture, which is based on fully connected layers. As the computational bottleneck in N-BEATS and the widely studied Transformer architectures are both GEMM, so the BM training accelerators described in this chapter directly apply to Transformers.

N-BEATS is applied for discrete time series forecast tasks. Given a length observed in history $\boldsymbol{y}_{in} = [y_1, \ldots, y_{H_b}]$, the task is to forecast the future time series $\boldsymbol{y}_{out} = [y_{H_b+1}, \ldots, y_{H_b+H_f}]$.

Figure 4.2 illustrates the N-BEATS architecture [130], the forward pass and error propagation (backward pass) for an N-BEATS block. $\boldsymbol{y}_{in}$ is the input and $\boldsymbol{y}_{out}$ is the forecast of the network. As shown in Figure 4.2(b), it comprises several N-BEATS blocks, each having four FC layers with ReLu activation, split into backcast and forecast branches.

The forward pass of an N-BEATS block is shown in Figure 4.2(a). We use n and k as block and layer indices for an N-BEATS block and the N-BEATS block indices respectively, where M_{blk} is the number of N-BEATS blocks, $n = 1, 2..., M_{blk}$, res_n is the residual vector to the nth N-BEATS block, and $A_{n,k}$ is the activation vector.

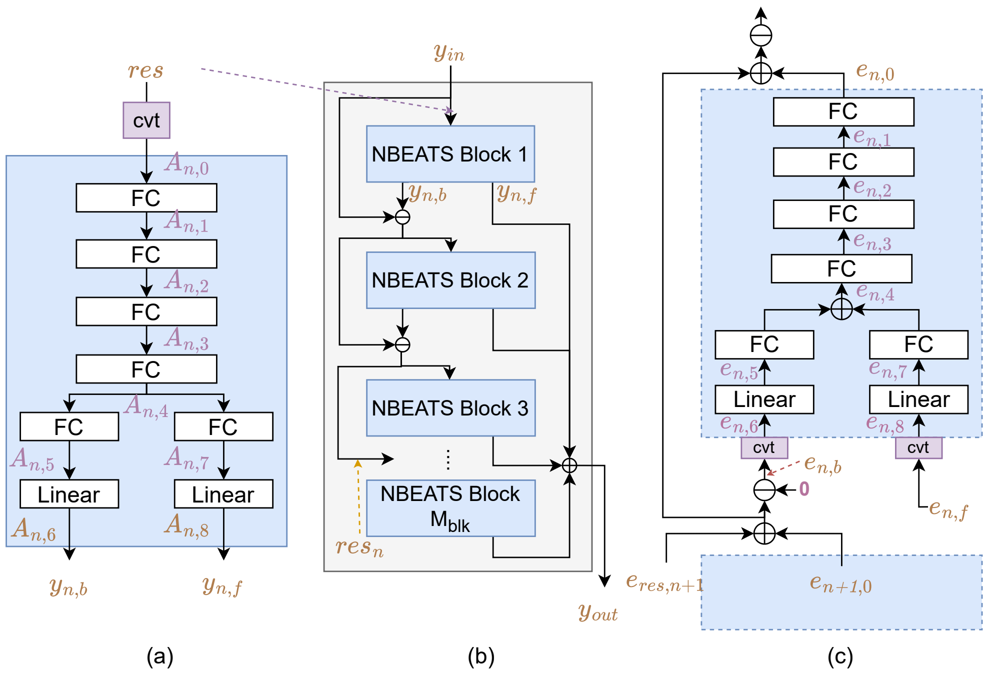


FIGURE 4.2: N-BEATS neural network model. Variables in yellow represent high-precision BM data (BM$\langle e_h, m_h \rangle$). Cvt block is the precision conversion block that converts high-precision data to low-precision data. (a) shows the N-BEATS block. The input of each block is first converted to low-precision format, then goes first to the 4-layer FC stack, and then the FC stack output activation goes to the backcast branch and forecast branch. (b) shows the N-BEATS model. (c) shows the error propagation of N-BEATS block.

A forward pass through N-BEATS block l involves propagation through a number of linear and FC layers,

$$A_{n,k} = \begin{cases} ReLu(z_{n,k}) & k = 0, 1, ..., 5, 7 \text{ (FC)} \\ z_{n,k} & k = 6, 8 \text{ (Linear)} \end{cases} \tag{4.1}$$

where $z_{n,k} = A_{n,k-1} W_{n,k}^T$ is a linear projection.

$A_{n,6}$ is the backcast branch output $y_{n,b}$; $A_{n,8}$ is the forecast branch output $y_{n,f}$. $y_{n,b}$ length is the same as the block input length H_b, and $y_{n,f}$ length is the same as the model output length H_f. The intermediate result $A_{n,5}$, $A_{n,7}$ in the forecast and backcast branches has a length of $H_b + H_f$.

For the residual computation, the backcast residual res of the nth N-BEATS block and the prediction output of the model are:

$$\begin{aligned} res_n &= in - \sum_{n=1}^{n=n} y_{n,b}, \\ y_{out} &= \sum_{n=1}^{n=M_{blk}} y_{n,b} \end{aligned} \tag{4.2}$$

N-BEATS uses mean absolute percentage error (MAPE) loss for training and sMAPE loss for validation.

$$\begin{aligned} C_{\text{MAPE}} &= \frac{1}{H}\sum_{i=1}^{H}\frac{|l_i - p_i|}{|l_i|} \\ C_{\text{SMAPE}} &= \frac{200}{H}\sum_{i=1}^{H}\frac{|l_i - p_i|}{|l_i| + |p_i|} \end{aligned} \tag{4.3}$$

where l_i is the actual value in time step i, and p_i is the scalar predicted value in time step i.

During training, back-propagation through each layer involves a repeated application of the chain rule [66]. The N-BEATS block error propagation is illustrated in Figure 4.2(c). The error of $y_{n,f}$ is defined as:

$$e_{n,f} = \frac{\partial C}{\partial y_{out}} = \begin{cases} -\frac{1}{Hl_i} & p_i \leq l_i \\ \frac{1}{Hl_i} & p_i > l_i \end{cases} \tag{4.4}$$

the error of $y_{n,b}$ is defined as:

$$e_{n,b} = \frac{\partial C}{\partial y_{n,b}} = -(e_{\ell+1,res} + e_{\ell+1,0}) \tag{4.5}$$

For each FC or Linear layer, the propagated error is

$$e_{n,k} = \frac{\partial C}{\partial A_{n,k}} = \begin{cases} (\frac{\partial C}{\partial A_{n,k+1}} \circ \sigma'_{n,k}) \times W_{n,k} & k = 0, 1, ..., 5, 7 \\ e_{n,6} \times W_{n,k} & k = 6 \\ e_{n,8} \times W_{n,k} & k = 8 \end{cases} \tag{4.6}$$

where $\sigma'_{n,k}$ is the derivation of ReLu function $\frac{\partial ReLu}{\partial z_{n,k}}$. We define $(x \circ y)_{ij} = x_{ij}y_{ij}$.

TABLE 4.1: Summary of mixed-precision FC/Linear layer computation

Index	In1	In2	Eq.	Out
①	$A_{n,0}$ BM$\langle e_{in}, m_{in}\rangle$	$W^T_{n,1}$ BM$\langle e_w, m_w\rangle$	(4.1)	$A_{n,1}$ BM$\langle e_{act}, m_{act}\rangle$
②	$A_{n,k-1}$ BM$\langle e_{act}, m_{act}\rangle$	$W^T_{n,k}$ BM$\langle e_w, m_w\rangle$	(4.1)	$A_{n,k}$ BM$\langle e_{act}, m_{act}\rangle$
③	$A_{n,k-1}$ BM$\langle e_{act}, m_{act}\rangle$	$W^T_{n,k}$ BM$\langle e_w, m_w\rangle$	(4.1)	$y_{n,b}$ *or* $y_{n,f}$ BM$\langle e_h, m_h\rangle$
④	$e_{n,k+1} \circ \sigma^{'}_{n,k}$ BM$\langle e_e, m_e\rangle$	$W_{n,k}$ BM$\langle e_w, m_w\rangle$	(4.6)	$e_{n,k}$ BM$\langle e_e, m_e\rangle$
⑤	$e_{n,1} \circ \sigma^{'}_{n,1}$ BM$\langle e_e, m_e\rangle$	$W_{n,1}$ BM$\langle e_w, m_w\rangle$	(4.6)	$e_{n,0}$ BM$\langle e_h, m_h\rangle$
⑥	$(e_{n,k})^T$ BM$\langle e_e, m_e\rangle$	$A_{n,k-1}$ BM$\langle e_{act}, m_{act}\rangle$	(4.7)	$g_{n,k}$ BM$\langle e_g, m_g\rangle$

The weight gradient for each FC layer and Linear layer is computed as follows.

$$g_{n,k} = \frac{\partial C}{\partial W_{n,k}} = (e_{n,k})^T \times A_{n,k-1} \tag{4.7}$$

Weight updates are done using SGD with learning rate α:

$$W^{'}_{n,k} = W_{n,k} - \alpha \cdot \frac{\partial C}{\partial W_{n,k}} \tag{4.8}$$

In the preceding description, forward and backward passes were given for a single input vector. In our actual implementation, inputs are processed in minibatches, with batch size B.

The forward and backward passes are mostly matrix multiplications as summarized in Table 4.1. The same table provides notation for the BM format applied to inputs, intermediate values, and outputs. Here BM$\langle e_h, m_h\rangle$ is high-precision format, others are low-precision format.

4.2.2 Stochastic Rounding

Stochastic rounding is used for weight updates [170, 193] to improve convergence in training and implemented as:

$$SRound(r) = \begin{cases} (-1)^s \times g(s, M+1) \times 2^{1-\eta} \times 2^{\beta} & 0 \leq rand \leq (M \times 2^{-m}) \\ (-1)^s \times g(s, M) \times 2^{1-\eta} \times 2^{\beta} & (M \times 2^{-m}) < rand \leq 1 \end{cases} \tag{4.9}$$

Here $rand$ is the random number between 0 and 1. In hardware, it's implemented with a linear feedback shift register (LFSR) generator. Thus SGD is implemented as:

$$W_{n,k}^{'} = W_{n,k} - \alpha \cdot SRound(\frac{\partial C}{\partial W_{n,k}}) \tag{4.10}$$

4.2.3 Related Work

Number Systems

Conventional training accelerators typically use 16 or 32-bit FP encoding (such as IEEE-754 single-precision, half-precision [73], or Google BFLOAT16 [3]). Different low-precision number systems have been applied to low-precision training, significantly improving the performance of hardware implementations. Seungkyu et al. [21] proposed a heterogeneous data type implementation for neural network training, using low-precision fixed-point activation/weight and half-precision error/gradient. A novel MAC architecture is designed to facilitate the heterogeneous data-type inputs configured with different precision levels of unbalanced bit-widths. The FP8 data type [15, 157] is another popular number system for low-precision training and uses 5-bit exponent and 2-bit mantissa, or 4-bit exponent and 3-bit mantissa data representations for training. Notably, FP8 was highlighted in a paper from NVIDIA, ARM, and Intel [114] in which they demonstrated the efficacy of the format on various training tasks, matching the accuracy of 16-bit approaches. However, FP8 was only applied to GEMM operations for fully connected and convolutional layers, with non-GEMM operations remaining in FP16 or BFLOAT16.

Examples include BM [40] and FP8 [114, 127], in which the latter utilizes a scaling factor for each layer of a Transformer neural network [125]. Previous work [199] has shown that BM can achieve similar accuracy to FP16 and with an area close to INT8 for inference. Microscaling [144] adopted a similar numerical format, but included special values such as NaN and Inf not present in block minifloat. The techniques in this paper could be directly used to implement Microscaling.

DSP Packing

DSP packing has been frequently used in neural network inference implementation to improve computational density in FPGAs, which have a fixed number of high-performance blocks. Xilinx proposed 8-bit integer [182] and 4-bit integer [181] for CNN inference. A recent paper [138] supports six 4-bit signed multiplications packed on one DSP for CNN inference. [175] enables four 8-bit FP multiplications on one DSP. The present work extends this idea by supporting different precision formats, as illustrated in Figure 4.11, and using it for training rather than inference.

FPGA Training Accelerators

Guo et al. [50] proposed a CNN training accelerator using sparsity and pruning that achieved 641GOP/s equivalent performance. Geng et al. [172] proposed the ALexNet training accelerator using the FPGA cluster. The power efficiency per FPGA is up to 3.4 times higher than the Tesla K80 GPU. Luo et al. [105] implemented an 8-bit CNN training accelerator by utilizing batch-level parallelism. Venkataramanaiah et al. [167] implemented a CNN accelerator for low-batch training. Finally, Guo et al. [48] described a BM$\langle 2, 5\rangle$ data type accelerator for single-batch CNN transfer training. This work differs from [48] in the use of mixed precision arithmetic to reduce precision loss from cross-block operations and utilizes DSP packing. To the best of our knowledge, it is the first reported FPGA-accelerated time-series forecasting network.

4.3 Implementation of BM Arithmetic

4.3.1 Cross-block BM Matrix Multiplication

Previous work [30, 114] used FP32 format as the high-precision accumulator and computed the inner-product in a conventional FP process. In this work, we choose an integer-based implementation, as described in Figure 4.3. Compared to FP based accumulation, it has fewer steps with a shorter latency than FP operation [199], and integer-based mixed precision implementation is less complicated than FP based mixed precision one.

Figure 4.3 provides a simplified example of a BM matrix multiplication. All inputs and outputs are in BM format, and it accepts two input matrices, A and B, and computes an output matrix C. P is a two-dimensional wide integer buffer of size $B_k \times B_k$ and holds an entire block of C. In cross-block matrix multiplication, matrices A and B are partitioned into multiple blocks of size $B_k \times B_k$, and each block has a shared exponent bias β_a^{uw} and β_b^{wv}, where u, v, w are block indices. The minifloat elements in A, a_i, are in BM$\langle e_a, m_a \rangle$ format, and the minifloat elements in matrix B are b_i.

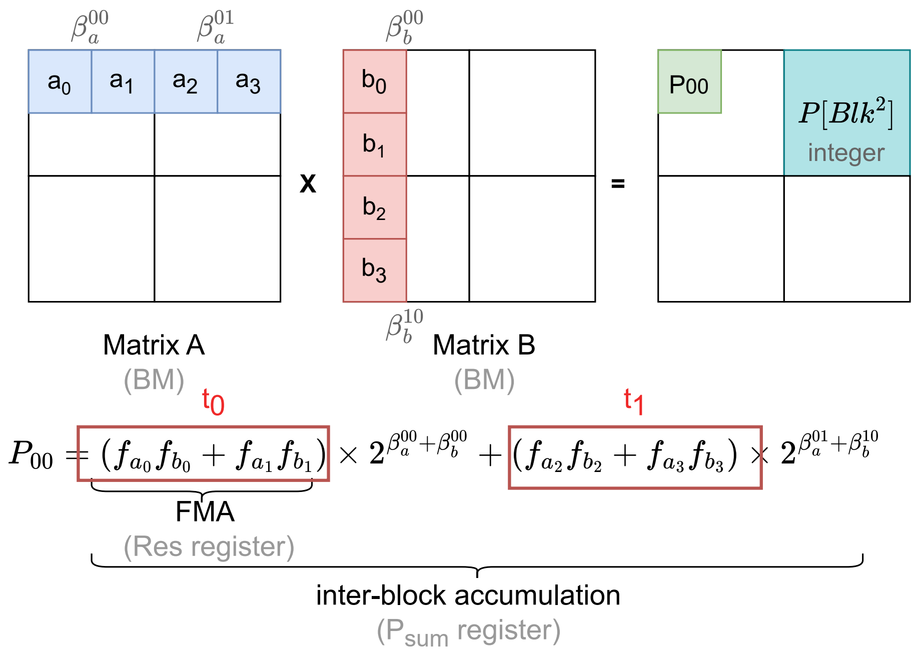
Computation Process

A block diagram of our implementation is illustrated in Figure 4.4 and detailed in the following paragraphs. The cross-block inner-product requires accumulation of FMA results (see Figure 4.3) and can be expressed mathematically as:

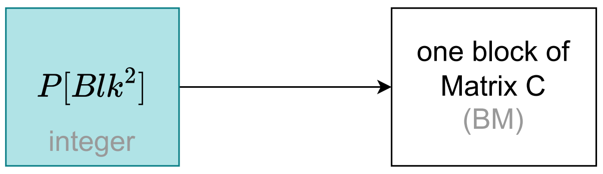
$$P_{ij} = \sum_{w} \{ \sum_{t=0}^{B_k-1} \text{fma}(A, B, t) \times 2^{\beta_a^{uw} + \beta_b^{wv}} \} \tag{4.11}$$

where fma(A, B, t) is the FMA result for block t, and P_{ij} is an element of the accumulation output block $P[B_k^2]$. During FMA computation, a_i and b_i are converted into integers first, and then doing MAC computation.

The β-comp block computes the shared exponent for the output block β_c^{uv} and the bias values Ka and Kb of the shared exponent used in inter-block accumulation.



(a) Inner Product



(b) Normalization

FIGURE 4.3: Simplified example of a BM GEMM requiring inner-product and post-inner-product (rescaling) computations. The matrices are 4×4 and each divided into 4, 2×2 blocks. (a) Illustrates the FMA, the inputs are converted to integer, multiplied and added together to form $t_0 = f_{a_0}f_{b_0} + f_{a_1}f_{b_1}$ and $t_1 = f_{a_2}f_{b_2} + f_{a_3}f_{b_3}$. In the inter-block accumulation step, the t_i values are aligned according to the shared exponent values and summed to give a high-precision inner-product P_{00}. The P_{ij} values for a block are saved in a wide integer buffer called $P[B_k^2]$. (b) The rescaling process is executed after all the inner products required for a particular block have been completed. The wide integer results are converted into BM format data and form part of the output matrix C.

The rescaling process converts the integer accumulation result into minifloat format data output. The accumulation results in $P[B_k^2]$ are wide integers in value and, in general, will exceed the dynamic range of the output minifloat representation. We find the maximum absolute value P_{max} of $P[B_k^2]$ and use it to determine a shift value, Z_{shift}, that avoids overflow.

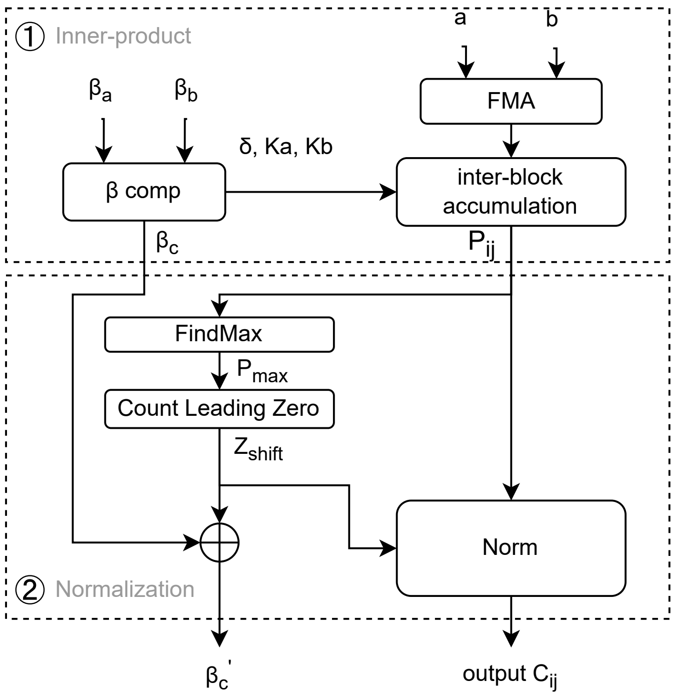


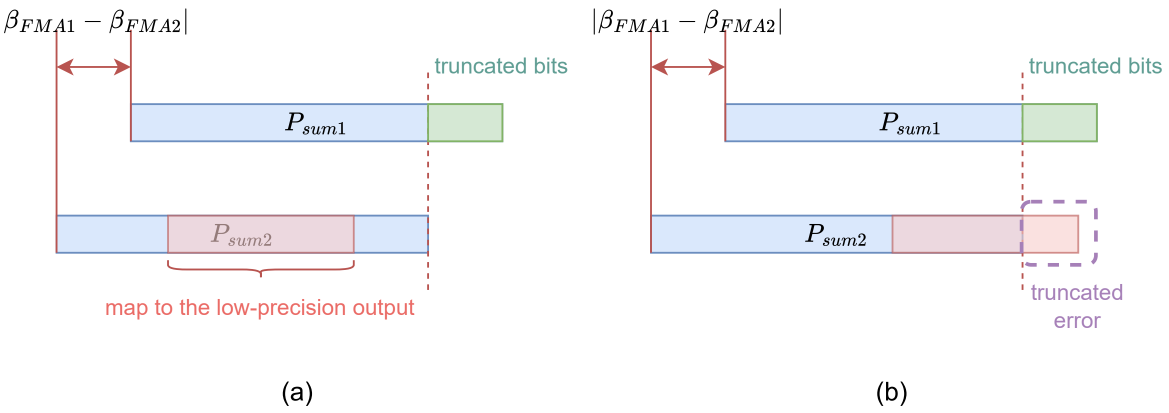
FIGURE 4.4: Block diagram illustrating cross-block BM matrix multiplication. The first step performs the minifloat FMA (FMA block) and shared exponent computation (β comp block), placing the result in $P[B_k^2]$. In the second step, each P_{ij} in $P[B_k^2]$ is examined to determine the appropriate shift for rescaling, which is performed in the Norm block. The resulting C output is in BM format.

Z_{shift} is used to adjust the output shared exponent and Norm block during conversion of the P_{ij} values in $P[B_k^2]$ to BM format in C_{ij}. The detail is provided in algorithm 4. Cnt_{denorm} and E_{dp} are constant boundary thresholds for rescaling.

In inter-block accumulation, aligned addition may introduce truncation error in the trailing bits of the smaller addend. Figure 4.5 illustrates this truncation error. When the integer accumulator P_{sum2} can accommodate the dynamic range of the low-precision output (as shown in Figure 4.5(a)), the truncation error remains unaffected—for example, in matrix multiplication with input and output precision BM$\langle 2, 5\rangle$. However, when P_{sum2} cannot cover the dynamic range of the low-precision output (as shown in Figure 4.5(b)), the truncated bits affect the output result, as seen in matrix multiplication with input precision BM$\langle 5, 2\rangle$ and output precision BM$\langle 2, 5\rangle$.

Algorithm 4: Accumulation buffer to BM$\langle e, m\rangle$ rescaling

```
Function Norm($P_{ij}$, $Z_{shift}$)
    $s \leftarrow (P_{ij} > 0)\ ?\ 0 : 1$; // Sign bit
    $P_{ij} \leftarrow |P_{ij}| \gg Z_{\text{shift}}$ ;
    $Z \leftarrow CLZ(P_{ij})$; // Check leading zeros
    if $Z < Cnt_{denorm}$ then
        // denormal and underflow
        $C_{ij} \leftarrow f(s, 0, P_{ij} \ll (Cnt_{\text{denorm}} - 1))$;
    else
        // normal
        $E \leftarrow E_{dp} - (Z + 1) + \eta_c$;
        $C_{ij} \leftarrow f(s, E, P_{ij} \ll Z)$;
    return $C_{ij}$
```



$\beta_{FMA1} - \beta_{FMA2}|$
truncated bits
P_{sum1}
P_{sum2}
map to the low-precision output
(a)
$|\beta_{FMA1} - \beta_{FMA2}|$
truncated bits
P_{sum1}
P_{sum2}
truncated error
(b)

FIGURE 4.5: The truncation error in aligned addition computation.

P_{sum} is designed with sufficient word length to ensure that C_{ij} does not overflow. Its word length is defined as follows:

$$K_{\text{add}} = 1 + (2^{e_a} - 1 + m_a) + (2^{e_b} - 1 + m_b) + W_{ex} + W_{tail} \tag{4.12}$$

Here, K_{add} denotes the word length of P_{sum}, W_{ex} represents extra bits added to prevent overflow. W_{tail} tailing bits are needed to avoid truncation as illustrated in Figure 4.5(b). The tailing bit length W_{tail} should cover the dynamic range of low-precision output. W_{ex} and

W_{tail} can be determined as follows:

$$W_{ex} \leq \lceil log_2(wB_k) \rceil$$
$$W_{tail} = \begin{cases} 0 & m_a + m_b \geq m_c \\ m_c - (m_a + m_b) & m_a + m_b < m_c \end{cases} \tag{4.13}$$

Here, wB_k is the number of MAC operations in the inner product.

Error Analysis

The computational error in the cross-block matrix multiplication is the rounding error, which is generated when converting the accumulation results to the low-precision output format during rescaling. Denoting the rounding error as ϵ_3, the rounding error bound can be determined as follows:

$$\epsilon_3 \leq 2^{-m_c} \tag{4.14}$$

Mixed Precision

The mixed precision computation process is the same as the single precision described in Figure 4.4, except for some changes in input and output. For input data, an extra mixed precision decoder is applied when converting a_i and b_i to integers according to different precision configurations. During rescaling, the rescaling process still follows algorithm 4, but there are several selectable boundary thresholds used to convert the integer accumulation result to different target precision outputs.

4.3.2 BM Vector Addition

The BM vector addition is used in residual computation, weight update, and error addition. The error addition is the back propagation of the branch structure inside N-BEATS blocks.

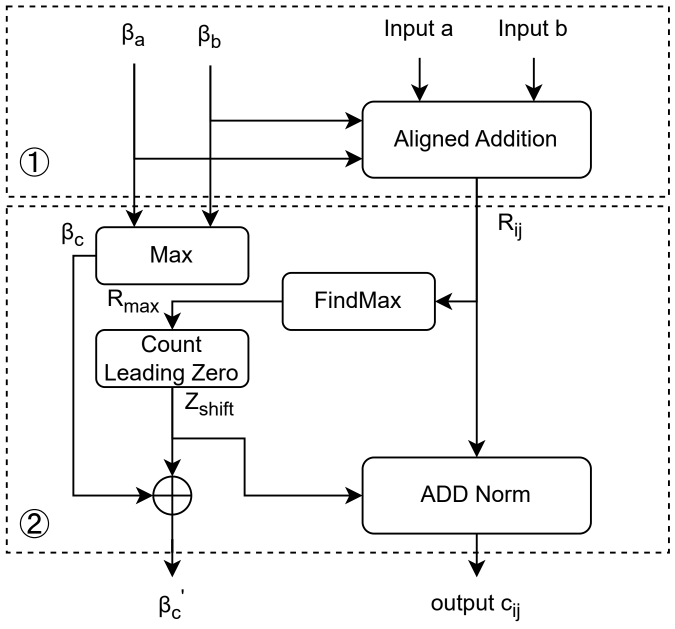


FIGURE 4.6: BM vector addition arithmetic. The first step converts the inputs a and b into integers, aligns them, and sums. The output R_{ij} is placed in the signed integer block $R[B_k^2]$. The second step computes the output shared exponent β_c. If there is overflow, it calculates the Z_{shift} value and adjusts β_c, then the norm block converts integer values in $R[B_k^2]$ into BM format.

The BM addition is also conducted in integer format and computed as follows:

$$a \pm b = \begin{cases} (S_a 2^{E_a} \pm S_b 2^{E_b} * 2^{\beta_B - \beta_A}) * 2^{\beta_A}, \beta_A \geq \beta_B \\ (S_a 2^{E_a} * 2^{\beta_A - \beta_B} \pm S_b 2^{E_b}) * 2^{\beta_B}, \beta_A < \beta_B \end{cases} \tag{4.15}$$

where S_a, S_b are the significands of a and b (Equation 2.33).

The addition process is summarized in Figure 4.6. The inputs a and b are converted to integers, aligned with the bias of shared exponent $|\beta_B - \beta_A|$ according to Equation 4.15, and added. The output is the signed integer R_{ij} in block $R[B_k^2]$, and the output shared exponent $\beta_c = max(\beta_a, \beta_b)$. Then, the maximum absolute value R_{max} of $R[B_k^2]$ is checked for overflow and used to calculate the Z_{shift} value if overflow occurs. Finally, The $ADD\ Norm$ converts integer value R_{ij} to BM format output c_{ij}.

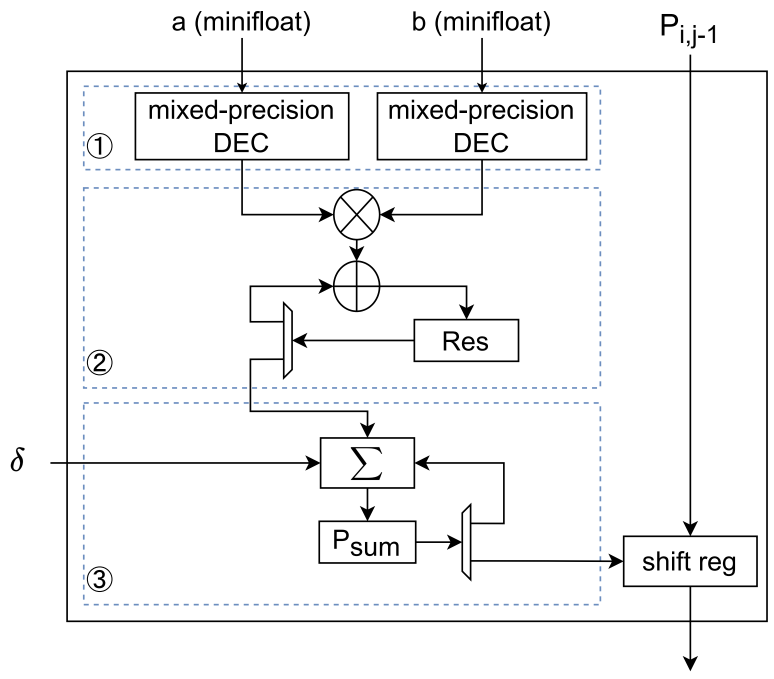


FIGURE 4.7: PE design of GEMM kernel. The PE is responsible for the scalar inner-product computation in Figure 4.3.

4.3.3 BM GEMM Kernel Architecture

The BM GEMM kernel, illustrated in Figure 4.8, is a 2D output-stationary systolic array [183]. It receives parallel input blocks A and B in minifloat format, as well as their shared exponents β_a and β_b.

PE Design

As shown in Figure 4.7, the PE performs the scalar inner product, described in Figure 4.4. The input formats for a and b can be different minifloat formats, configurable at run time. The first step is the mixed precision decoder. The second step is FMA computation, and step 3 is inter-block accumulation. δ is the bias of shared exponent between two blocks, which is calculated in β-comp block. After the inner product computation finishes, P_{ij} for each PE is streamed out of the PE array using the shift register.

BM GEMM Kernel

The GEMM has $T \times T$ PE blocks in PE array. The tile size T of the GEMM is sized to get the best computation performance while constrained by on-chip resources, while the block size B_k is set small to get better training accuracy. The GEMM design supports configurable block size under the assumption that the block size is smaller than or equal to the tile size. The example in Figure 4.8 has a block size of $\frac{T}{2} \times \frac{T}{2}$. The PE array computes the inner product of a tile matrix. The β block in PE array corresponds to the β comp block in Figure 4.4.

After the inner-product computation finishes, the P_{sum} in each PE is pipelined out and written to the $P[B_k^2]$ buffer. The shared exponent result is also pipelined out through β blocks. $P[B_k^2]$ buffer is a ping-pong buffer. Then the FindMax computation is done in parallel by streaming P_{sum} through columnar $FindMax$ units and reduced by shifting to the leftmost column to obtain a single result P_{max}. $Calibr$ block calculates the shared exponent result β_c' and Z_{shift}. Finally, the Norm block converts P_{ij} values in $P[B_k^2]$ buffer into BM output data C.

GEMM Pipeline Organization

As shown in Figure 4.9, execution is divided into four stages. If the GEMM input matrix sizes are $H_{row} \times K$ and $K \times H_{col}$, the latency of naive implementation is approximately:

$$l_{naive} = \left\lceil \frac{H_{row} \times H_{col}}{T \times T} \right\rceil \times (K + 3 \times T) \tag{4.16}$$

The pipeline is optimized in two ways. Firstly, the inner product and rescaling computations of the two tiles have no data dependency and execute in parallel. Secondly, inside the rescaling block (steps 2-4 in Figure 4.9), there is also no data dependency between different accumulation result blocks vertically in the PE array. As shown in Figure 4.9(b), each stage inside the rescaling is pipelined, which can reduce latency to $2B_k + T$. Thus, the inner product computation can be pipelined with rescaling. The latency of the pipelined BM GEMM is:

$$l_{\text{pipe}} = \left\lceil \frac{H_{row} \times H_{col}}{T \times T} \right\rceil \times K + 2B_k + T \tag{4.17}$$

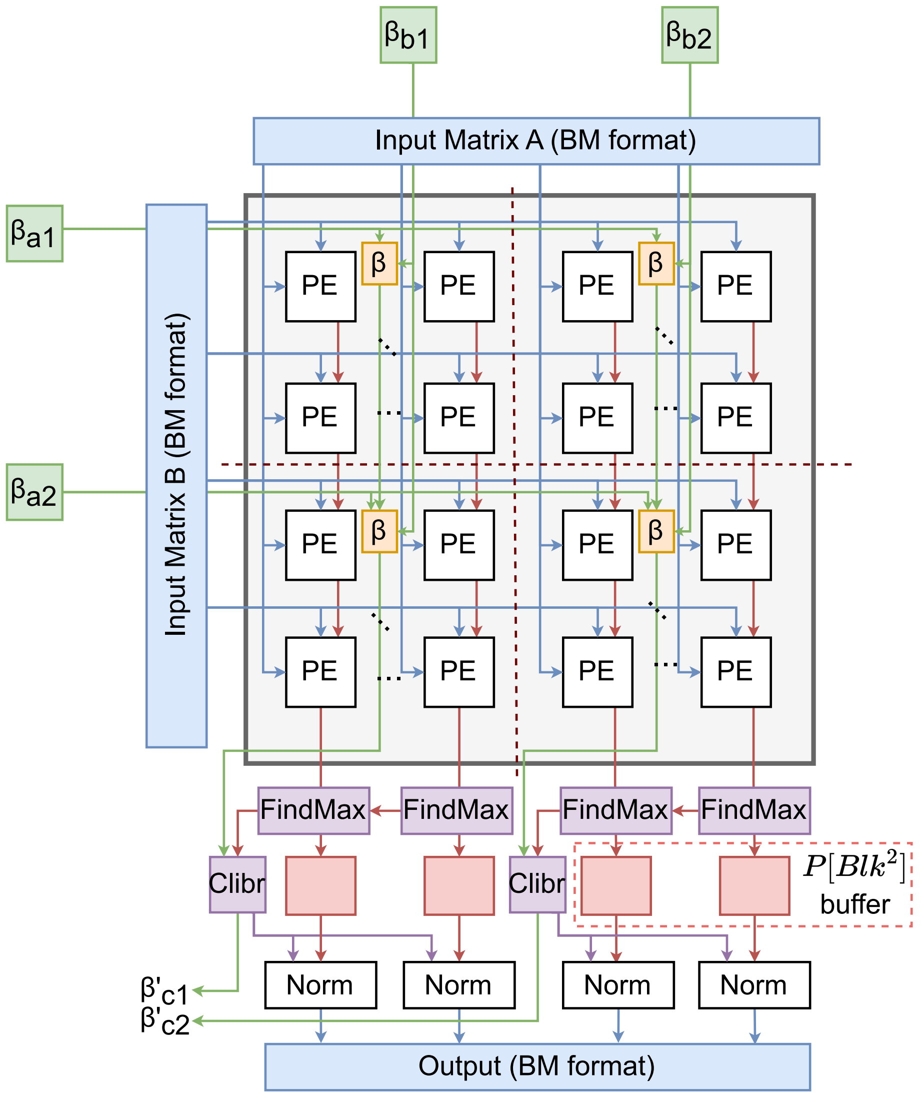
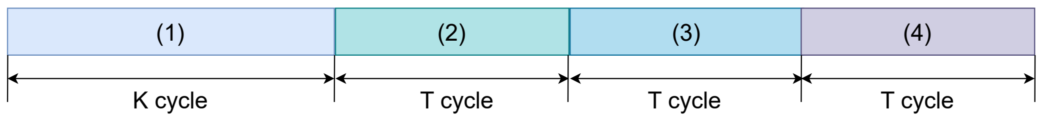


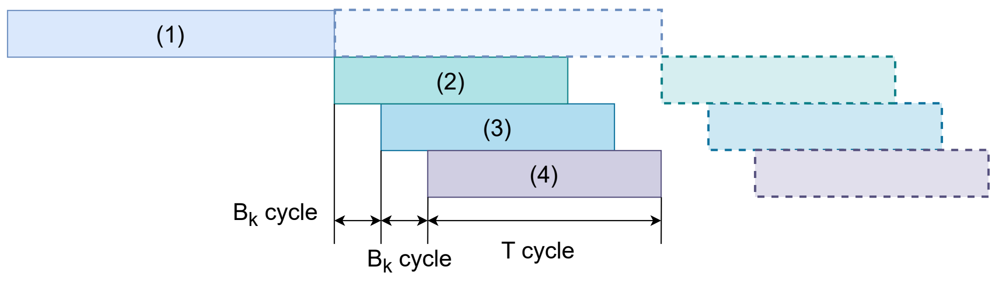
FIGURE 4.8: Block size configurable GEMM kernel with tile size $T \times T$ ($T = 4$). The red dashed lines represent the boundaries of the block, and the block size in this example is $\frac{T}{2} \times \frac{T}{2}$. The GEMM kernel is the implementation of Figure 4.4

4.3.4 DSP Packing for 4-bit BM Training

To increase the computational density of training, we optimize the arithmetic implementation with DSP packing for 4-bit BM by packing the multiply operation. We propose a DSP packing scheme and DSP-packed PE design. In contrast to previous works for inference [138, 175, 181, 182], our DSP packing technique supports different precisions during training.



(a) Naive implementation of BM GEMM



(b) Pipelined implementation of BM GEMM

FIGURE 4.9: Pipeline organization for BM GEMM. (a) is a naive implementation, (b) is the fine-grained pipeline for BM GEMM. (1) is inner product; (2) is Find Max in each column, and stream P_{sum} to $P[B_k^2]$ buffer; (3) is Find Max P_{max} in each block by left shifting, and calculate Z_{shift}, $\beta_c^{'}$ in Clibr block; (4) is Norm.

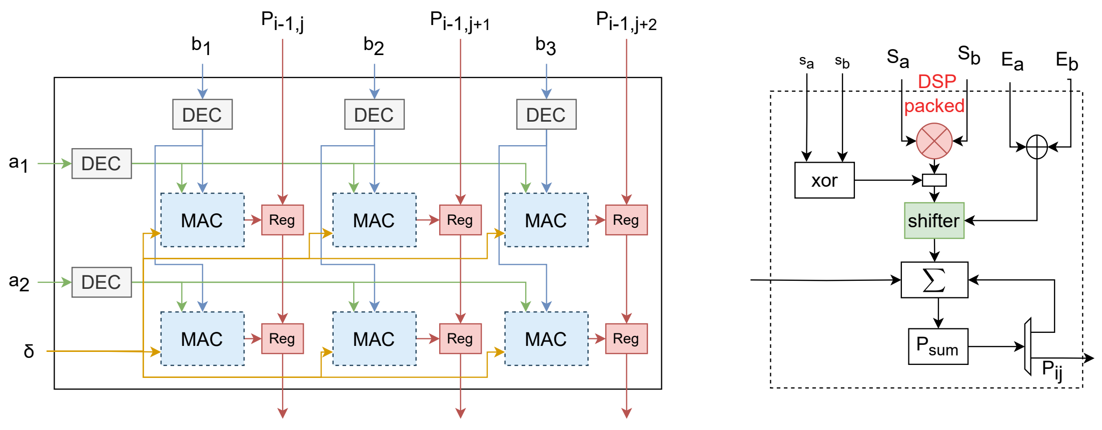
DSP Packed PE Design

The DSP-packed PE design for 4-bit BM is illustrated in Figure 4.10. Each PE takes two input a_i values and three input b_is. There are six MAC units for one PE, and each PE uses one DSP. In the MAC unit implementation, due to its small data word length, some computations such as exponent addition, mixed precision decoder, and sign bit xor operation are implemented as bit operations using lookup table (LUT) resources.

Our approach for the case of the 4-bit BM inner product is to pack six integer operations in a single DSP as illustrated in Figure 4.11. Multiplication of significands is performed on the DSP, and the remaining parts, such as addition and shifting, are implemented with LUTs.

In our implementation of N-BEATS, the weight, error, and gradient use a 4-bit BM format. Figure 4.11(a) enumerates all possible configurations for the 4-bit BM format data. In particular, significand lengths of 1 bit to 3 bits need to be supported.

A maximum of 3 bits is needed for one of the significands of a 4-bit BM input, and the multiplier output is 6 bits. To increase accuracy, we use unsigned BM$\langle 0, 4\rangle$ for activations



(a) DSP packed PE design for 4-bit BM

(b) MAC unit in PE

FIGURE 4.10: DSP packed PE design for 4-bit BM MAC. (a) is the PE design, the DEC block is the mixed precision decoder, and the Reg block is the shift register to pipeline out the accumulation result P_{ij}. (b) is the MAC unit design.

$A_{n,k}$ since it is the output of ReLu and non-negative; in this case, the multiplication needed is 4×3 bit, and the output is 7 bits.

As illustrated in Figure 4.11(b), input A of the DSP is used for two multiplier inputs and has a maximum word length of 25 bits. Input B is used for the other three inputs and has a maximum word length of 18 bits. The output, C, is populated with six 7-bit results.

The total word length needed to support our packing scheme is 25×18 bits; this can fit in a Xilinx DSP48E2, which supports 27×18 bit multiplies.

The MAC unit, used in each PE, is illustrated in Figure 4.10. Figure 4.11(c) described how the inner product is computed. The implementation still follows step one in Figure 4.4, but it merges the FMA and inter-block accumulation computation.

Rescaling and Rounding

The rescaling and rounding of DSP-packed GEMM is the same as the conventional BM GEMM, and the GEMM architecture also uses the architecture in Figure 4.8.

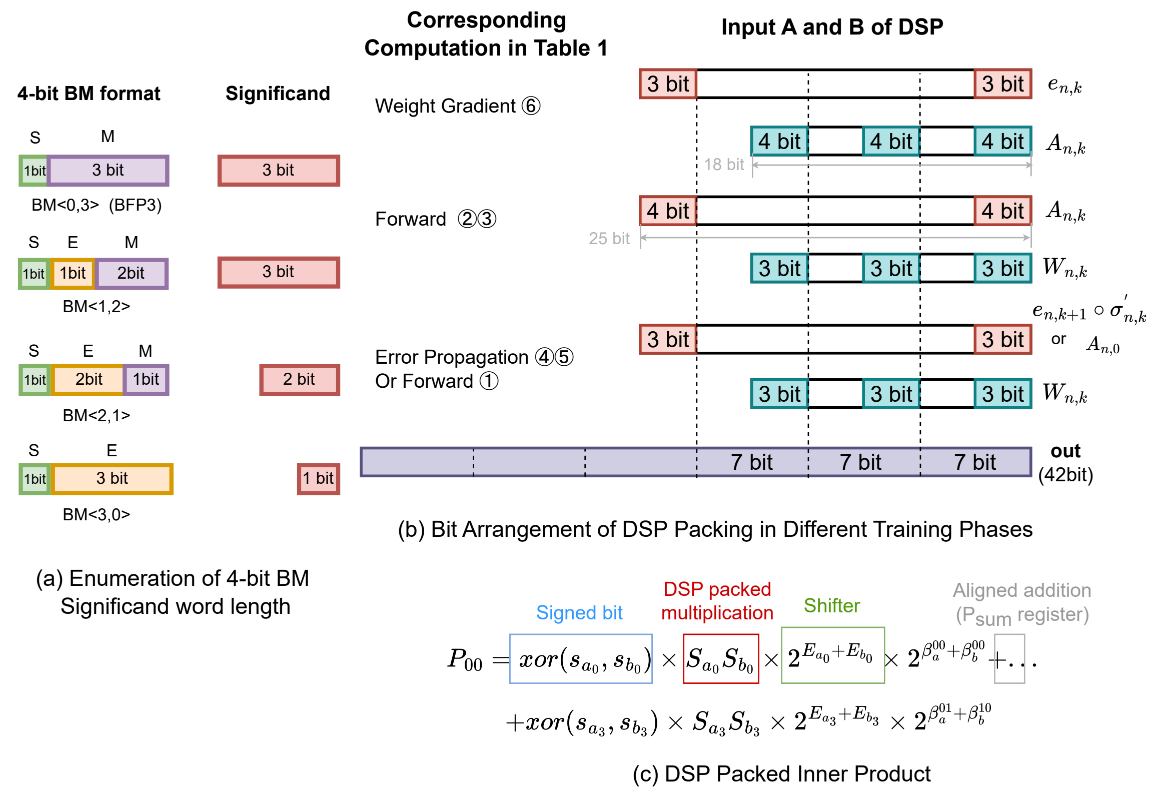

4-bit BM format
Significand
Corresponding Computation in Table 1
Input A and B of DSP
S M
1bit 3 bit
BM<0,3> (BFP3)
S E M
1bit 1bit 2bit
BM<1,2>
S E M
1bit 2bit 1bit
BM<2,1>
S E
1bit 3 bit
BM<3,0>
3 bit
3 bit
2 bit
1 bit
(a) Enumeration of 4-bit BM Significand word length
Weight Gradient ⑥
Forward ②③
Error Propagation ④⑤ Or Forward ①
3 bit 3 bit $e_{n,k}$
4 bit 4 bit 4 bit $A_{n,k}$
18 bit
4 bit 4 bit $A_{n,k}$
25 bit
3 bit 3 bit 3 bit $W_{n,k}$
$e_{n,k+1} \circ \sigma'_{n,k}$ or $A_{n,0}$
3 bit 3 bit
3 bit 3 bit 3 bit $W_{n,k}$
7 bit 7 bit 7 bit
out (42bit)
(b) Bit Arrangement of DSP Packing in Different Training Phases
Signed bit
DSP packed multiplication
Shifter
Aligned addition (P_{sum} register)
$P_{00} = xor(s_{a_0}, s_{b_0}) \times S_{a_0}S_{b_0} \times 2^{E_{a_0}+E_{b_0}} \times 2^{\beta_a^{00}+\beta_b^{00}} + \ldots$
$+xor(s_{a_3}, s_{b_3}) \times S_{a_3}S_{b_3} \times 2^{E_{a_3}+E_{b_3}} \times 2^{\beta_a^{01}+\beta_b^{10}}$
(c) DSP Packed Inner Product

FIGURE 4.11: DSP packing scheme for mixed precision 4-bit BM MAC. (a) is the bit arrangement for DSP packing in different training phases. The total number of multiplications is six, with two numbers in input A, three numbers in input B, and six 7-bit numbers in output C. (b) is the simplified example of DSP-packed inner product implementation. Based on Figure 4.3, the inner product merges the FMA and inter-block accumulation as one step.

4.4 N-BEATS Training Accelerator

4.4.1 Training Accelerator Design

A block diagram of the N-BEATS training accelerator system is given in Figure 4.12. Weights, activations (used for back propagation), input data, and label values are stored in high bandwidth memory (HBM). Four separate HBM interfaces are used for independent access to weights, activations, input/label data, and the shared exponent for input/label data. Shared exponents of weight and activations are stored in FPGA BRAM. The training implementation algorithm is provided in algorithm 5. FC_forward() computes the FC layer forward pass, FC_backward() computes the error propagation, gradient computation, and weight update

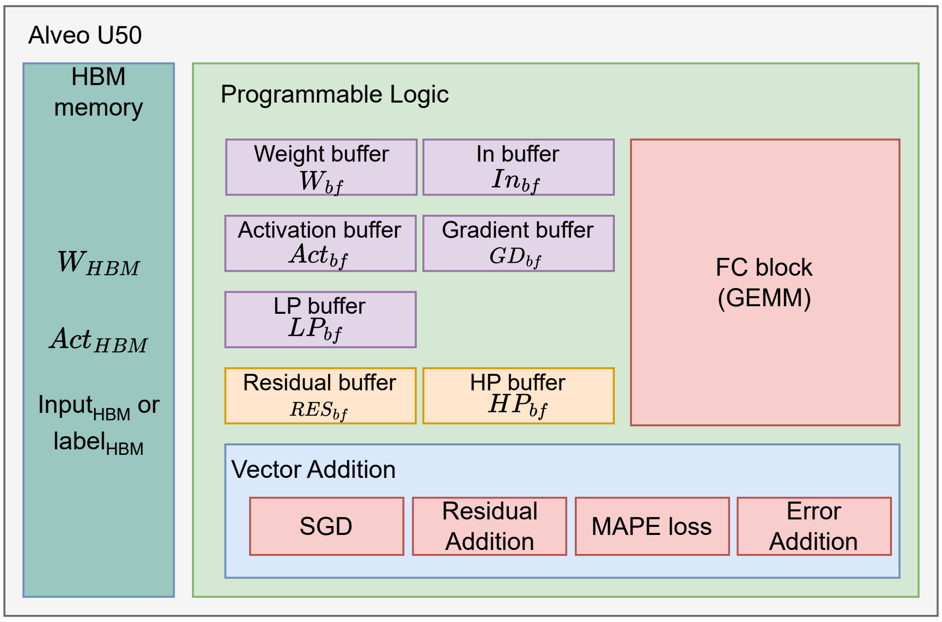


FIGURE 4.12: N-BEATS training accelerator. The computing block is marked with red; the low-precision BM buffer is marked purple; the high-precision BM buffer is marked yellow. FC block is illustrated in Figure 4.13

of FC layer in the backward pass. FC block computes FC_forward() and Error, Gradient computation of FC_backward() mentioned in algorithm 5. Weight update of FC_backward() is computed in SGD block.

The on-chip buffer sizes and precisions are detailed in Table 4.2. In this table, the largest FC layer weight size in N-BEATS is $L_k \times L_k$, where L_k is an N-BEATS size parameter specified in Section 4.5.4. The LP buffer stores the low-precision output of the FC block (Figure 4.13). HP buffer stores high-precision output (BM$\langle e_h, m_h\rangle$) from the FC block, such as residual values. In forward pass, the Residual buffer RES$_{bf}$ is used to store the backcast and forecast residual values, RES$_{bf}[0]$ and RES$_{bf}[1]$ in algorithm 5 represents different area of the buffer to store backcast and forecast branch residual data. In back-propagation, the Residual buffer stores the error of two branch residuals. Forward and backward passes are done layer-by-layer with weights, activations, and error values of each layer being loaded to the appropriate on-chip buffers from HBM; updated, and written back to HBM.

Algorithm 5: Forward and Back Propagation of N-BEATS block

```
Function NBEATS_training
    $RES_{bf}[0] \leftarrow Input_{HBM}$;
    repeat $M_{blk}$ times
        NBEATS_block_forward();
    $RES_{bf}[1] \leftarrow MAPE(label_{HBM},\ RES_{bf}[1])$; //Equation 4.3
    repeat $M_{blk}$ times
        NBEATS_block_backward();

Function NBEATS_block_forward
    $In_{bf}[0] \leftarrow Cvt(RES_{bf}[0])$;
    $Act_{HBM} \leftarrow write(In_{bf}[0])$;
    repeat 4 times
        FC_forward(); //4-FC layer
    if not last block then
        repeat 2 times
            FC_forward(); //backcast branch
        $RES_{bf}[0] \leftarrow RES_{bf}[0] - HP_{bf}$; //Equation 4.2
    repeat 2 times
        FC_forward(); //forecast branch
    $RES_{bf}[1] \leftarrow RES_{bf}[1] + HP_{bf}$; //Equation 4.2

Function NBEATS_block_backward
    $In_{bf}[1] \leftarrow Cvt(RES_{bf}[1])$;
    repeat 2 times
        FC_backward(); //forecast branch
    if not last block then
        $In_{bf}[0] \leftarrow Cvt(-RES_{bf}[0])$; //Equation 4.5
        repeat 2 times
            FC_backward(); //backcast branch
    $In_{bf}[0] \leftarrow Error\ Addition(In_{bf}[0], In_{bf}[1])$;
    repeat 4 times
        FC_backward(); //4-FC layer
    if not first block then
        $RES_{bf}[0] \leftarrow RES_{bf}[0] + HP_{bf}$;
```

The ResADD block does the residual addition. The In$_{bf}$ buffer stores the input values for the FC block, which are activations for forward and errors for backward. In$_{bf}[0]$ and In$_{bf}[1]$ mentioned in algorithm 5 is the different In$_{bf}$ area for backcast and forecast activation/error.

TABLE 4.2: On-chip buffer size and precision of the training accelerator

	Size	Precision
Weight buffer	$L_k \times L_k$	BM$\langle e_w, m_w \rangle$
Activation buffer	$B \times L_k$	BM$\langle e_{act}, m_{act} \rangle$
In buffer	$2 \times B \times K_k$	BM$\langle e_{act}, m_{act} \rangle$/BM$\langle e_e, m_e \rangle$/ BM$\langle e_{in}, m_{in} \rangle$
LP buffer	$B \times L_k$	BM$\langle e_{act}, m_{act} \rangle$/BM$\langle e_e, m_e \rangle$/ BM$\langle e_g, m_g \rangle$
HP buffer	$B \times L_k$	BM$\langle e_h, m_h \rangle$
Gradient buffer	$L_k \times L_k$	BM$\langle e_g, m_g \rangle$
Residual buffer	$B \times (H_b$ $+H_f)$	BM$\langle e_h, m_h \rangle$

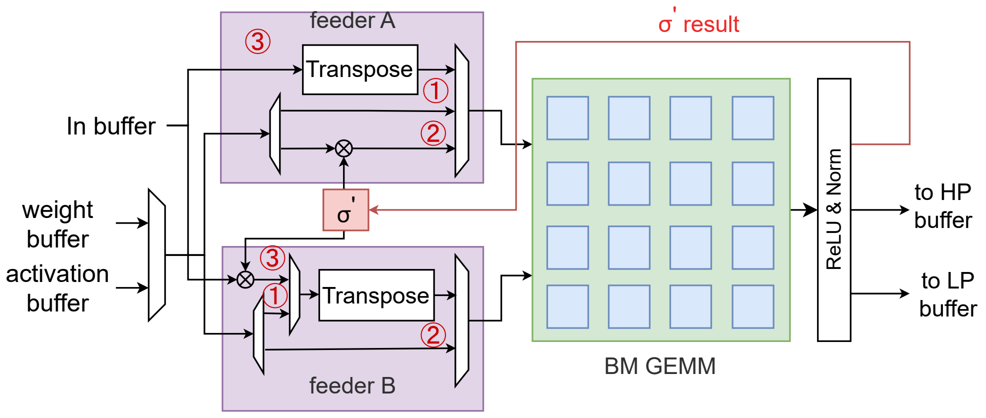


FIGURE 4.13: FC block diagram. FeederA and FeederB have multiple paths for different training phases. ① is the forward propagation; ② is error propagation; ③ is gradient computation. The BM GEMM is illustrated in Figure 5.2, but the Norm block in the GEMM is changed to ReLu&Norm block.

4.4.2 FC Block

The FC block is shown in Figure 4.13. FC block accelerates the matrix multiplication described in Table 4.1, and its computation is illustrated in Figure 4.14. This block feeds parallel data input from the In buffer and the parallel data from the weight or activation buffer to the GEMM kernel. feederA and feederB pre-process this input data by transposing it when necessary for different training phases. The transpose block is a shift register array, which can simultaneously read one tile of the matrix and feed another transposed tile data to the GEMM kernel.

The BM GEMM in FC block is the GEMM kernel illustrated in Figure 5.2. feederA and feederB are configured with different paths for different training computations. Path ① is the forward computation (Equation 4.1). In forward (Figure 4.14(a)), the activation input multiplies with the transposed weight and then calculates the activation output. During forward computation, each activation input is saved to HBM for gradient computation, and the derivation of ReLu is calculated for backward computation.

Path ② is the error propagation (Equation 4.6). In error propagation (Figure 4.14(b)), the error multiplies with the weight and gets the previous layer error. If it is the error of block input $e_{n,0}$, the GEMM kernel outputs the high precision BM data.

path ③ is the gradient computation (Equation 4.7). In weight gradient computation (Figure 4.14(c)), the activation is read from HBM and then transposed and multiplied with the transposed error. The output of the GEMM kernel is the transposed gradient. The output gradient is transposed so that each parallel data in the tile can compute the weight update directly with the weight without transposition, but the tile fetch order is different.

The ReLu&Norm block fuses the Norm block in Figure 5.2 and the ReLu block. The differential value of ReLu σ' is calculated in the forward pass and used in the backward. The data output has two modes: high precision (BM$\langle e_h, m_h \rangle$) connected to the HP buffer, and low-precision connected to the LP buffer.

4.5 Results

4.5.1 Experimental Setting

We evaluated the performance of our accelerator on a Xilinx Alveo U50 board. The design is written in Vitis HLS 2021.2 with a target frequency of 200 MHz. The BM computations are implemented using the HLS ap_int or ap_uint data types as primitives.

To evaluate accuracy and performance, we used the same model architecture and parameters as the original N-BEATS paper [130]. The model used M_{blk}=30 N-BEATS blocks, with the

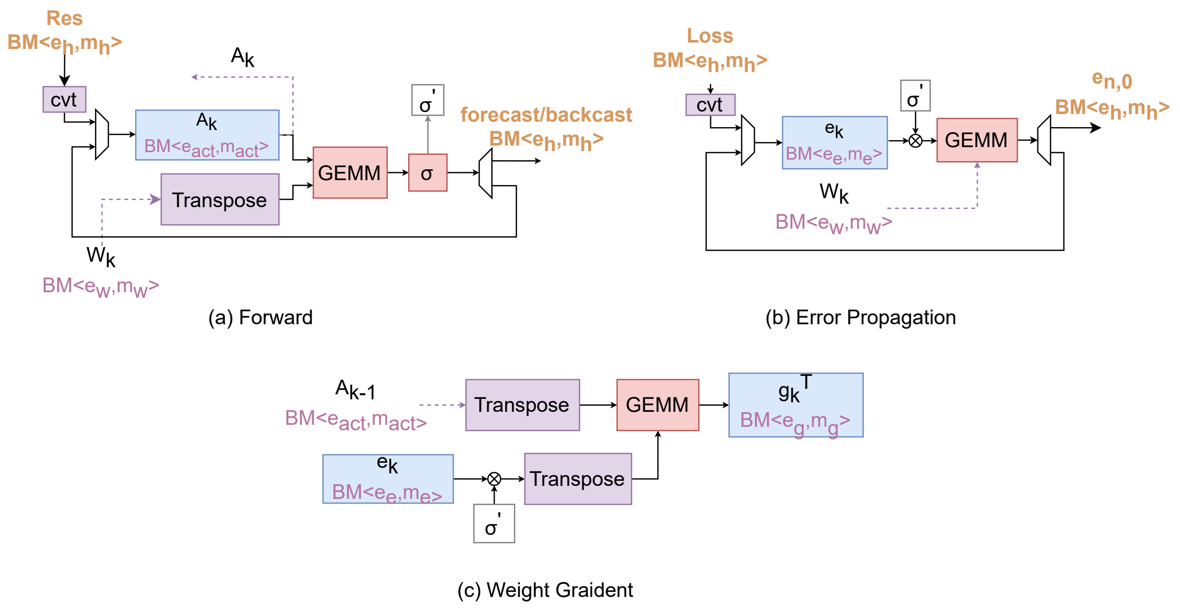


FIGURE 4.14: Illustration of the forward and backward computation steps of an FC block. The dashed line represents the data that needs to be read/written between the HBM memory and the on-chip buffer. The blue box represents the data stored in the on-chip buffer; the red box represents the computing blocks.

weight matrix size $L_k = 512$. Input data is arranged as a 2D array of dimension $B \times H_b$ where the batch size is $B = 1024$ and a forecast horizon H_f is 6. The backcast length is $H_b = 2H_f$, and $\theta = H_f + H_b$.

Experiments include training and validation on the M4 benchmark of the M4-Yearly dataset [110]. The M4 dataset was used to verify that the BM number system achieved minimal loss in accuracy. The N-BEATS model itself has been shown to achieve excellent accuracy over a wider set of benchmarks, including the M3, M4, and TOURISM datasets [110]. Our proposed method can also be applied to other problems, and the throughput is not dependent on the dataset.

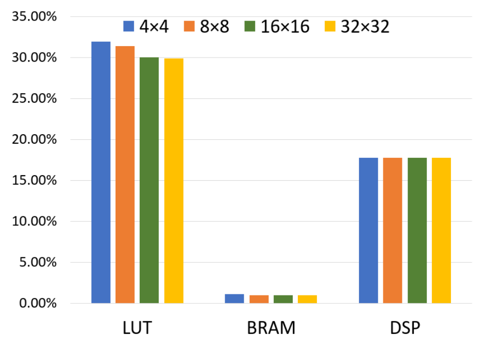
The GPU selected for comparison is an Nvidia GTX 1080. This was chosen because both this and the Alveo U50 were fabricated in a taiwan semiconductor manufacturing company (TSMC) 16 nm FinFET process. On the GPU, FP32 precision was used without quantization, and power was measured using the nvidia-smi tool. Throughput was calculated from the time taken to train the model from scratch using PyTorch.

TABLE 4.3: Precision configurations used in this work

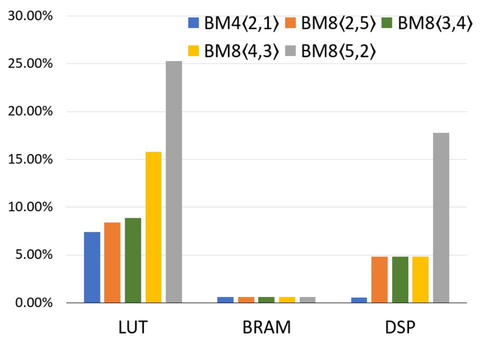
	BM$\langle e_{in}, m_{in}\rangle$	BM$\langle e_w, m_w\rangle$	BM$\langle e_{act}, m_{act}\rangle$	BM$\langle e_e, m_e\rangle$	BM$\langle e_g, m_g\rangle$	BM$\langle e_h, m_h\rangle$
BM8-uniform	BM$\langle 0, 7\rangle$	BM$\langle 0, 7\rangle$	BM$\langle 0, 7\rangle$	BM$\langle 0, 7\rangle$	BM$\langle 0, 7\rangle$	BM$\langle 0, 15\rangle$
BM4-mixed	BM$\langle 0, 3\rangle$	BM$\langle 2, 1\rangle$	unsigned BM$\langle 0, 4\rangle$	BM$\langle 0, 3\rangle$	BM$\langle 0, 3\rangle$	BM$\langle 0, 15\rangle$
BM4-uniform-1	BM$\langle 0, 3\rangle$	BM$\langle 0, 3\rangle$	BM$\langle 0, 3\rangle$	BM$\langle 0, 3\rangle$	BM$\langle 0, 3\rangle$	BM$\langle 0, 15\rangle$
BM4-uniform-2	BM$\langle 0, 3\rangle$	BM$\langle 0, 3\rangle$	BM$\langle 0, 3\rangle$	BM$\langle 0, 3\rangle$	BM$\langle 0, 3\rangle$	BM$\langle 0, 3\rangle$

4.5.2 BM GEMM Area Exploration

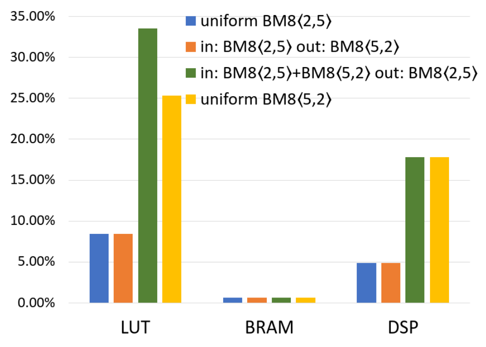
Figure 4.15 shows the area comparison of BM GEMM under different configurations; no DSP packing is applied in the implementation. Figure 4.15(a) illustrates block size configuration impact. In this experiment, the GEMM has 32×32 PEs, and data precision is set to uniform BM8$\langle 2, 5\rangle$. It shows that the impact of block size on area is very small, with a small increase in LUT consumption as the block size becomes smaller.



(a) Block size configuration



(b) $\langle e, m\rangle$ configuration



(c) Mixed precision configuration

FIGURE 4.15: BM GEMM area comparison under different configuration

Figure 4.15(b) shows the impact of the BM parameters on resource utilization for a GEMM with 16×16 PEs, and block size 8×8; the input and output data precision in each $\langle e, m\rangle$

configuration are the same. In BM8 number system, the hardware resources required for BM8$\langle 2, 5\rangle$ and BM8$\langle 3, 4\rangle$ are similar. BM8$\langle 4, 3\rangle$ has similar DSP and BRAM usage, but LUT consumption is doubled, and BM8$\langle 5, 2\rangle$ has approximately triple the LUT and quadruple the DSP consumption than BM8$\langle 2, 5\rangle$. Thus BM8$\langle 2, 5\rangle$ and BM8$\langle 3, 4\rangle$ are preferred from an area perspective over BM8$\langle 4, 3\rangle$ and BM8$\langle 5, 2\rangle$ configurations. Compared to BM8, the MAC units in BM4$\langle 2, 1\rangle$ configuration do not require DSPs, and the LUT usage of the GEMM is smaller than BM8$\langle 2, 5\rangle$.

Figure 4.15(c) shows the resource impact of the mixed precision configuration. In this design, GEMM kernel has 16×16 PEs, and the size is 8×8. The GEMM kernel for a uniform BM8$\langle 2, 5\rangle$ configuration has the same area as the BM8$\langle 2, 5\rangle$ input and BM8$\langle 5, 2\rangle$ output. The mixed precision design, supporting BM8$\langle 2, 5\rangle$ + BM8$\langle 5, 2\rangle$ inputs has approximately a 30% LUT overhead compared with uniform BM8$\langle 5, 2\rangle$ GEMM. The resource utilization of a configurable GEMM is determined by the largest minifloat exponent word length.

4.5.3 Effect of Rounding Scheme

Figure 4.16 shows a tiny N-BEATS training example written in HLS C to demonstrate the difference in the rounding scheme. The model used M_{blk}=2 N-BEATS blocks, with the weight matrix size $L_k = 8$. The x-axis is the number of epochs, and the y-axis is the MAPE loss. The HLS C-stochastic curve is for stochastic rounding of weight update, while the HLS C-nearest curve uses round to nearest. The figure shows that the HLS C-stochastic curve can converge to the same loss value as PyTorch, while the HLS C-nearest curve does not converge. Here, the PyTorch curve also uses stochastic rounding for weight update computation.

4.5.4 Accuracy and Performance of N-BEATS Training

Three different precision configurations for software and hardware evaluation were tested, as shown in Table 4.3. Table 4.4 shows the N-BEATS training accuracy with different BM configurations, evaluated using sMAPE loss. Firstly, the impact of block size is less significant in the BM8-uniform configuration; its training result is close to FP32. For 4-bit, as the block

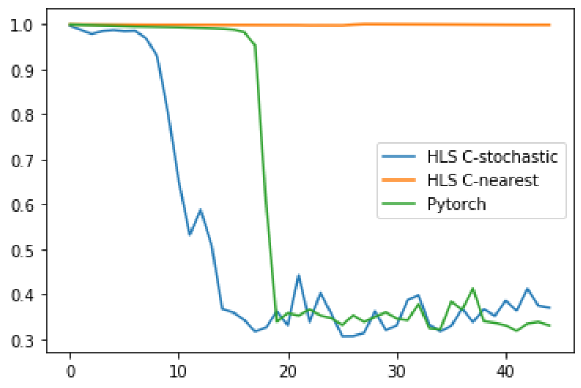


FIGURE 4.16: Effect of different rounding schemes with tiny N-BEATS training example

TABLE 4.4: sMAPE loss for N-BEATS training

Block size	BM8-uniform	BM4-mixed	BM4-uniform-1
16×16	12.95	14.47	17.82
64×64	12.98	14.79	18.45
256×256	12.97	15.00	18.88
whole matrix	12.97	15.69	23.21
FP32		12.93	
BM4-uniform-2		32.01	

TABLE 4.5: area comparison of N-BEATS training accelerator with different precision configuration

	LUT	BRAM	DSP	Freq. (MHz)
BM8-uniform	273K	942	1111	157
	36.23%	80.93%	18.72%	
BM4-mixed	218K	649	1111	183
	28.85%	55.76%	18.72%	
BM4-uniform-1	180K	649	1111	179
	23.80%	55.76%	18.72%	

size is reduced, both BM4-mixed and BM4-uniform-1 configurations achieve smaller sMAPE loss. Secondly, we note that a high precision residual during training is also important; sMAPE loss decreases from 32.01 to 23.21 when a BM$\langle 0, 15 \rangle$ format residual is applied. Thus, mixed precision significantly improves accuracy.

Table 4.5 shows an area comparison of the N-BEATS training accelerator for different precision configurations. The training accelerator employs a GEMM kernel with 32×32

TABLE 4.6: Performance comparison between various training accelerators

	[167]	[172]	[105]	[48]	Ours (no packing)	Ours (packing)	GPU
Device	Stratix 10 MX	VC709	MAX5	ZCU102	Alveo U50	Alveo U50	GTX 1080
number system	FP16	INT16	INT8	BM$\langle 2,5 \rangle$	BM4-mixed	BM4-mixed	FP32
Freq(MHz)	185	-	200	225	183	159	1733
Network	ResNet 20	AlexNet	VGG-like	VGG-like	N-BEATS	N-BEATS	N-BEATS
DSP	1040(26%)	$\approx$ 2880(80%)	6241(91%)	373(15%)	1111(19%)	1103(19%)	-
LUT/ALM	239K(34%)	-	679K	147K(54%)	218K(29%)	498K(66%)	-
BRAM/M20K	2558(37%)	-	1232(29%)	1255(69%)	649(56%)	1003(86%)	-
Batch size	1	-	128	1	1024	1080	1024
Tput. (Gops)	180	1022 (per FPGA)	1417	209	299.03	779.12	**2991**
Power(W)	20	32	13.5	7.7	20.86	18.38	220
Gops/W	9	31.97	**105**	27.1	14.34	42.4	13.6
Gops/DSP	0.17	0.36	0.34	0.56	0.27	**0.71**	-

PEs, without DSP packing. The model training parameters are identical to those in Table 4.4. The same design, configured with different number systems, was used to generate Table 4.5. For all designs, the target frequency was set to 200 MHz, while the frequencies reported in Table 4.5 and Table 4.6 reflect the actual operating frequency. The discrepancy between the actual and target frequencies is caused by the critical path within the PE array. Each input of matrices A and B is connected to every row or column of the PEs, resulting in significant fan-out. This issue is constrained by the HLS tool, as HLS cannot implement a pipelined systolic input architecture.

The BM4-uniform-1 configuration accelerator has 17.5% lower LUT consumption than BM4-mixed, and uses 31.1% less BRAM and 20.3% LUT than the BM8-uniform configuration. An advantage of a smaller word length is that on-chip memory size requirements are reduced. Our hardware resource constraint is on-chip memory, and DSP packing significantly reduces LUT utilization. In the present design, the performance is limited by on-chip memory, so unfortunately, additional computational capability will not improve throughput.

The mixed precision 4-bit accelerator performance with and without packing is given in Table 4.6. It uses a GEMM kernel with 36×24 PEs, with each PE employing a single DSP to implement six, 2×3 significand multipliers. Overall, the GEMM kernel has 72×72 MAC units.

For the packing configuration in Table 4.6, the batch size used was 1080 to fit the tile size, and a block size of 12×12 was chosen. This resulted in an sMAPE loss of 14.5. The accelerator achieves a throughput of 779.12 Gops and system power of 18.38 W from the Vitis analyzer. The implementation with packing has lower power consumption due to the lower clock frequency.

The GPU we chose for comparison is an Nvidia GTX 1080, as both it and the AMD Alveo U50 were fabricated using a TSMC 16 nm FinFET process. The GTX 1080 GPU implementation has a throughput of 2991 Gops with FP32 precision for training, and power consumption of 220 W measured using the nvidia-smi tool. The FPGA implementation demonstrated 3.12x better Gops/W than the GPU, and better Gops/DSP performance than other FPGA training implementations, albeit for different sizes and types of neural networks.

We note that a direct comparison of the different implementations in Table 4.6 is not particularly meaningful as the underlying neural network architectures are different, and computational demands are dependent on the network topology. The main purpose is to demonstrate that the proposed architecture achieves good utilization of the FPGA resources and can sustain high throughput for training. Regarding the batch size, a small value optimizes latency, whereas a large one is best for throughput. For example, [167] and [48] in Table 4.6 are optimized for latency. This is because "Low-batch training greatly reduces memory requirement and unlocks opportunities for FPGAs" [167]. The N-BEATS paper uses a batch size of 1024 for training, and we used the same number in our implementation. Since matrix multiplication is the dominant computation in N-BEATS training, a larger batch size can increase the utilization of the GEMM kernel and thus improve the system's throughput.

4.6 Summary

In this chapter, we demonstrated the feasibility of training time series forecasting networks using BM arithmetic. Our mixed precision scheme, combined with a high-precision residual, was used to implement N-BEATS at primarily 4-bit precision (Mix BM4) and achieve a sMAPE loss of 14.47, which is similar to an FP32 result of 12.93 on the M4-Yearly dataset.

We proposed a BM arithmetic implementation scheme that incorporates a novel BM GEMM architecture and 4-bit DSP-packed PE design. Our Mix BM4 implementation uses 31.1% less BRAM and 20.3% less LUTs compared to BM8, and with DSP packing, the accelerator achieves a throughput of 779 Gops, DSP utilization 0.7 Gops/DSP, and power efficiency 42.4 Gops/W.

CHAPTER 5

Delay Update: efficient rescaling for training

5.1 Introduction

Chapters 3 and 4 demonstrate the effectiveness of block arithmetic in accelerating neural network inference and training. Reducing precision decreases memory bandwidth requirements, reduces memory footprint, and minimizes the size of the MAC unit. Recently, block arithmetic has been adopted by major technology companies, including Microsoft, AMD, Intel, Meta, NVIDIA, and Qualcomm, in the form of the MX format [129, 144]. A detailed definition of the MX format is provided in Chapter 2.

In the MX format, rescaling is required not only for the initial conversion of input values but also for the conversion of intermediate values, such as inner products in GEMM and convolution operations, as illustrated in Figure 5.1(a). The intermediate value Z is a high-precision accumulator, typically represented in formats such as FP32, FP64, or long word-length integers to prevent overflow during inner-product computation. We use the term *rescaling*, along with the block scale calculation function $\mathcal{R}_X(\cdot)$ and the element conversion function $\mathcal{R}_P(\cdot)$, to describe this conversion process:

$$X = \mathcal{R}_X(Z) \tag{5.1}$$

$$P = \mathcal{R}_P(Z). \tag{5.2}$$

P represents the low-precision MX data after rescaling, and X denotes the block scale. A common approach is to use the maximum absolute value within the block to determine the block scale, a method commonly referred to as maximum calibration [26, 40, 179, 186, 194, 199, 200].

However, the maximum value can only be determined after all elements in the block are available. Consequently, the data dependency between computing the block scale and performing element conversion introduces additional latency and requires extra on-chip buffer capacity during rescaling. This dependency significantly undermines the performance benefits of block arithmetic. An example using global blocks is shown in Figure 5.1(b). In this case, the buffer in Figure 5.1(b) must store the partial sum V, and the rescaling latency cannot be effectively pipelined.

It is critical to eliminate this dependency to reduce buffering requirements and decrease the initiation interval of a pipelined implementation. The delayed scaling method addresses this challenge by estimating the block scale based on historical block scales. In this chapter, we analyze the GEMM performance of the maximum calibration method compared to the delayed scaling method. We introduce a new delayed scaling technique, called delay update, which removes this critical dependency while remaining simple to implement and adaptable to training with local blocks. The contributions of this chapter are as follows:

- We analyze the impact of the delayed scaling method on GEMM performance and demonstrate its hardware benefits, achieving training latency reductions of up to 40%.
- We propose a delay update scheme for training. This method estimates the block scale using the maximum absolute value from the last minibatch iteration and applies a weighted moving average filter to constrain rapid fluctuations in block scale values.
- Empirical results show that the delay update scheme achieves accuracy comparable to the maximum calibration method.

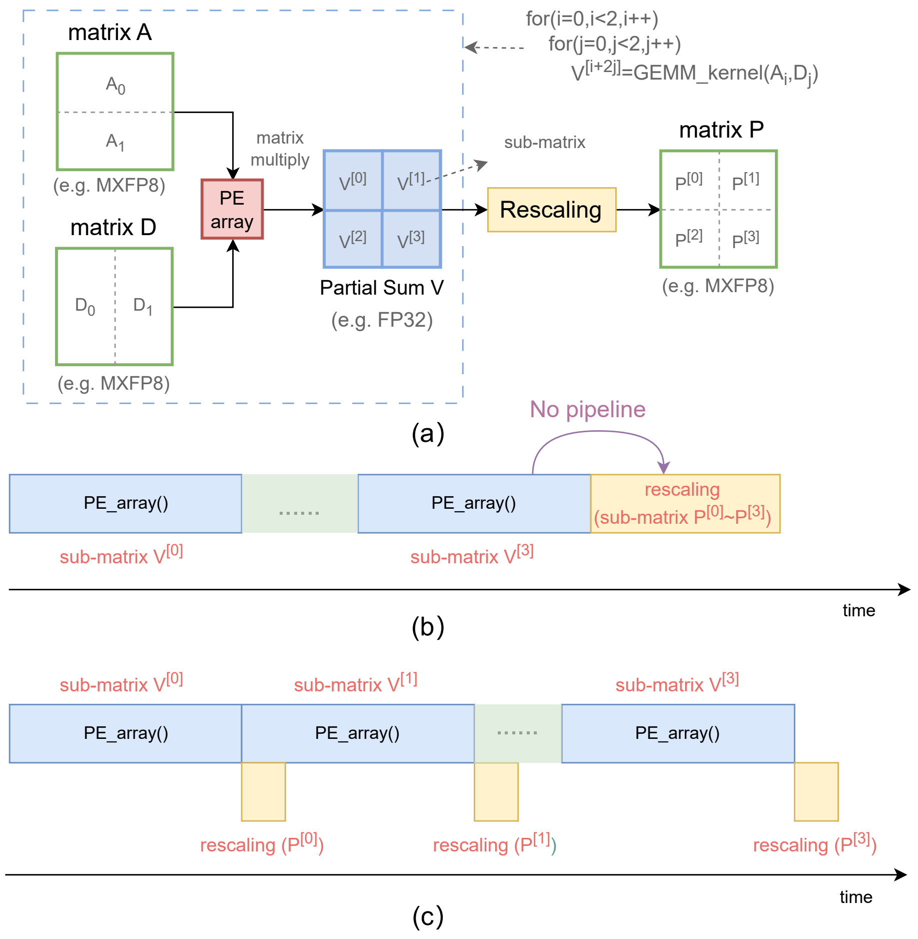


FIGURE 5.1: Rescaling computation. (a) Example of rescaling in MX-format GEMM computation. Matrices A, D, and P represent the low-precision input and output matrices with global blocks, while matrix V denotes the high-precision partial sum matrix. Rescaling converts the high-precision partial sum into the low-precision output. (b) Example of GEMM using the maximum calibration method. In this approach, rescaling can only begin after the matrix multiplication of all submatrices is completed. (c) Example of GEMM using the delayed scaling method. Each submatrix can be rescaled immediately after its multiplication finishes, allowing matrix multiplication to be pipelined with rescaling.

5.2 Background

5.2.1 Rescaling Methods

Rescaling is typically applied during data format conversion and intermediate value conversion in computations such as GEMM and convolution computations in neural networks. Many rescaling methods have been proposed in recent days:

Maximum Calibration

Rescaling is commonly applied during data format conversion and the conversion of intermediate values in computations such as GEMM and convolution operations in neural networks. Several rescaling methods have been proposed in recent years:

$$X = \mathcal{R}_X(Z) = \lfloor \log_2(\max_{z_u \in Z}(|z_u|)) \rfloor \tag{5.3}$$

Z is the set of elements in the block, and z_u is the high-precision elements (e.g. FP32). Then the elements in the block are scaled according to

$$P = \mathcal{R}_P(Z) = [z_u/2^X]_{u=0}^{b-1} \tag{5.4}$$

and quantized to the MX data type P. However, the computation in Equation 5.4 can only begin after the calculation in Equation 5.3 is completed. The latency of maximum calibration is the sum of the computation latencies for $\mathcal{R}_X(Z)$ and $\mathcal{R}_P(Z)$. An extra buffer is also required to store the vector Z, as illustrated in Figure 5.1(b). Recent approaches—such as entropy calibration [116], percentile calibration [112], and optimal clipping [146]—employ alternative strategies to determine the block scale, rather than relying on the maximum absolute value. While these methods may improve training accuracy compared to maximum calibration, their computational complexity must also be taken into account.

Maximum calibration can be explicitly used in low-precision arithmetic training or inference [26, 40, 179, 186, 194, 199, 200], or implicitly used by the conversion between a high precision FP32 output and low-precision input in the next layer [36, 59].

Delayed Scaling

Delayed scaling, on the other hand, estimates scaling factors based on data distributions from preceding minibatch training iterations [4, 93, 185]:

$$S^{(t)} = \mathcal{R}_X(X^{(t-1)}, X^{(t-2)}, ...) \tag{5.5}$$

t is the index of the minibatch training iteration. $X^{(t)}$ is the block scale value at the tth iteration; $X^{(t)}$ value can be calculated through the maximum calibration method. The elements in the block are scaled with $S^{(t)}$:

$$P = \mathcal{R}_P[z_u/2^{S^{(t)}}]_{u=0}^{b-1} \tag{5.6}$$

Using the estimated scaling factors facilitates the elimination of data dependencies, as illustrated in Figure 5.1(c). These delayed scaling techniques rely on the assumption that statistical properties remain consistent across successive minibatch training iterations. FlexPoint [93] introduces the Autoflex scaling algorithm, which estimates the block scale S by leveraging the maximum block scale and its standard deviation from a sequence of previous minibatch iterations' block scale values $\vec{X}$:

$$S^{(t+1)} \leftarrow \alpha[max(\vec{X}) + \beta \cdot std(\vec{X}) + \gamma X^{(t)}] \tag{5.7}$$

AmirAli et al. [4] apply delayed scaling to activations using exponential moving average block scales, whereas weight quantization continues to follow maximum calibration during QAT. Hisakatsu et al. [185] estimate the block scale using the Q_{max} value from the previous minibatch training iteration, along with a constant offset c:

$$S^{(t+1)} \leftarrow Q_{max} - 7 + c \tag{5.8}$$

Here, Q_{max} denotes the maximum absolute value after filtering out the tensor's top γ percent of values. In addition to block arithmetic, several FP8 training approaches also incorporate delayed scaling [125, 134]. While these methods simplify the rescaling process, the computation involved in estimating the block scale remains complex.

5.2.2 Other Scaling Methods

Some other commonly used scaling methods are applied in low-precision FP training, but these methods differ from the rescaling.

Loss Scaling Loss scaling is employed to preserve small gradient values during FP16 training [115]. It involves applying an empirically determined scaling factor $\alpha > 1$ to the loss, mitigating the risk of underflow in error and gradient values. In the context of backpropagation via the chain rule, the loss $\mathcal{L}$ (Chapter 2) is scaled to prevent potential underflow in the computed gradients:

$$\begin{aligned} \mathcal{L}' &= \alpha \mathcal{L} \\ g' &= \frac{\partial \mathcal{L}'}{\partial w} \end{aligned} \tag{5.9}$$

The gradient g is then rescaled before the weight update computation:

$$g = \frac{g'}{\alpha} \tag{5.10}$$

However, the loss scaling doesn't solve the data dependency problem in mixed precision training [115]. The computation of error and gradient still needs the rescaling in Equation 5.3 and Equation 5.4.

Unit Scaling

Unit scaling is a technique designed to address the limited dynamic range of FP16 and FP8 during training [10, 122]. It is based on the hypothesis of ideal scaling, wherein all tensors—including weights, activations, errors, and gradients—are initialized and scaled to the range [-1, 1) with unit variance. This is achieved by introducing a static scaling factor:

$$Gemm(A, D) = \alpha \cdot A \cdot D \tag{5.11}$$

A and D represent tensors or matrices involved in forward or backward computations. The unit scaling factor α is derived from an analysis of each computational operation within the neural network. It is computed during the weight initialization phase and remains fixed throughout training. The factor α constrains the output of each operation to the range [-1, 1). Given that both A and D are scaled to have unit variance, the resulting matrix multiplication maintains unit variance. As a result, rescaling becomes unnecessary. This approach has demonstrated successful training of BERT and LLM models using both FP8 and FP16 formats.

TABLE 5.1: The summary of various rescaling methods

	Arithmetic	Dependency in rescaling	Dynamic scaling	Complexity	Local block
Loss scaling[115]	FP16	✓	×	Low	×
Unit scaling[10, 122]	FP8/FP16	×	×	Low	×
Maximum calibration[165]	Block arithmetic	✓	✓	High	✓
Delayed scaling[4, 93, 185]	Block arithmetic	×	✓	Middle	×
Delay update	Block arithmetic	×	✓	Low	✓

Various rescaling methods are compared in Table 5.1. The proposed approach falls under the category of delayed scaling. In contrast to existing methods, it simplifies the computation of block scales and is well-suited for training with local blocks.

5.3 GEMM Performance Analysis

Delayed scaling methods decouple the dependency between $\mathcal{R}_X(Z)$ and $\mathcal{R}_P(Z)$, thereby reducing both the latency of GEMM computations and the required buffer size. This section presents a comparative analysis of GEMM design and performance between the maximum calibration approach and delayed scaling techniques. We examine two distinct GEMM design scenarios to evaluate their hardware implications.

The GEMM kernel comprises a PE array for matrix multiplication and associated rescaling blocks. We adopt an integer-based MAC arithmetic implementation, as detailed in Chapter 3 and 4. During computation, tensors are partitioned into matrices and fed into the GEMM kernel. The size of the PE array within the kernel is T^2, where T denotes the tile size, and is constrained by the on-chip resources available on the FPGA. The block scale size is B^2, which depends on the configuration of the MX arithmetic parameters. The relationship between tile size and block size leads to different MX GEMM kernel designs.

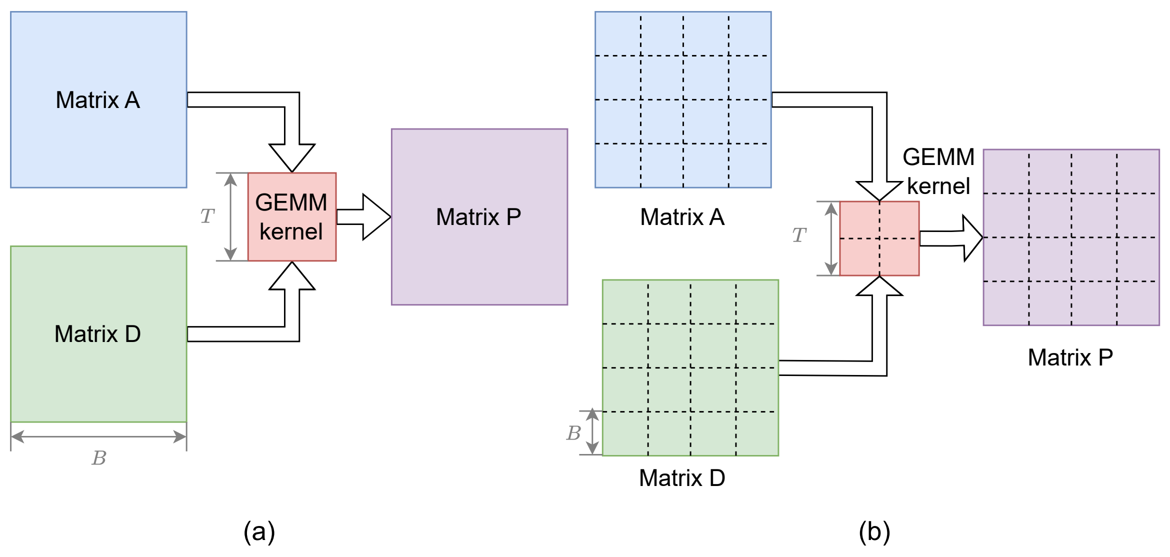


FIGURE 5.2: An example of how the tile size and block size relationship impacts GEMM design. (a) $B \geq T$ case. (b) $T > B$ case.

Figure 5.2(a) illustrates a design scenario where the block size exceeds the tile size. In this case, rescaling must be performed after all tile computations are completed, requiring storage of all intermediate results before rescaling. Conversely, Figure 5.2(b) depicts a design scenario where the tile size is larger than the block size, allowing rescaling to be pipelined during each tile computation.

Let the matrix multiplication be defined as $A \times D = V$, followed by rescaling as $P = \mathcal{R}(V)$. The matrices are defined as follows: $A \in \mathbb{R}^{H_{row} \times K}$, $D \in \mathbb{R}^{K \times H_{col}}$, $V \in \mathbb{R}^{H_{row} \times H_{col}}$, and $P \in \mathbb{R}^{H_{row} \times H_{col}}$. Given a tile size of T^2, the total number of tiles N required for matrix multiplication is computed as:

$$N = \lceil \frac{H_{row} \times H_{col}}{T \times T} \rceil \tag{5.12}$$

Unlike z_u in Equation 5.4 or Equation 5.6, v_u in V represents the output of the PE array and is stored as a long integer accumulator associated with the block scale $X_V^{'}$. Assume that matrices A, D, and P have global block scales denoted by X_A, X_D, and X_P, respectively. Let X_V represent the adjusted block scale $X_V^{'}$ after applying $\mathcal{R}_X(V)$. Under these assumptions, Equation 5.1 can be reformulated as follows:

$$X_V = \mathcal{R}_X(V) = X_V^{'} + \delta \tag{5.13}$$

where

$$\delta = max(\lfloor \log_2(\max_{v_u \in V}(|v_u|)) \rfloor - E_p, 0) \tag{5.14}$$

δ adjusts the block scale of V if the maximum absolute value can cause overflow during element conversion $\mathcal{R}_P(V)$, E_p is the boundary value. Similarly, Equation 5.2 can be rewritten as:

$$P = \mathcal{R}_P(V) = [v_u \gg \delta]_{u=0}^{u=b} \tag{5.15}$$

During the rescaling computation, we assume that $\mathcal{R}_X(V)$ and $\mathcal{R}_P(V)$ each utilize Q parallel computation units.

5.3.1 $B \geq T$ Case

This scenario arises in certain works that employ global block scaling [146, 199] (see Chapter 2). The matrix multiplication under maximum calibration is illustrated in algorithm 6, where $V^{[i,j]}$ denotes the $[i, j]$-th tile of matrix V.

Algorithm 6: Matrix multiplication of $B \geq T$ case using Maximum calibration

```
for $i \leftarrow 0$ to $H_{row}$ / $T$ do
    for $j \leftarrow 0$ to $H_{col}$ / $T$ do
        $V^{[i,j]} \leftarrow$ PE_array($A, D$); // Matrix multiplication for tile
        $v_m \leftarrow max(V^{[i,j]})$; // Find Max scalar
        $v_{max} \leftarrow max(v_{max}, v_m)$;
$X_V \leftarrow \mathcal{R}_X(V)$; // Equation 5.13
$P \leftarrow \mathcal{R}_P(V)$; // Equation 5.15
```

Figure 5.3(a) illustrates the GEMM kernel design, while Figure 5.4(a) presents its latency under the maximum calibration method. In this configuration, all tiles within a block must be computed before initiating the rescaling process. Assuming that the matrix multiplication latency for each tile is K clock cycles and that each MAC operation requires one cycle, the

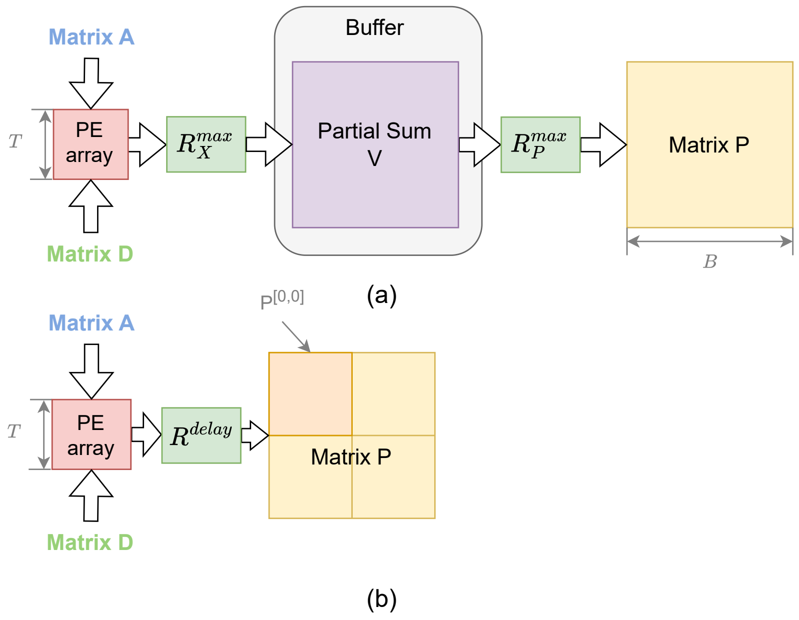


FIGURE 5.3: GEMM kernel design for $B \geq T$ case. (a) The GEMM of maximum calibration. (b) The GEMM of delayed scaling.

overall GEMM computation latency under maximum calibration is given by:

$$l_{max} = N \times K + l_{R_X}^{max} + l_{R_P}^{max} \tag{5.16}$$

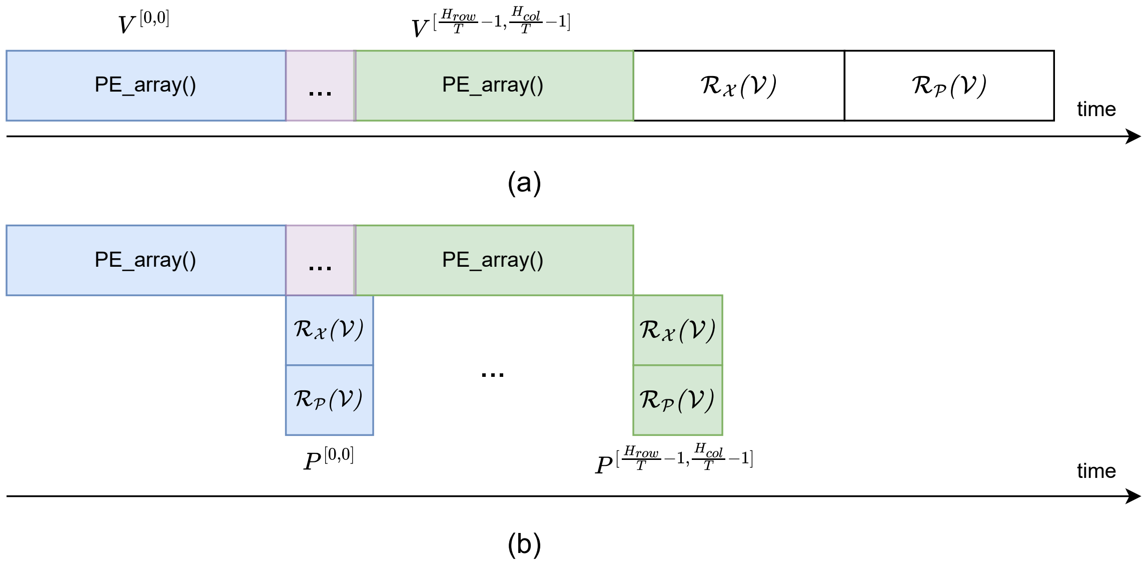
where $l_{R_X}^{max}$ and $l_{R_P}^{max}$ are the latencies for the $\mathcal{R}_X(V)$ and $\mathcal{R}_P(V)$ computations respectively. If we neglect overhead, both latencies can be simplified as $H_{row} \times H_{col}$ clock cycles. If each computation has Q parallel blocks, then the latency is:

$$l_{R_X}^{max} = l_{R_P}^{max} = \lceil \frac{H_{row} \times H_{col}}{Q} \rceil \tag{5.17}$$

Due to the data dependency between rescaling and matrix multiplication, the matrix multiplication latency $N \times K$ cannot be pipelined with the rescaling latency $l_{R_X}^{max} + l_{R_P}^{max}$. As a result, all partial sum results V must be stored in memory during the matrix multiplication phase, as illustrated in Figure 5.3(a):

$$\text{MEM}_{max} = H_{row} \times H_{col} \tag{5.18}$$

Delayed scaling enables rescaling to be performed immediately after completing each tile's matrix multiplication, as illustrated in Figure 5.4(b). Both $\mathcal{R}_X(V)$ and $\mathcal{R}_P(V)$ computations



$V^{[0,0]}$
$V^{[\frac{H_{row}}{T}-1,\frac{H_{col}}{T}-1]}$
PE_array()
...
PE_array()
$\mathcal{R}_{\mathcal{X}}(\mathcal{V})$
$\mathcal{R}_{\mathcal{P}}(\mathcal{V})$
time
(a)
PE_array()
...
PE_array()
$\mathcal{R}_{\mathcal{X}}(\mathcal{V})$
$\mathcal{R}_{\mathcal{P}}(\mathcal{V})$
...
$\mathcal{R}_{\mathcal{X}}(\mathcal{V})$
$\mathcal{R}_{\mathcal{P}}(\mathcal{V})$
$P^{[0,0]}$
$P^{[\frac{H_{row}}{T}-1,\frac{H_{col}}{T}-1]}$
time
(b)

FIGURE 5.4: $B \geq T$ case latency analysis. (a) The matrix multiplication latency of maximum calibration. (b) The matrix multiplication latency of the delayed scaling.

can begin as soon as the corresponding tile multiplication is finished. Consequently, the matrix multiplication latency can be overlapped with the rescaling computation latency, as shown in Figure 5.6(b). The GEMM computation latency for the delayed update method is given by:

$$l_{delay} = N \times max(K, l_R^{delay}) + l_R^{delay} \tag{5.19}$$

where l_R^{delay} is the rescaling latency:

$$l_R^{delay} = \lceil \frac{T^2}{Q} \rceil \tag{5.20}$$

Since rescaling can be performed as the partial sum results are pipelined out from the PE array, there is no need to store intermediate results in memory:

$$\text{MEM}_{delay} = 0 \tag{5.21}$$

5.3.2 $T > B$ Case

For the local block case (Chapter 2) [36, 59, 186], each tile comprises multiple blocks requiring individual rescaling. The matrix multiplication process in this configuration is described in algorithm 7, where $V_{(m,n)}^{[i,j]}$ denotes the (m,n)-th block within the $[i,j]$-th tile of matrix V.

Algorithm 7: Matrix multiplication of $T > B$ case using Maximum calibration

```
for $i \leftarrow 0$ to $H_{row}$ / $T$ do
    for $j \leftarrow 0$ to $H_{col}$ / $T$ do
        $V^{[i,j]} \leftarrow$ PE_array($A, D$); // Matrix multiplication for tile
        for $m \leftarrow 0$ to $T/B$ do
            for $n \leftarrow 0$ to $T/B$ do
                $X_{(m,n)}^{[i,j]} \leftarrow \mathcal{R}_X(V_{(m,n)}^{[i,j]})$;// Equation 5.13
                $P_{(m,n)}^{[i,j]} \leftarrow \mathcal{R}_X(V_{(m,n)}^{[i,j]})$; // Equation 5.15
```

A naive GEMM kernel implementation for maximum calibration is illustrated in Figure 5.5(a). In this design, one tile of the partial sum result is stored in a buffer, and rescaling is performed block by block within the tile. The latency of such a naive implementation can be expressed as:

$$l_{max}^{naive} = N \times (K + l_R^{naive}) \tag{5.22}$$

In the naive implementation, the buffer size required is T^2. The rescaling latency for this approach, denoted as l_R^{naive}, is illustrated in Figure 5.6(a). If we set $Q = T$, then there are $M = \frac{T}{B}$ blocks operating in parallel for rescaling computation. Under these conditions, the rescaling latency l_R^{naive} is given by:

$$l_R^{naive} = l_{R_X}^{naive} + l_{R_P}^{naive} = 2 \times O(\frac{T^2}{T}) = 2M \times B \tag{5.23}$$

And the buffer needed for rescaling is:

$$\text{MEM}_{\text{naive}} = T^2 \tag{5.24}$$

A pipelined implementation is illustrated in Figure 5.5(b), where matrix multiplication is overlapped with the rescaling of adjacent tiles, as well as with intra-tile rescaling between blocks, as shown in Figure 5.6(a). This GEMM kernel design was proposed in [200].

As depicted in Figure 5.6(a), each block requires B clock cycles to complete either $\mathcal{R}_P(V)$ or $\mathcal{R}_X(V)$ computation during each clock cycle. Meanwhile, the total number of pipelined

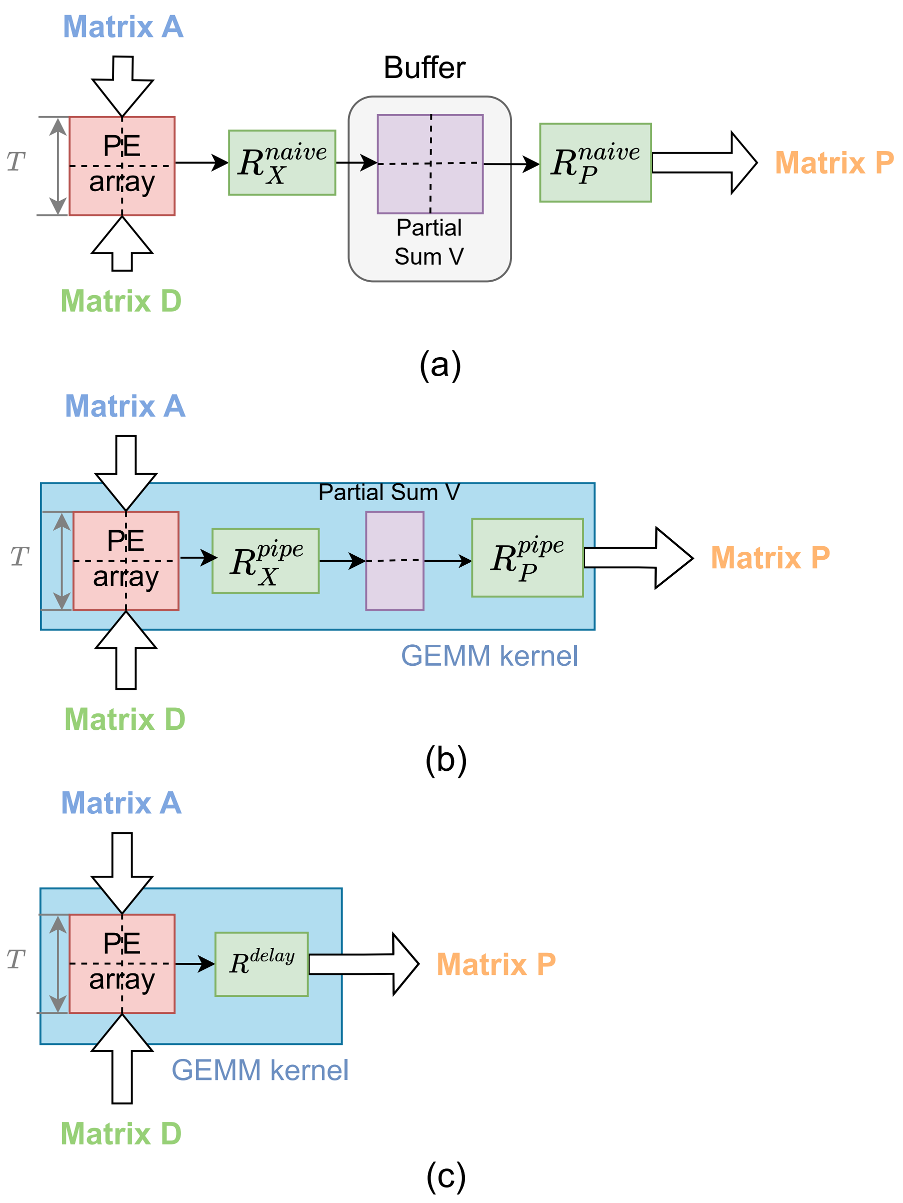


FIGURE 5.5: GEMM kernel design for $T > B$ case. (a) The naive implementation for maximum calibration. (b) The pipelined implementation for maximum calibration. (c) The implementation of delayed scaling.

blocks being rescaled in parallel is $M = \frac{T}{B}$. Therefore, the rescaling latency for a single tile is:

$$l_R^{pipe} = M \times B + B \tag{5.25}$$

And the pipelined GEMM latency for the maximum calibration method is:

$$l_{max}^{pipe} = N \times max(K, l_R^{pipe}) + l_R^{pipe} \tag{5.26}$$

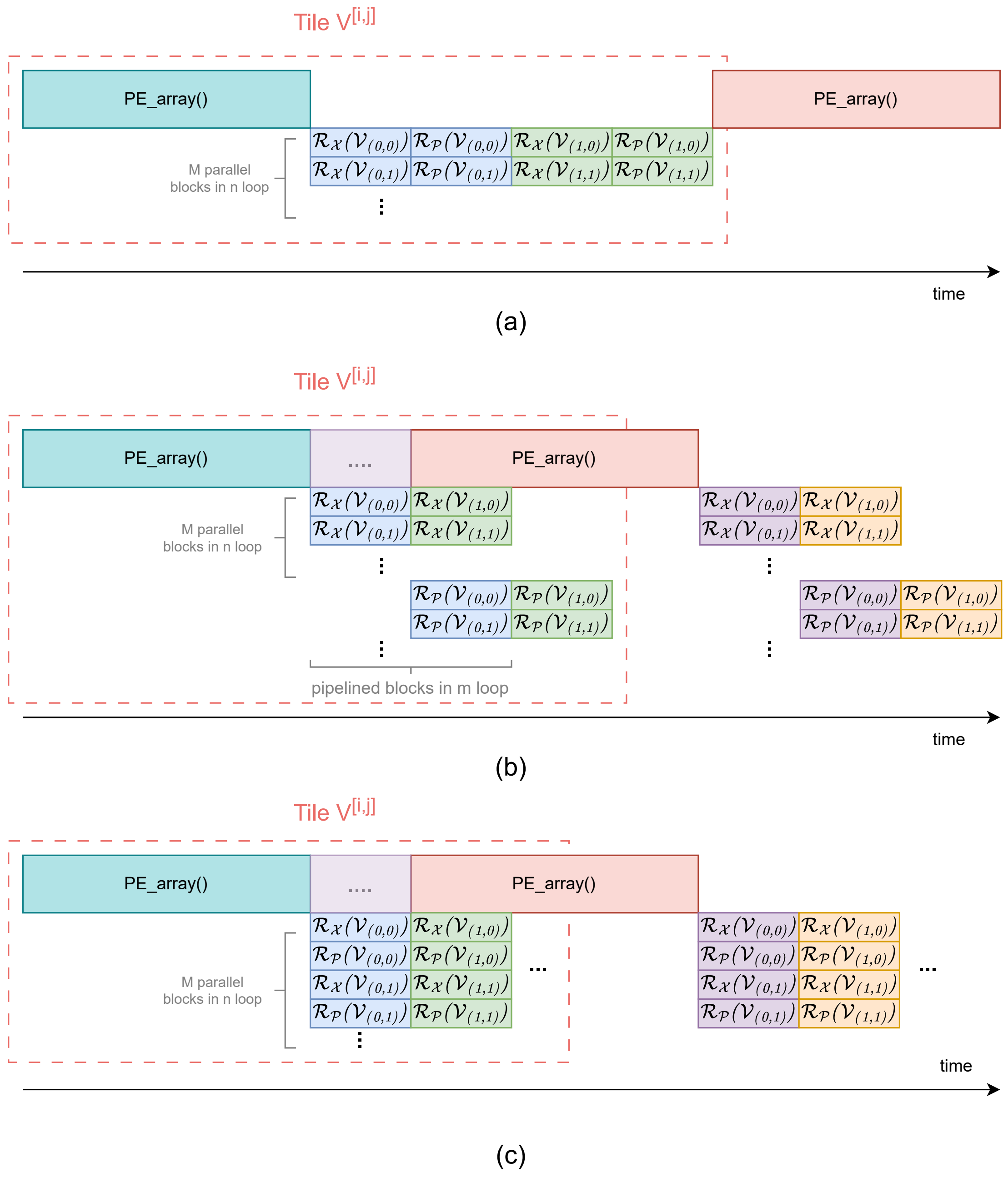

Tile V[i,j]
PE_array()
M parallel blocks in n loop
$\mathcal{R}_\mathcal{X}(\mathcal{V}_{(0,0)})$
$\mathcal{R}_\mathcal{P}(\mathcal{V}_{(0,0)})$
$\mathcal{R}_\mathcal{X}(\mathcal{V}_{(1,0)})$
$\mathcal{R}_\mathcal{P}(\mathcal{V}_{(1,0)})$
$\mathcal{R}_\mathcal{X}(\mathcal{V}_{(0,1)})$
$\mathcal{R}_\mathcal{P}(\mathcal{V}_{(0,1)})$
$\mathcal{R}_\mathcal{X}(\mathcal{V}_{(1,1)})$
$\mathcal{R}_\mathcal{P}(\mathcal{V}_{(1,1)})$
time
(a)
....
pipelined blocks in m loop
(b)
(c)

FIGURE 5.6: $T > B$ case latency analysis. (a) The naive matrix multiplication latency of maximum calibration. (b) The pipelined matrix multiplication latency of maximum calibration. (c) The matrix multiplication latency of the delay update.

During the pipelined computation of $\mathcal{R}_X(V)$ and $\mathcal{R}_P(V)$, the blocks within $V^{[i,j]}$ must be buffered until $\mathcal{R}_X(V)$ determines the block scale value, after which $\mathcal{R}_P(V)$ can proceed. Given that M blocks are processed in parallel during rescaling, the total buffer required for rescaling is:

$$\text{MEM}_{pipe} = M \times B^2 \tag{5.27}$$

The pipelined implementation offers significantly lower latency compared to the naive approach. Furthermore, if the latencies of $\mathcal{R}_X(V)$ and $\mathcal{R}_P(V)$ differ, a ping-pong buffer is required to maintain throughput. In this case, the buffer size becomes $\text{MEM}_{pipe} = 2M \times B^2$, as proposed in [200].

A complex pipelined design can be avoided by adopting the delayed scaling method. As illustrated in Figure 5.6(b), M parallel blocks can simultaneously perform both $\mathcal{R}_X(V)$ and $\mathcal{R}_P(V)$ computations. Consequently, the rescaling latency for the delayed update method is:

$$l_R^{delay} = M \times B \tag{5.28}$$

and its GEMM latency is:

$$l_{delay} = N \times max(K, l_R^{delay}) + l_R^{delay} \tag{5.29}$$

As illustrated in Figure 5.6(c), rescaling computation can also be performed as the partial sum V is pipelined out from the PE array, eliminating the need to store intermediate results:

$$\text{MEM}_{delay} = 0 \tag{5.30}$$

A comparison of the two cases reveals the following:

- In the $B \geq T$ case, the delayed scaling method offers significant advantages in reducing both buffer size and rescaling latency.
- In the $T > B$ case, both the pipelined maximum calibration and delayed scaling method implementations achieve lower latency compared to the naive approach.

However, the delayed scaling method further benefits from reduced memory requirements and a simplified GEMM kernel design.

5.4 Delay Update Algorithm

The objective of the delayed scaling method is to eliminate the dependency described in Equation 5.3 and Equation 5.4 by estimating the block scale value based on statistical observations. Figure 5.7 illustrates the evolution of block scale values for N-BEATS during training. It can be observed that the block scales of weights and activations remain relatively stable, whereas those of errors and gradients exhibit strong temporal trends. This observation raises a question: Is it feasible to simplify block scale estimation by utilizing the block scale value from the last minibatch training iteration?

We begin by examining the case of training with global blocks, under the hypothesis that the values of different tensors between two adjacent minibatch training iterations remain similar. Specifically, the change in weights across iterations is typically small, i.e.,

$$X_w^{(t)} \approx X_w^{(t-1)} \tag{5.31}$$

Likewise, given that input data is commonly normalized during pre-processing, we hypothesize:

$$X_x^{(t)} \approx X_x^{(t-1)} \tag{5.32}$$

In contrast, the error and gradient tensors may exhibit greater variability due to differences in loss values across minibatches. Nevertheless, the deviation in their block scale values between iterations can be bounded, provided the training loss decreases steadily over time.

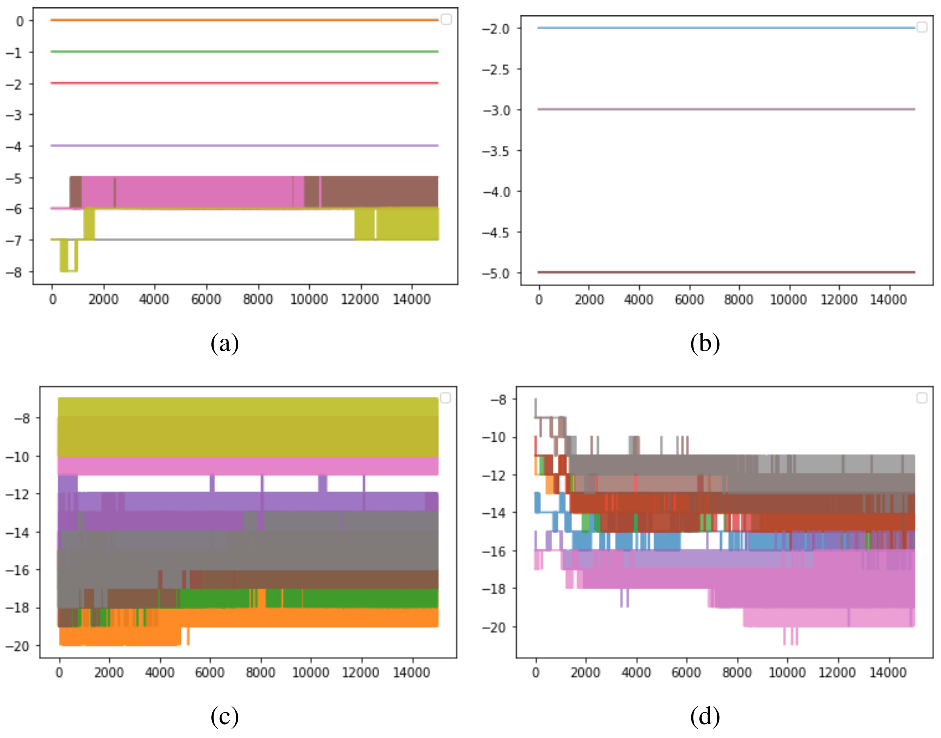


FIGURE 5.7: Block scale values of N-BEATS during training using the maximum calibration method. Each subfigure displays the block scale values of the eight FC layers within the 29th N-BEATS block in different colors. Training is conducted with a global block size. The x-axis indicates the minibatch training iteration, while the y-axis represents the block scale value. (a) Activation. (b) Weight. (c) Error. (d) Gradient.

$$\begin{aligned} \Delta X_e &= |X_e^{(t)} - X_e^{(t-1)}| < c_e \\ \Delta X_g &= |X_g^{(t)} - X_g^{(t-1)}| < c_g \end{aligned} \tag{5.33}$$

ΔX is the block scale bias between two adjacent iterations, and c is the constant. ΔX will introduce errors that we assume can be handled during training as the neural network is robust to the noise [9, 42, 148]. Based on this hypothesis, the delay update method uses the block scale in the last minibatch training iteration for the $\mathcal{R}_P(V)$ calculation:

$$\begin{aligned} S^{(t)} &= X^{(t-1)} = \mathcal{R}_X^{(t-1)}(V) \\ \mathcal{R}_P^{(t)}(V) &= [V_u/2^{S^{(t)}}]_{u=0}^{b-1} \end{aligned} \tag{5.34}$$

Let $S^{(t)}$ denote the estimated block scale value in the t-th iteration, and let $\mathcal{R}_X^{(t-1)}(V)$ represent the block scale result obtained from the $(t-1)$-th minibatch training iteration. The block scale computation follows the same procedure as described in Equation 5.3 and Equation 5.13 under the maximum calibration method.

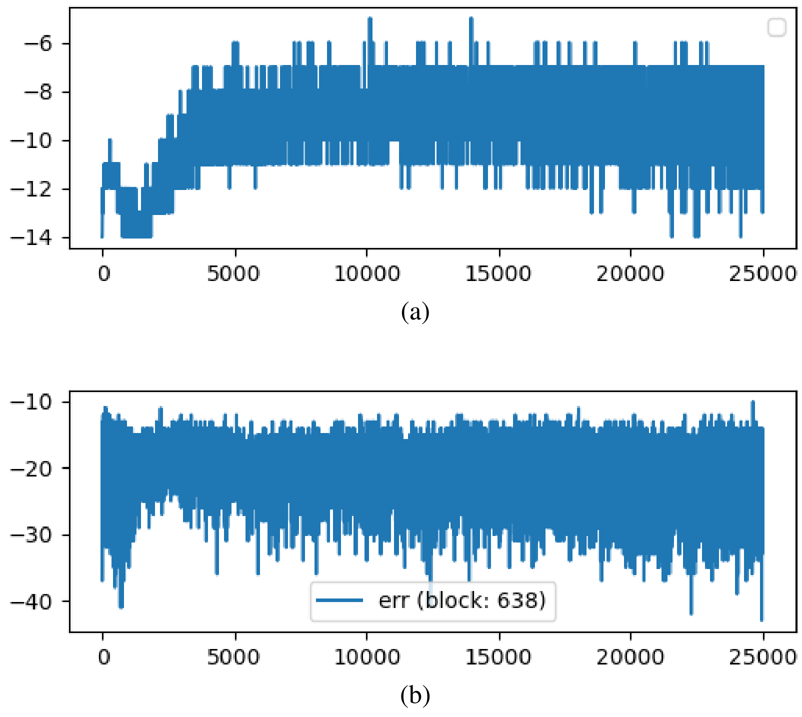


FIGURE 5.8: Block scale values of the error tensor during training using the maximum calibration method. The x-axis represents the minibatch training iteration, and the y-axis indicates the block scale value. (a) Training with global blocks. (b) Training with local blocks (638th block in the first layer).

The method proposed above primarily targets training with global blocks. However, its performance degrades when applied to local blocks, as discussed in section 5.5. Figure 5.8 illustrates the variation of the error block scale X_e during training using the maximum calibration method. It is evident that the range of variation in X_e shown in Figure 5.8(a) is significantly narrower than that in Figure 5.8(b). Moreover, the magnitude of change ΔX_e in (a) is substantially greater than that in (b), contributing to the observed accuracy drop.

Previous delay scaling approaches have predominantly focused on training with global blocks [93, 185]. To mitigate the impact of ΔX_e in local blocks, we introduce a weighted moving average filter:

$$S^{(t)} = \frac{1}{\lambda} log_2 (\sum_{i=1}^{F} \eta_i \cdot 2^{\lambda \cdot X^{(t-i)}}) \tag{5.35}$$

η_i denotes the weight assigned to each historical block scale value, and F represents the window size of the filter. The hyperparameter λ is configured based on specific scenarios. When $\lambda = 1$, the moving average filter reduces to a logarithmic filter, which exhibits heightened sensitivity to changes in ΔX:

$$S^{(t)}_{\lambda=1} = log_2 (\sum_{i=1}^{F} \eta_i \cdot 2^{X^{(t-i)}}) \tag{5.36}$$

As $\lambda \to 0$, the moving average filter approaches a weighted linear filter, which is more responsive to large block scale values X:

$$S^{(t)}_{\lambda\to 0} \approx \sum_{i=1}^{F} \eta_i \cdot X^{(t-i)} \tag{5.37}$$

During training, the delay update method is applied to all tensors. A weighted moving average filter is applied to the error values for local blocks. In the first epoch, the network utilizes weights pretrained using the maximum calibration method. From the second epoch onward, training proceeds with the delay update method.

5.5 Results

We selected three distinct neural network training tasks to evaluate the proposed method. Specifically, N-BEATS is used for time series prediction, ResNet for computer vision, and

Transformer for text classification. All algorithms are implemented in PyTorch and executed on an NVIDIA GeForce RTX 3090 GPU.

A detailed introduction to N-BEATS is provided in Section 4.2.1. The configuration employed consists of 30 N-BEATS blocks, each comprising 8 FC layers with a maximum weight size of 512×512, and residual connections between blocks. The training and validation batch size is set to 1024. All tensors are represented in 8-bit MXINT8 format, while residuals are maintained in FP32. The dataset used for N-BEATS is the M4 benchmark, which includes M4-Yearly, M4-Quarterly, M4-Monthly, and M4-Daily subsets. The validation metric is the sMAPE loss, where a lower value indicates better performance. The optimizer used for weight updates is SGD.

ResNet [62], a widely adopted architecture for image classification, is evaluated using the CIFAR and IMAGENET datasets. Specifically, ResNet-18 is used for CIFAR-10, ResNet-34 for CIFAR-100, and ResNet-50 for IMAGENET. All tensors are represented in 8-bit MXINT8 format, with residuals preserved in FP32. The batch size for training and validation is 128, except for ResNet-50, which uses a batch size of 256 to reduce training latency. The evaluation metric for computer vision tasks is top-1 classification accuracy. The optimizer used is the default SGD with momentum, where the momentum coefficient is set to 0.9.

We employ a two-layer encoder Transformer architecture for text classification with an FC output layer (referred to as Transformer-Tiny). The word embedding dimension is 256, and the hidden layer size in the feed-forward network is $d_{\text{ff}} = 1024$. In the multi-head attention mechanism, the key and value dimensions are set to $d_k = d_v = 32$, with $h = 8$ attention heads and a model dimension of $d_{\text{model}} = 256$. The batch size during training is 20, and the dropout rate is set to 0.5. All tensors are represented in 8-bit MXINT8 format, while residuals are retained in FP32. The optimizer used is Adam, the default choice for Transformer models. The dataset for Transformer-Tiny is the internet movie database (IMDB) Movie Reviews dataset, a binary sentiment classification benchmark comprising 50,000 reviews labeled as either positive or negative.

TABLE 5.2: Training performance comparison between delay update and maximum calibration with global blocks

	Network	Metrics	Dataset	Maximum calibration result	Delay update result
Time series Prediction	N-BEATS	SMAPE loss (↓)	M4-Yearly	14.51	14.69
			M4-Quarterly	13.40	13.66
			M4-Monthly	12.79	13.30
			M4-Daily	4.69	4.65
Computer vision	ResNet-18	Top 1 accuracy (↑)	CIFAR-10	93.73%	93.60%
	ResNet-34		CIFAR-100	73.89%	74.16%
	ResNet-50		IMAGENET	68.12%	68.18%
Text classification	Transformer-Tiny	accuracy (↑)	IMDB	86.41%	86.69%

Table 5.2 presents a performance comparison between the maximum calibration method and the delay update method. In this table, all tensors are trained using global blocks. The results indicate that the delay update method achieves performance comparable to the maximum calibration method. In some instances—such as ResNet-34 on CIFAR-100 and Transformer-Tiny on IMDB—the delay update method even outperforms the maximum calibration approach in terms of accuracy.

Table 5.3 reports the performance of Transformer-Tiny trained with local blocks using the delay update method. The results show that applying local blocks to weights and gradients has minimal impact on accuracy. However, applying local blocks to activations and errors yields different outcomes. Specifically, when local blocks of size 32×32 are used for activations, the accuracy decreased slightly from 86.52% to 85.45%. In contrast, applying local blocks to errors—reducing the block size from 128×128 to 32×32—results decrease in accuracy from 80.05% to 73.35%. Similar trends are observed in ResNet and N-BEATS models.

Table 5.4 demonstrates the impact of the weighted moving average filter on model performance. As the block size decreases, the performance of the maximum calibration method improves. The "w/o" column reports results obtained using the delay update method without the moving average filter. These results show a decline in performance as the block size becomes smaller. For instance, N-BEATS achieves a sMAPE loss comparable to that of the maximum calibration method, but the loss increases noticeably with smaller block sizes.

TABLE 5.3: Local block training of Transformer-Tiny using the delay update method on IMDB with block size configuration

weight/gradient block size	activation block size	error block size	accuracy
global	global	global	86.69%
32 × 32	global	global	86.52%
	128 × 128	128 × 128	80.05%
	64 × 64	64 × 64	77.06%
	32 × 32	32 × 32	73.35%
	32 × 32	global	85.45%

We apply the weighted moving average filter to ResNet-34 and Transformer-Tiny. As illustrated in Figure 5.9, the ΔX_e values in ResNet range from -10 to +10, with most values concentrated near zero. Accordingly, we set $\lambda = 1$ to employ a logarithmic moving average filter, using a window size of $F = 3$ and uniform weights $\eta_i = 1$. In contrast, Transformer-Tiny exhibits a wider ΔX_e range (from -20 to +20), prompting the use of a linear moving average filter with $\lambda \to 0$ (e.g., 0.001). For this configuration, we set $F = 3$ and use weights $[0.232, 0.301, 0.232]$. The "w" column in Table 5.4 shows that training with the weighted moving average filter enables ResNet-34 and Transformer-Tiny to achieve accuracy levels close to those obtained with the maximum calibration method.

TABLE 5.4: Local block training w/o moving average filter

Network	Block size	Maximum calibration	w/o	w
NBEATS	16 × 16	14.63	14.89	/
	32 × 32	14.64	14.85	/
	64 × 64	14.66	14.79	/
ResNet34	32 × 32	74.45%	70.62%	72.27%
	64 × 64	73.83%	72.46%	73.32%
	128 × 128	73.40%	72.56%	73.80%
Transformer-Tiny	32 × 32	86.60%	73.35%	84.58%
	64 × 64	86.48%	75.72%	85.90%
	128 × 128	86.51%	79.61%	85.89%

Figure 5.10 and Figure 5.11 illustrate the validation accuracy across training epochs. Both figures correspond to the 32 × 32 block size configuration results reported in Table 5.4. The results show that the delay update method, when combined with the weighted moving average filter, achieves a convergence speed comparable to that of the maximum calibration method.

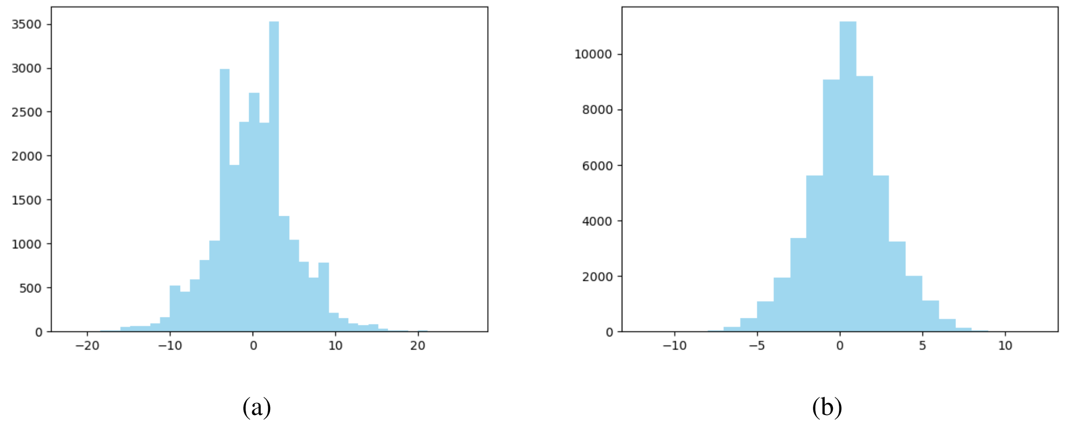


FIGURE 5.9: Histogram of error block scale bias ΔX_e during training. The x-axis represents the ΔX_e values, and the y-axis indicates the frequency count. (a) Histogram for Transformer-Tiny. (b) Histogram for ResNet-34.

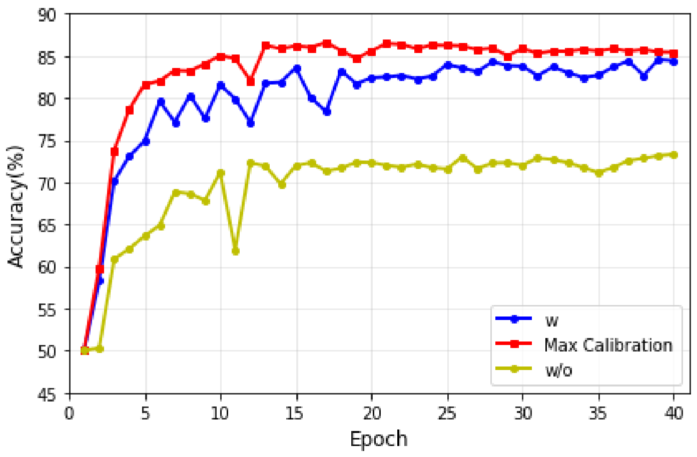


FIGURE 5.10: Accuracy comparison of Transformer-Tiny during training. The block size configuration is 32×32.

We estimate the training latency of Transformer-Tiny and N-BEATS based on the methodology described in section 5.3. The training latencies reported in Figure 5.12 and Figure 5.13 represent the relative latencies between the maximum calibration method and the delay update method. The latency of the maximum calibration method is normalized to 1, and the latency of the delay update method is expressed as a ratio relative to that of the maximum calibration method.

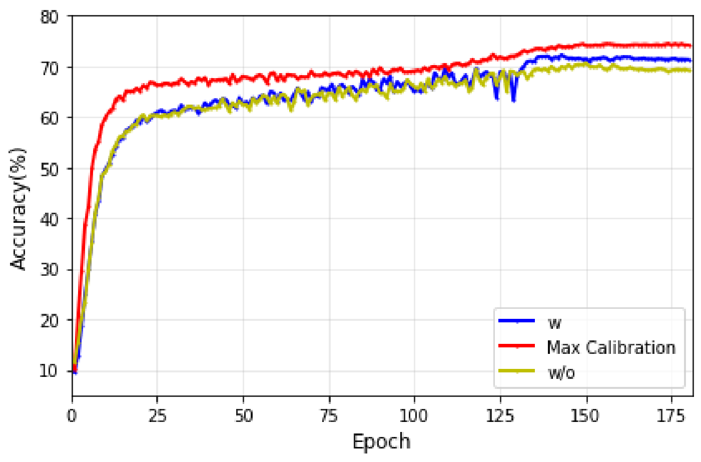


FIGURE 5.11: Accuracy comparison of ResNet during training. The block size configuration is 32×32.

We estimate the relative training latencies of the two methods by measuring the forward and backward propagation latencies for a single minibatch training iteration. Since the latency per iteration remains constant throughout training, the relative latency of the entire training process is equivalent to that of a single iteration. To approximate the single iteration latency, we focus on its total matrix multiplication time, as it constitutes the dominant component.

Figure 5.12 presents the latency comparison under global block configurations. The results include tile sizes of $T = 64$ and $T = 32$, with Q set equal to T. The maximum calibration latencies are normalized to 1, and the delay update latencies are expressed as relative ratios. The results show that the delay update method reduces training latency by approximately 20% to 30%. Furthermore, the latency reduction is more pronounced for $T = 64$ compared to $T = 32$.

Figure 5.13 presents the latency comparison under local block configurations. The local block size used is 16×16, and the results include tile sizes of $T = 64$ and $T = 32$, with Q set equal to T. The maximum calibration implementation shown in Figure 5.13 corresponds to a naive implementation. As discussed in section 5.3, the latency of the pipelined implementation is approximately equal to that of the delay update method. The results indicate that, in the local block case, the delay update method reduces training latency by approximately 10% to 40%. Moreover, the latency reduction is more substantial for $T = 64$ compared to $T = 32$. Overall,

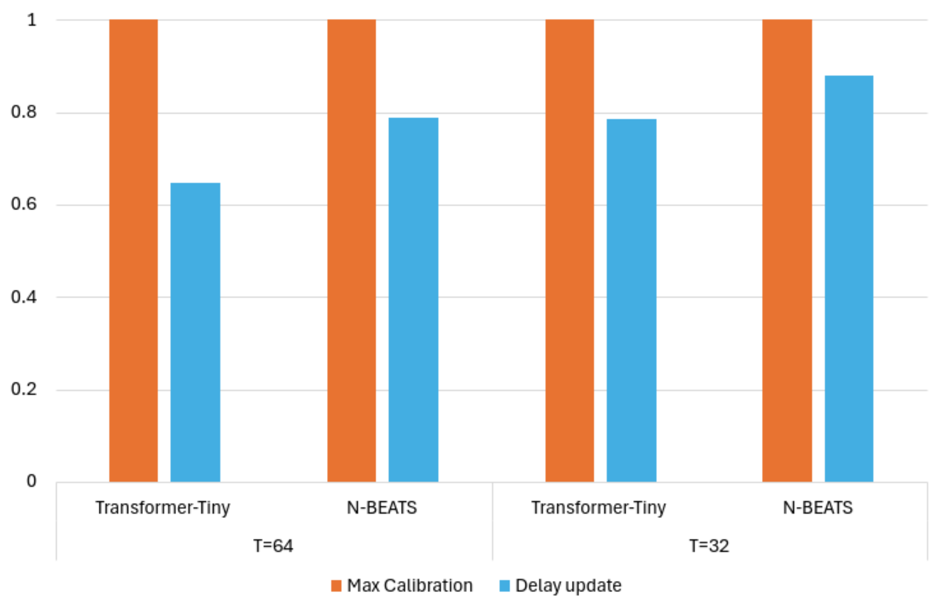


FIGURE 5.12: Normalized training latency comparison using global blocks. The latency of the maximum calibration method is normalized to 1, and the latency of the delay update method is expressed as a relative ratio.

both the pipelined and delay update implementations offer significant improvements over the naive implementation.

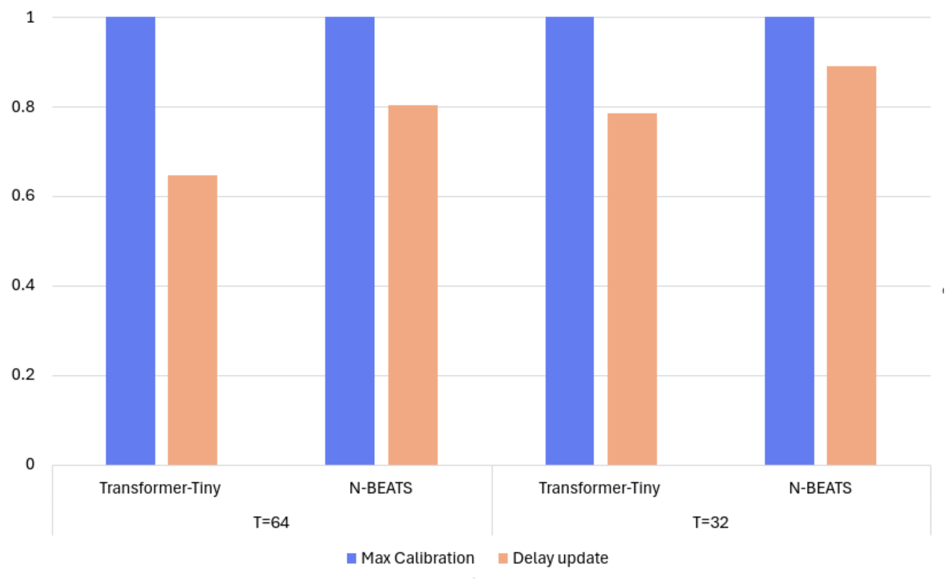


FIGURE 5.13: Normalized training latency comparison using local blocks. The latency of the maximum calibration method is normalized to 1, and the latency of the delay update method is expressed as a relative ratio.

A memory efficiency analysis of LLMs, including BERT-large and GPT-2 medium/large, is illustrated in Figure 5.14 and Figure 5.15. Latency estimation for these models is also based on the methodology described in section 5.3. The results include tile sizes of $T = 64, 128$ and 256, with Q set equal to T. It is observed that employing a larger tile size in conjunction with the delayed scaling method can lead to significant memory savings.

The memory efficiency analysis of block arithmetic GEMM focuses on the total memory required to buffer intermediate results during the rescaling process. Given that on-chip memory capacity and memory bandwidth vary across different FPGA platforms, the latency associated with the maximum calibration method may increase if the available on-chip resources are insufficient to accommodate the rescaling buffer requirements.

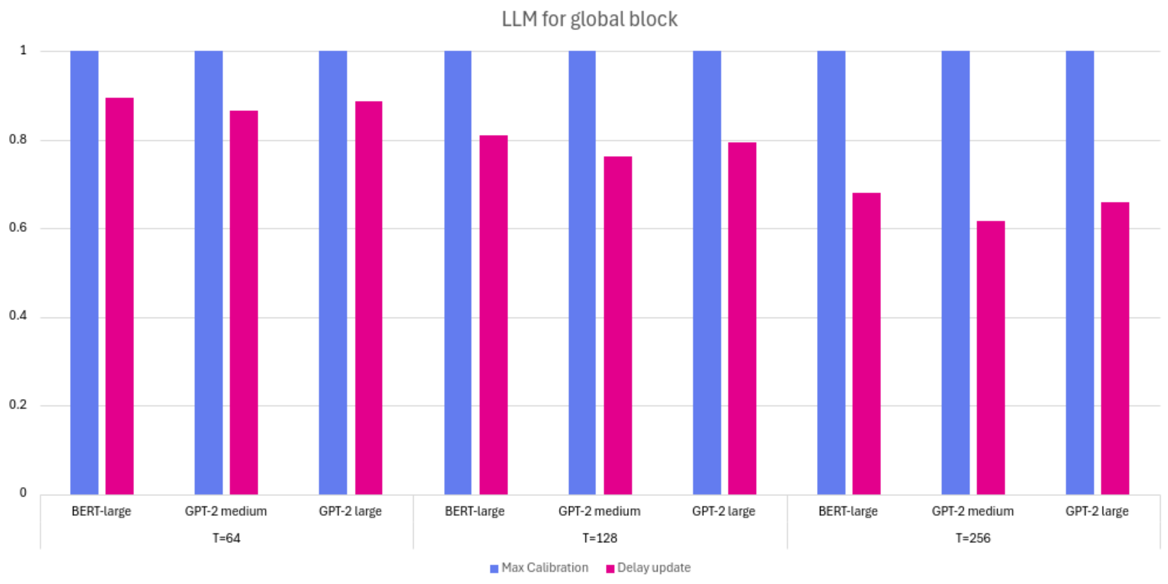


FIGURE 5.14: Normalized training latency comparison using global blocks for LLM, including BERT-large and GPT-2 medium/large.

From Figure 5.14, Figure 5.15, Figure 5.12, and Figure 5.13, we observe that the latency improvement achieved by using global blocks is comparable to that of using local blocks. This phenomenon can be explained by analyzing matrix multiplication as discussed in section 5.3. For global blocks, the latency improvement can be represented as the ratio between the delayed

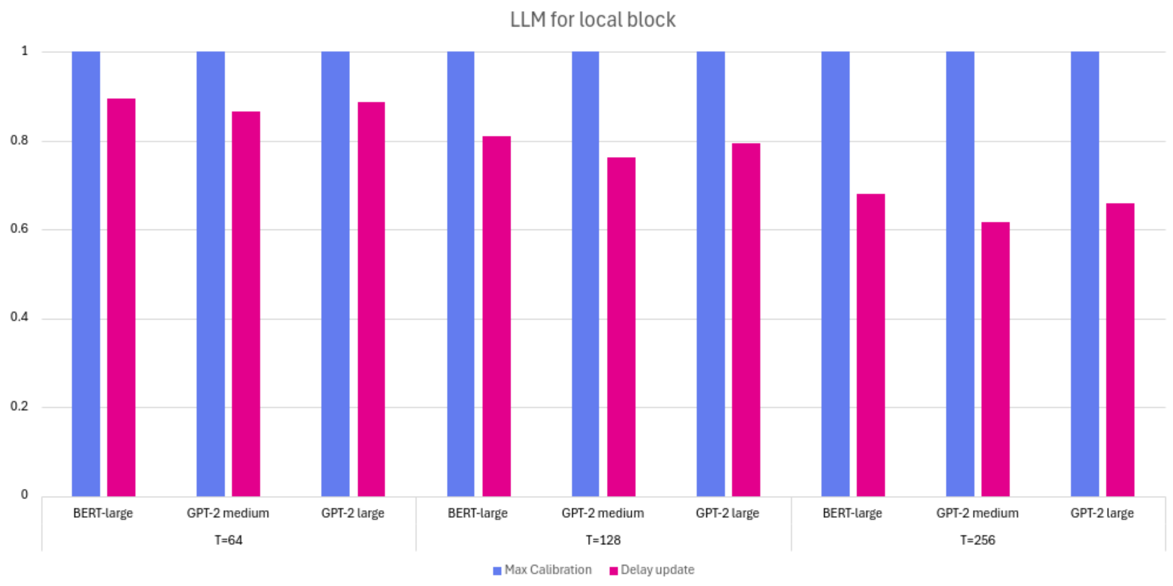


FIGURE 5.15: Normalized training latency comparison using local blocks for LLM, including BERT-large and GPT-2 medium/large.

scaling latency (Equation 5.19) and the maximum calibration latency (Equation 5.16):

$$r_1 = \frac{N \times K + \lceil \frac{T^2}{T} \rceil}{N \times K + 2 \times \lceil \frac{NT^2}{T} \rceil} = \frac{NK + T}{NK + 2NT} \tag{5.38}$$

Similarly, the latency improvement for local blocks is:

$$r_2 = \frac{N \times K + T + B}{N \times K + 2T} = \frac{NK + T + B}{NK + 2NT} \tag{5.39}$$

When $NK + T >> B$, which typically corresponds to a large matrix multiplication, we have $r_1 = r_2$. In Transformer and LLM models, the matrix multiplication size is generally large; therefore, the latency improvement between global blocks and local blocks is nearly identical.

5.6 Summary

This chapter introduces a novel delayed scaling method for block-scale estimation, known as the delay update method. This approach estimates the block scale using the logarithm of the maximum absolute value from the last minibatch training iteration. By doing so, it eliminates the data dependency between maximum value computation and element-wise data conversion.

Compared to existing delayed scaling techniques, the delay update method is both simpler to implement and more adaptable to training scenarios involving local blocks.

We also provide a quantitative analysis demonstrating the superior hardware efficiency of the delay update method over the maximum calibration method when applied to the GEMM kernel. Based on the relationship between GEMM tile size and block size, we observe the following: 1) When the block size is large, the delay update method offers a significant performance advantage over the maximum calibration method. 2) When the tile size increases, the delay update method continues to reduce memory usage and simplifies the GEMM design. Furthermore, latency estimations indicate that the delay update method can reduce training time by approximately 40% for both Transformer-Tiny and N-BEATS models.

CHAPTER 6

Conclusion

This thesis was concerned with the implementation of block arithmetic for both inference and training. The overall contributions of this thesis can be summarized as follows:

At the arithmetic level, this work investigates the implementation of block arithmetic. Chapters 3 and 4 present integer-based block arithmetic for global and local blocks. Using an integer-based accumulator combined with normalization only after the inner-product computation reduces rounding errors during computation. Furthermore, the advantages of integer-based design make it more feasible to implement run-time configurable precision for training.

At the GEMM kernel level, this dissertation examines kernel design under different block arithmetic implementations. Chapter 3 introduces a naive BM GEMM kernel for the global block. Although this implementation incurs significant latency and requires a large buffer for rescaling, the latency and buffer can be partially hidden by overlapping with ReLu computation during inference. Chapter 4 presents a pipelined GEMM kernel for the local block. This design effectively mitigates rescaling latency by pipelining inner-product computation, rescaling across tiles, and exploring advanced pipeline schemes between blocks within the same tile.

For rescaling, Chapter 5 further addresses the challenges associated with block arithmetic and introduces the proposed delayed scaling method called the delay update. This method reduces the latency overhead inherent in block arithmetic. Its primary advantage lies in its simplicity and adaptability to local block configurations. Performance analysis demonstrates that the delayed scaling method significantly reduces rescaling latency and buffer size for the global block while simplifying GEMM kernel design for the local block.

At the accelerator design level, we utilize N-BEATS-based inference and training accelerators to demonstrate the advantages of block arithmetic. Chapter 3 introduces an inference accelerator to highlight the hardware benefits of block arithmetic. Chapter 4 investigates the limits of block arithmetic in both hardware and software contexts. On the software side, we employ mixed precision and smaller block sizes to enhance the training accuracy of N-BEATS, enabling competitive performance even with 4-bit precision. On the hardware side, we optimize the use of DSP resources by proposing a DSP packing scheme for a 4-bit BM MAC unit, which supports six independent multiplications per DSP. These software and hardware techniques are generalizable and applicable to training other neural network architectures.

6.1 Future Outlook

Neural network training acceleration is an emerging research area that remains relatively underexplored due to its inherent complexity. Nevertheless, numerous research opportunities exist for advancing block arithmetic and hardware-efficient neural network training.

6.1.1 Block Arithmetic

Block arithmetic is valuable for extending the dynamic range of low-precision elements within a block while maintaining reduced bit-width per element. Block arithmetic is not restricted to a specific data type; it can be applied to standard formats (e.g., integer and minifloat) and unconventional formats (e.g., binary and ternary). Its applicability also extends beyond neural network training and inference. Block arithmetic can be effectively employed in a variety of domains, including signal processing and image processing. Furthermore, it can also be applied to specialized applications such as LogicNet [178] and LUTNet [169], highlighting its potential as a general-purpose computational method.

However, several open questions remain regarding the block arithmetic, particularly in the context of training. First, there is currently no theoretical framework guiding its use. For instance, it is unclear how to determine the optimal block size for a given application, what the impact of the block is when the elements also have exponent bits, what the relationship is between exponent bit length and block size, and how one-level and two-level block structures differ in practice.

Second, exploring block variants presents a promising research direction. For example, considering sparsity: when the block size is small, the shared exponent and regular block structure could potentially be leveraged to design new structured sparsity algorithms.

Third, rescaling remains an area for further investigation. Chapter 5 introduced the delayed update method for training, which performs well across various neural networks. However, smaller block sizes still impact training accuracy. For larger-scale networks (e.g., LLM) with small block sizes (e.g., block size 32 in MXINT8 or MXFP8), identifying an efficient rescaling strategy remains an open challenge.

6.1.2 Hardware-efficient Training Algorithm

Due to the constraints imposed by forward and backward propagation algorithms, opportunities for hardware-level parallelism in training are limited, as discussed in Chapter 2. Nevertheless, developing hardware-efficient training algorithms remains an interesting and promising research direction.

One of the primary challenges in training is the high memory bandwidth requirement during forward and backward computations. A major contributor to this issue is the need to store activations during the forward pass and retrieve them during the backward pass. Several existing studies have proposed compression techniques that exploit activation sparsity, such as the sparsity introduced by the ReLu function. Designing a general-purpose compression and decompression scheme for activations during forward and backward computation is an interesting area for future exploration.

Beyond memory compression, some research efforts have investigated alternative training algorithms to replace the traditional backpropagation algorithm. For example, works such as [52, 53] demonstrate improvements for specific network architectures; however, these approaches often lack generality. Additionally, large-scale training in data centers introduces significant communication overhead during each minibatch iteration, representing another interesting optimization area.

Bibliography

[1] https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf.

[2] *3rd Generation Intel® Xeon® Scalable Processors, codename Cooper Lake Specification Update — intel.com*. https://www.intel.com/content/www/us/en/content-details/634897/3rd-generation-intel-xeon-scalable-processors-codename-cooper-lake-specification-update.html.

[3] Martín Abadi et al. 'Tensorflow: Large-scale machine learning on heterogeneous distributed systems'. In: *arXiv preprint arXiv:1603.04467* (2016).

[4] AmirAli Abdolrashidi et al. 'Pareto-optimal quantized resnet is mostly 4-bit'. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 3091–3099.

[5] Felix Abecassis et al. 'Pretraining Large Language Models with NVFP4'. In: *arXiv preprint arXiv:2509.25149* (2025).

[6] R. Agrawal and R. Srikant. 'Fast algorithms for mining association rules'. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. Morgan Kaufmann. 1994, pp. 487–499.

[7] Dario Amodei et al. 'Deep speech 2: End-to-end speech recognition in english and mandarin'. In: *International conference on machine learning*. PMLR. 2016, pp. 173–182.

[8] *Arm Updates Its Neoverse Roadmap: New BFloat16, SVE Support — fuse.wikichip.org*. https://fuse.wikichip.org/news/4564/arm-updates-its-neoverse-roadmap-new-bfloat16-sve-support/.

[9] Sanjeev Arora et al. 'Stronger generalization bounds for deep nets via a compression approach'. In: *International conference on machine learning*. PMLR. 2018, pp. 254–263.

[10] Charlie Blake, Douglas Orr and Carlo Luschi. 'Unit scaling: Out-of-the-box low-precision training'. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 2548–2576.

[11] Andrew Boutros, Aman Arora and Vaughn Betz. 'Field-programmable gate array architecture for deep learning: Survey & future directions'. In: *arXiv preprint arXiv:2404.10076* (2024).

[12] Andrew Boutros and Vaughn Betz. 'FPGA architecture: Principles and progression'. In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29.

[13] *Brain floating-point format (bfloat16) - WikiChip — en.wikichip.org*. https://en.wikichip.org/wiki/brain_floating-point_format.

[14] Rina Diane Caballar and Cole Stryker. *A list of large language models — ibm.com*. https://www.ibm.com/think/topics/large-language-models-list.

[15] Léopold Cambier et al. 'Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks'. In: *arXiv preprint arXiv:2001.05674* (2020).

[16] Yu-Hsin Chen et al. 'Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices'. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308.

[17] Wenlin Chen et al. 'Compressing neural networks with the hashing trick'. In: *International conference on machine learning*. PMLR. 2015, pp. 2285–2294.

[18] Xi Chen et al. 'FxpNet: Training a deep convolutional neural network in fixed-point representation'. In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 2494–2501.

[19] Yuxiang Chen et al. 'Oscillation-reduced mxfp4 training for vision transformers'. In: *arXiv preprint arXiv:2502.20853* (2025).

[20] Brian Chmiel et al. ‘FP4 All the Way: Fully Quantized Training of LLMs’. In: *arXiv preprint arXiv:2505.19115* (2025).

[21] Seungkyu Choi, Jaekang Shin and Lee-Sup Kim. ‘A deep neural network training architecture with inference-aware heterogeneous data-type’. In: *IEEE Transactions on Computers* 71.5 (2021), pp. 1216–1229.

[22] ‘Convolutional Neural Network with INT4 Optimization on Xilinx Devices’. In: *Xilinx White Paper* (2020).

[23] Corinna Cortes and Vladimir Vapnik. ‘Support-vector networks’. In: *Machine Learning* 20.3 (1995), pp. 273–297.

[24] Thomas M. Cover and Peter E. Hart. ‘Nearest neighbor pattern classification’. In: *IEEE Transactions on Information Theory* 13.1 (1967), pp. 21–27.

[25] Jiequan Cui et al. ‘Generalized Kullback-Leibler Divergence Loss’. In: *arXiv preprint arXiv:2503.08038* (2025).

[26] Steve Dai et al. ‘Vs-quant: Per-vector scaled quantization for accurate low-precision neural network inference’. In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 873–884.

[27] William J Dally. *High performance hardware for machine learning*. https://media.nips.cc/Conferences NIPS-Tutorial-2015.pdf. 2015. (Visited on 22/07/2012).

[28] Tri Dao. ‘Flashattention-2: Faster attention with better parallelism and work partitioning’. In: *arXiv preprint arXiv:2307.08691* (2023).

[29] Bita Darvish Rouhani et al. ‘Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point’. In: *Advances in neural information processing systems* 33 (2020), pp. 10271–10281.

[30] Bita Darvish Rouhani et al. ‘With Shared Microexponents, A Little Shifting Goes a Long Way’. In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 2023, pp. 1–13.

[31] Johannes de Fine Licht, Grzegorz Kwasniewski and Torsten Hoefler. ‘Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis’. In: *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA’20)*. 2020.

[32] Jia Deng et al. ‘ImageNet: A large-scale hierarchical image database’. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[33] Cem Dilmegani. *The Future of Large Language Models in 2025 — research.aimultiple.com*. https://research.aimultiple.com/future-of-large-language-models.

[34] Alexey Dosovitskiy et al. ‘An image is worth 16x16 words: Transformers for image recognition at scale’. In: *arXiv preprint arXiv:2010.11929* (2020).

[35] *DPUCAHX8H for Convolutional Neural Networks Product Guide (PG367)*. https://docs.amd.com/r/en-US/pg367-dpucahx8h/Resource-Utilization.

[36] Mario Drumond et al. ‘Training dnns with hybrid block floating point’. In: *Advances in Neural Information Processing Systems* 31 (2018).

[37] David Elam and Cesar Lovescu. ‘A block floating point implementation for an N-point FFT on the TMS320C55X DSP’. In: *Texas Instruments Application Report* (2003).

[38] R. David Evans, Lufei Liu and Tor M. Aamodt. ‘JPEG-ACT: Accelerating Deep Learning via Transform-based Lossy Compression’. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 860–873. DOI: 10.1109/ISCA45697.2020.00075.

[39] Maxim Fishman et al. ‘Scaling FP8 training to trillion-token LLMs’. In: *arXiv preprint arXiv:2409.12517* (2024).

[40] Sean Fox et al. ‘A block minifloat representation for training deep neural networks’. In: *International Conference on Learning Representations*. 2020.

[41] Sean Fox et al. ‘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.

[42] Jonathan Frankle and Michael Carbin. ‘The lottery ticket hypothesis: Finding sparse, trainable neural networks’. In: *arXiv preprint arXiv:1803.03635* (2018).

[43] Kazuki Fujii, Taishi Nakamura and Rio Yokota. ‘Balancing Speed and Stability: The Trade-offs of FP8 vs. BF16 Training in LLMs’. In: *arXiv preprint arXiv:2411.08719* (2024).

[44] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2022.

[45] David Goldberg. 'What every computer scientist should know about floating-point arithmetic'. In: *ACM computing surveys (CSUR)* 23.1 (1991), pp. 5–48.

[46] Ashish Gondimalla et al. 'SparTen: A sparse tensor accelerator for convolutional neural networks'. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 151–165.

[47] Ian Goodfellow et al. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.

[48] Chuliang Guo et al. 'BOOST: Block Minifloat-Based On-Device CNN Training Accelerator with Transfer Learning'. In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE. 2023, pp. 1–9.

[49] Kaiyuan Guo et al. 'Angel-eye: A complete design flow for mapping CNN onto embedded FPGA'. In: *IEEE transactions on computer-aided design of integrated circuits and systems* 37.1 (2017), pp. 35–47.

[50] Kaiyuan Guo et al. 'Compressed CNN training with FPGA-based accelerator'. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2019, pp. 189–189.

[51] John L Gustafson and Isaac T Yonemoto. 'Beating floating point at its own game: Posit arithmetic'. In: *Supercomputing frontiers and innovations* 4.2 (2017), pp. 71–86.

[52] Donghyeon Han and Hoi-Jun Yoo. 'Hnpu-v1: An adaptive dnn training processor utilizing stochastic dynamic fixed-point and active bit-precision searching'. In: *On-Chip Training NPU-Algorithm, Architecture and SoC Design*. Springer, 2023, pp. 121–161.

[53] Donghyeon Han and Hoi-Jun Yoo. *On-Chip Training NPU-Algorithm, Architecture and SoC Design*. Springer, 2023.

[54] Donghyeon Han et al. 'A 1.32 TOPS/W energy efficient deep neural network learning processor with direct feedback alignment based heterogeneous core architecture'. In: *2019 Symposium on VLSI Circuits*. IEEE. 2019, pp. C304–C305.

[55] Donghyeon Han et al. ‘A low-power deep neural network online learning processor for real-time object tracking application’. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.5 (2018), pp. 1794–1804.

[56] Donghyeon Han et al. ‘HNPU-V2: A 46.6 FPS DNN Training Processor for Real-World Environmental Adaptation based Robust Object Detection on Mobile Devices.’ In: *HCS*. 2022, pp. 1–18.

[57] Song Han et al. ‘EIE: Efficient inference engine on compressed deep neural network’. In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 243–254.

[58] Song Han 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*. 2017, pp. 75–84.

[59] Simla Burcu Harma et al. ‘Accuracy Boosters: Epoch-Driven Mixed-Mantissa Block Floating Point for DNN Training’. In: *Architecture and System Support for Transformer Models (ASSYST@ ISCA 2023)*. 2023.

[60] Trevor Hastie, Robert Tibshirani and Jerome Friedman. ‘Linear Methods for Classification’. In: *The Elements of Statistical Learning*. Springer, 2009, pp. 101–137.

[61] Kaiming He et al. ‘Deep residual learning for image recognition’. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[62] Kaiming He et al. ‘Deep residual learning for image recognition’. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[63] Kaiming He et al. ‘Mask r-cnn’. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

[64] Kaiming He et al. ‘Masked autoencoders are scalable vision learners’. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 16000–16009.

[65] Kartik Hegde et al. ‘Extensor: An accelerator for sparse tensor algebra’. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 319–333.

[66] Catherine F Higham and Desmond J Higham. ‘Deep learning: An introduction for applied mathematicians’. In: *Siam review* 61.4 (2019), pp. 860–891.

[67] Tin Kam Ho. 'Random decision forests'. In: *Proceedings of the 3rd International Conference on Document Analysis and Recognition*. Vol. 1. IEEE. 1995, pp. 278–282.

[68] Sepp Hochreiter and Jürgen Schmidhuber. 'Long short-term memory'. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[69] https://www.facebook.com/48576411181. *Experts Weigh in on 500B Stargate Project for AI*. https://spectrum.ieee.org/stargate?utm_source=chatgpt.com.

[70] Sitao Huang et al. 'Accelerating sparse deep neural networks on FPGAs'. In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2019, pp. 1–7.

[71] IBM. *IBM Power10 Scale Out Servers Technical Overview*. https://www.redbooks.ibm.com/redpapers/pdfs/redp5675.pdf.

[72] *Ice Lake SP: Overview and Technical Documentation — intel.com*. https://www.intel.com/content/www/us/en/products/platforms/details/ice-lake-sp.html.

[73] IEEE. *754-2019 - IEEE Standard for Floating-Point Arithmetic*. July 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[74] *Improve your model's performance with bfloat16*. https://cloud.google.com/tpu/docs/bfloat16.

[75] *Introducing NVFP4 for Efficient and Accurate Low-Precision Inference*. https://developer.nvidia.com/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/.

[76] *Introduction to Quantization on PyTorch*. https://pytorch.org/blog/introduction-to-quantization-on-pytorch/.

[77] Benoit Jacob et al. 'Quantization and training of neural networks for efficient integer-arithmetic-only inference'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713.

[78] Abhishek Kumar Jain et al. 'Sparse deep neural network acceleration on HBM-enabled FPGA platform'. In: *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2021, pp. 1–7.

[79] Animesh Jain et al. 'Gist: Efficient Data Encoding for Deep Neural Network Training'. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 776–789. DOI: 10.1109/ISCA.2018.00070.

[80] Xiaojun Jia et al. 'Las-at: adversarial training with learnable attack strategy'. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2022, pp. 13398–13408.

[81] Chen Jin. *Graphcore launches C600 PCIe card for AI compute — graphcore.ai*. https://www.graphcore.ai/posts/graphcore-launches-c600-pcie-card-for-ai-compute.

[82] S. C. Johnson. 'Hierarchical clustering schemes'. In: *Psychometrika* 32.3 (1967), pp. 241–254.

[83] I. T. Jolliffe. *Principal Component Analysis*. Springer, 2002.

[84] Dhiraj Kalamkar et al. 'A study of BFLOAT16 for deep learning training'. In: *arXiv preprint arXiv:1905.12322* (2019).

[85] Konstantinos Kanellopoulos et al. 'Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations'. In: *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 2019, pp. 600–614.

[86] Sanghoon Kang et al. '7.4 GANPU: A 135TFLOPS/W Multi-DNN Training Processor for GANs with Speculative Dual-Sparsity Exploitation'. In: *2020 IEEE International Solid-State Circuits Conference - (ISSCC)*. 2020, pp. 140–142. DOI: 10.1109/ISSCC19947.2020.9062989.

[87] Gabriel Katul, Brani Vidakovic and John Albertson. 'Estimating global and local scaling exponents in turbulent flows using discrete wavelet transformations'. In: *Physics of Fluids* 13.1 (2001), pp. 241–250.

[88] Changhyeon Kim et al. 'A 2.1TFLOPS/W Mobile Deep RL Accelerator with Transposable PE Array and Experience Compression'. In: *2019 IEEE International Solid-State Circuits Conference - (ISSCC)*. 2019, pp. 136–138. DOI: 10.1109/ISSCC.2019.8662447.

[89] Heesu Kim et al. ‘GradPIM: A practical processing-in-DRAM architecture for gradient descent’. In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 249–262.

[90] Jiwoo Kim et al. ‘An Investigation of FP8 Across Accelerators for LLM Inference’. In: *arXiv preprint arXiv:2502.01070* (2025).

[91] Sangyeob Kim et al. ‘A 146.52 TOPS/W Deep-Neural-Network Learning Processor with Stochastic Coarse-Fine Pruning and Adaptive Input/Output/Weight Skipping’. In: *2020 IEEE Symposium on VLSI Circuits*. 2020, pp. 1–2. DOI: 10.1109/VLSICircuits18222.2020.9162795.

[92] Urs Köster et al. ‘Flexpoint: An adaptive numerical format for efficient training of deep neural networks’. In: *Advances in neural information processing systems* 30 (2017).

[93] Urs Köster et al. ‘Flexpoint: An adaptive numerical format for efficient training of deep neural networks’. In: *Advances in neural information processing systems* 30 (2017).

[94] Alex Krizhevsky and Geoffrey Hinton. *Learning multiple layers of features from tiny images*. Tech. rep. UTML TR 2009-001. University of Toronto, 2009.

[95] Alex Krizhevsky, Ilya Sutskever and Geoffrey E Hinton. ‘Imagenet classification with deep convolutional neural networks’. In: *Advances in neural information processing systems* 25 (2012).

[96] Ulrich W Kulisch and Willard L Miranker. *Computer arithmetic in theory and practice*. Academic press, 2014.

[97] Andrey Kuzmin et al. ‘FP8 Quantization: The Power of the Exponent’. In: *arXiv preprint arXiv:2208.09225* (2022).

[98] Yann LeCun, Yoshua Bengio and Geoffrey Hinton. ‘Deep learning’. In: *nature* 521.7553 (2015), pp. 436–444.

[99] Yann LeCun et al. ‘Gradient-based learning applied to document recognition’. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[100] Jinmook Lee et al. ‘UNPU: An Energy-Efficient Deep Neural Network Accelerator With Fully Variable Weight Bit Precision’. In: *IEEE Journal of Solid-State Circuits* 54.1 (2019), pp. 173–185. DOI: 10.1109/JSSC.2018.2865489.

[101] Jinsu Lee and Hoi-Jun Yoo. ‘An Overview of Energy-Efficient Hardware Accelerators for On-Device Deep-Neural-Network Training’. In: *IEEE Open Journal of the Solid-State Circuits Society* 1 (2021), pp. 115–128. DOI: 10.1109/OJSSCS.2021.3119554.

[102] Jinsu Lee et al. ‘7.7 LNPU: A 25.3TFLOPS/W Sparse Deep-Neural-Network Learning Processor with Fine-Grained Mixed Precision of FP8-FP16’. In: *2019 IEEE International Solid-State Circuits Conference - (ISSCC)*. 2019, pp. 142–144. DOI: 10.1109/ISSCC.2019.8662302.

[103] Joonhyung Lee et al. ‘Faster Inference of LLMs using FP8 on the Intel Gaudi’. In: *arXiv preprint arXiv:2503.09975* (2025).

[104] Cheng-Hsun Lu, Yi-Chung Wu and Chia-Hsiang Yang. ‘A 2.25 TOPS/W Fully-Integrated Deep CNN Learning Processor with On-Chip Training’. In: *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 2019, pp. 65–68. DOI: 10.1109/A-SSCC47793.2019.9056967.

[105] Cheng Luo et al. ‘Towards efficient deep neural network training by FPGA-based batch-level parallelism’. In: *Journal of Semiconductors* 41.2 (2020), p. 022403.

[106] L. van der Maaten and G. Hinton. ‘Visualizing data using t-SNE’. In: *Journal of Machine Learning Research* 9 (2008), pp. 2579–2605.

[107] *Machine Learning vs Artificial Intelligence: What’s the Difference? | MIT Professional Education — professionalprograms.mit.edu*. https://professionalprograms.mit.edu/blog/technology/machine-learning-vs-artificial-intelligence/.

[108] Raju Machupalli, Masum Hossain and Mrinal Mandal. ‘Review of ASIC accelerators for deep neural network’. In: *Microprocessors and Microsystems* 89 (2022), p. 104441.

[109] J. MacQueen. ‘Some methods for classification and analysis of multivariate observations’. In: 1 (1967), pp. 281–297.

[110] Spyros Makridakis, Evangelos Spiliotis and Vassilios Assimakopoulos. 'The M4 Competition: Results, findings, conclusion and way forward'. In: *International Journal of Forecasting* 34.4 (2018), pp. 802–808. DOI: `10.1016/j.ijforecast.2018`. URL: `https://ideas.repec.org/a/eee/intfor/v34y2018i4p802-808.html`.

[111] Nestor Maslej et al. 'Artificial intelligence index report 2025'. In: *arXiv preprint arXiv:2504.07139* (2025).

[112] Jeffrey L McKinstry et al. 'Discovering low-precision networks close to full-precision networks for efficient inference'. In: *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*. IEEE. 2019, pp. 6–9.

[113] *MGX Fund Management Limited - Wikipedia — en.wikipedia.org*. `https://en.wikipedia.org/wiki/MGX_Fund_Management_Limited`.

[114] Paulius Micikevicius et al. 'Fp8 formats for deep learning'. In: *arXiv preprint arXiv:2209.05433* (2022).

[115] Paulius Micikevicius et al. 'Mixed precision training'. In: *arXiv preprint arXiv:1710.03740* (2017).

[116] Szymon Migacz. '8-bit inference with tensorrt'. In: *GPU technology conference*. Vol. 2. 4. 2017, p. 5.

[117] *Minifloat - Wikipedia — en.wikipedia.org*. `https://en.wikipedia.org/wiki/Minifloat`.

[118] Asit Mishra et al. 'WRPN: Wide reduced-precision networks'. In: *arXiv preprint arXiv:1709.01134* (2017).

[119] Tom M Mitchell and Tom M Mitchell. *Machine learning*. Vol. 1. 9. McGraw-hill New York, 1997.

[120] *Multilayer perceptron - Wikipedia — en.wikipedia.org*. `https://en.wikipedia.org/wiki/Multilayer_perceptron`.

[121] Markus Nagel et al. 'A white paper on neural network quantization'. In: *arXiv preprint arXiv:2106.08295* (2021).

[122] Saaketh Narayan et al. *µnit Scaling: Simple and Scalable FP8 LLM Training*. 2025. arXiv: 2502.05967 [cs.LG]. URL: https://arxiv.org/abs/2502.05967.

[123] Seock-Hwan Noh et al. 'FlexBlock: A flexible DNN training accelerator with multi-mode block floating point support'. In: *IEEE Transactions on Computers* 72.9 (2023), pp. 2522–2535.

[124] Badreddine Noune et al. '8-bit numerical formats for deep neural networks'. In: *arXiv preprint arXiv:2206.02915* (2022).

[125] NVIDIA. *Using FP8 with Transformer Engine*. 2022. URL: https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html.

[126] *NVIDIA A100 Tensor Core GPU Architecture*. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[127] *NVIDIA H100 Tensor Core GPU Architecture*. https://resources.nvidia.com/en-us-tensor-core.

[128] *NVIDIA Tensor Cores: Versatility for HPC and AI — nvidia.com*. https://www.nvidia.com/en-us/data-center/tensor-cores/.

[129] *OCP Microscaling Formats (MX) Specification*. https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf.

[130] Boris N Oreshkin et al. 'N-BEATS: Neural basis expansion analysis for interpretable time series forecasting'. In: *International Conference on Learning Representations*. 2019.

[131] Subhankar Pal et al. 'Outerspace: An outer product based sparse matrix multiplication accelerator'. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 724–736.

[132] Yoonho Park et al. 'GRLC: Grid-based run-length compression for energy-efficient CNN accelerator'. In: *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 2020, pp. 91–96.

[133] Houwen Peng et al. ‘Fp8-lm: Training fp8 large language models’. In: *arXiv preprint arXiv:2310.18313* (2023).

[134] Houwen Peng et al. ‘Fp8-lm: Training fp8 large language models’. In: *arXiv preprint arXiv:2310.18313* (2023).

[135] Sergio P Perez et al. ‘Training and inference of large language models using 8-bit floating point’. In: *arXiv preprint arXiv:2309.17224* (2023).

[136] Billy Perrigo. *What to Know About ’Stargate,’ — time.com*. https://time.com/7209167/stargate-openai-donald-trump/?utm_source=chatgpt.com.

[137] Gabriele Prato, Ella Charlaix and Mehdi Rezagholizadeh. ‘Fully quantized transformer for machine translation’. In: *arXiv preprint arXiv:1910.10485* (2019).

[138] Liu Qi et al. ‘SSiMD: Supporting Six Signed Multiplications in a DSP Block for Low-Precision CNN on FPGAs’. In: *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2023, pp. 161–169.

[139] Jiantao Qiu et al. ‘Going deeper with embedded FPGA platform for convolutional neural network’. In: *Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays*. 2016, pp. 26–35.

[140] J. Ross Quinlan. ‘Induction of decision trees’. In: *Machine Learning* 1.1 (1986), pp. 81–106.

[141] QuinnRadich. *What is a machine learning model? — learn.microsoft.com*. https://learn.microsoft.com/en-us/windows/ai/windows-ml/what-is-a-machine-learning-model.

[142] Joseph Redmon and Ali Farhadi. ‘YOLO9000: better, faster, stronger’. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271.

[143] Shaoqing Ren et al. ‘Faster r-cnn: Towards real-time object detection with region proposal networks’. In: *Advances in neural information processing systems* 28 (2015).

[144] Bita Darvish Rouhani et al. ‘Microscaling Data Formats for Deep Learning’. In: *arXiv preprint arXiv:2310.10537* (2023).

[145] S. T. Roweis and L. K. Saul. ‘Nonlinear dimensionality reduction by locally linear embedding’. In: *Science* 290.5500 (2000), pp. 2323–2326.

[146] Charbel Sakr et al. ‘Optimal clipping and magnitude-aware differentiation for improved quantization-aware training’. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 19123–19138.

[147] B. Schölkopf, A. Smola and K.-R. Müller. ‘Nonlinear component analysis as a kernel eigenvalue problem’. In: *Advances in Neural Information Processing Systems*. Vol. 10. 1998, pp. 583–589.

[148] Michael L Seltzer, Dong Yu and Yongqiang Wang. ‘An investigation of deep neural networks for noise robust speech recognition’. In: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE. 2013, pp. 7398–7402.

[149] *Semi-Supervised Learning, Explained — altexsoft.com*. https://www.altexsoft.com/blog/semi-supervised-learning/.

[150] Jae-sun Seo et al. ‘A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons’. In: *2011 IEEE Custom Integrated Circuits Conference (CICC)*. 2011, pp. 1–4. DOI: 10.1109/CICC.2011.6055293.

[151] Gil Shomron, Tal Horowitz and Uri Weiser. ‘SMT-SA: Simultaneous multithreading in systolic arrays’. In: *IEEE Computer Architecture Letters* 18.2 (2019), pp. 99–102.

[152] Sunil Shukla et al. ‘A Scalable Multi-TeraOPS Core for AI Training and Inference’. In: *IEEE Solid-State Circuits Letters* 1.12 (2018), pp. 217–220. DOI: 10.1109/LSSC.2019.2902738.

[153] Karen Simonyan and Andrew Zisserman. ‘Very deep convolutional networks for large-scale image recognition’. In: *arXiv preprint arXiv:1409.1556* (2014).

[154] *Stochastic gradient descent - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

[155] Jian-Wei Su et al. ‘15.2 A 28nm 64Kb Inference-Training Two-Way Transpose Multibit 6T SRAM Compute-in-Memory Macro for AI Edge Chips’. In: *2020 IEEE International Solid-State Circuits Conference - (ISSCC)*. 2020, pp. 240–242. DOI: 10.1109/ISSCC19947.2020.9062949.

[156] Shiliang Sun et al. ‘A survey of optimization methods from a machine learning perspective’. In: *IEEE transactions on cybernetics* 50.8 (2019), pp. 3668–3681.

[157] Xiao Sun et al. ‘Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks’. In: *Advances in neural information processing systems* 32 (2019).

[158] Xiao Sun et al. ‘Ultra-low precision 4-bit training of deep neural networks’. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1796–1807.

[159] *Supported Data Types 2014; Gaudi Documentation 1.20.1 documentation — docs.habana.ai*. https://docs.habana.ai/en/latest/TPC/TPC_C_Language_Spec/Supported_Data_Types.html.

[160] Mingxing Tan et al. ‘Mnasnet: Platform-aware neural architecture search for mobile’. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 2820–2828.

[161] *Tensor - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/Tensor.

[162] Tesla. *A Guide to Tesla’s Configurable Floating Point Formats & Arithmetic*. https://cdn.motor1.com/pdf-files/535242876-tesla-dojo-technology.pdf.

[163] Albert Tseng, Tao Yu and Youngsuk Park. ‘Training llms with mxfp4’. In: *arXiv preprint arXiv:2502.20586* (2025).

[164] Fengbin Tu et al. ‘Evolver: A deep learning processor with on-device quantization–voltage–frequency tuning’. In: *IEEE Journal of Solid-State Circuits* 56.2 (2020), pp. 658–673.

[165] Vincent Vanhoucke, Andrew Senior, Mark Z Mao et al. ‘Improving the speed of neural networks on CPUs’. In: *Proc. deep learning and unsupervised feature learning NIPS workshop*. Vol. 1. 2011. 2011, p. 4.

[166] Ashish Vaswani et al. ‘Attention is all you need’. In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017, pp. 5998–6008.

[167] Shreyas K Venkataramanaiah et al. ‘FPGA-based low-batch training accelerator for modern CNNs featuring high bandwidth memory’. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020, pp. 1–8.

[168] *Volta (microarchitecture) - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/Volta_(microarchitecture).

[169] Erwei Wang et al. *LUTNet: Rethinking Inference in FPGA Soft Logic*. 2019. arXiv: 1904.00938 [cs.LG]. URL: https://arxiv.org/abs/1904.00938.

[170] Naigang Wang et al. ‘Training deep neural networks with 8-bit floating point numbers’. In: *Advances in neural information processing systems* 31 (2018).

[171] Ruizhe Wang et al. ‘Optimizing Large Language Model Training Using FP4 Quantization’. In: *arXiv preprint arXiv:2501.17116* (2025).

[172] Tianqi Wang et al. ‘FPDeep: Scalable acceleration of CNN training on deeply-pipelined FPGA clusters’. In: *IEEE Transactions on Computers* 69.8 (2020), pp. 1143–1158.

[173] Yu Emma Wang, Gu-Yeon Wei and David Brooks. ‘Benchmarking TPU, GPU, and CPU platforms for deep learning’. In: *arXiv preprint arXiv:1907.10701* (2019).

[174] *What is reinforcement learning? | IBM — ibm.com*. https://www.ibm.com/think/topics/reinforcement-learning.

[175] Chen Wu et al. ‘Low-precision Floating-point Arithmetic for High-performance FPGA-based CNN Acceleration’. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15.1 (2021), pp. 1–21.

[176] Ephrem Wu et al. ‘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.

[177] Ephrem Wu et al. ‘Compute-Efficient Neural-Network Acceleration’. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 191–200. ISBN: 9781450361378. DOI: 10.1145/3289602.3293925. URL: https://doi.org/10.1145/3289602.3293925.

[178] Haiyu Wu et al. *LogicNet: A Logical Consistency Embedded Face Attribute Learning Network*. 2024. arXiv: 2311.11208 [cs.CV]. URL: https://arxiv.org/abs/2311.11208.

[179] Hao Wu et al. 'Integer quantization for deep learning inference: Principles and empirical evaluation'. In: *arXiv preprint arXiv:2004.09602* (2020).

[180] Guangxuan Xiao et al. 'Smoothquant: Accurate and efficient post-training quantization for large language models'. In: *International conference on machine learning*. PMLR. 2023, pp. 38087–38099.

[181] Xilinx. *Convolutional Neural Network with INT4 Optimization on Xilinx Devices (WP521)*. https://docs.xilinx.com/v/u/en-US/wp521-4bit-optimization.

[182] Xilinx. *Deep Learning with INT8 Optimization on Xilinx Devices White Paper (WP486)*. https://docs.xilinx.com/v/u/en-US/wp486-deep-learning-int8.

[183] AMD Xilinx. *Systolic Array*. https://xilinx.github.io/Vitis_Accel_Examples/2019.2/html/systolic_array.html.

[184] Wenhan Xiong et al. 'Simple local attentions remain competitive for long-context tasks'. In: *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2022, pp. 1975–1986.

[185] Hisakatsu Yamaguchi et al. 'Training deep neural networks in 8-bit fixed point with dynamic shared exponent management'. In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021, pp. 1536–1541.

[186] Guandao Yang et al. 'SWALP: Stochastic weight averaging in low precision training'. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 7015–7024.

[187] Shihui Yin and Jae-Sun Seo. 'A 2.6 TOPS/W 16-bit fixed-point convolutional neural network learning processor in 65-nm CMOS'. In: *IEEE Solid-State Circuits Letters* 3 (2019), pp. 13–16.

[188] M. J. Zaki et al. 'New algorithms for fast discovery of association rules'. In: *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*. AAAI Press. 1997, pp. 283–286.

[189] Shulin Zeng et al. ‘Flightllm: Efficient large language model inference with a complete mapping flow on fpgas’. In: *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 2024, pp. 223–234.

[190] Hongyang Zhang et al. ‘Theoretically principled trade-off between robustness and accuracy’. In: *International conference on machine learning*. PMLR. 2019, pp. 7472–7482.

[191] Jintao Zhang et al. ‘Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration’. In: *arXiv preprint arXiv:2410.02367* (2024).

[192] Jintao Zhang et al. ‘Sageattention2 technical report: Accurate 4 bit attention for plug-and-play inference acceleration’. In: *arXiv e-prints* (2024), arXiv–2411.

[193] Sai Qian Zhang, Bradley McDanel and HT Kung. ‘Fast: Dnn training under variable precision block floating point with stochastic rounding’. In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2022, pp. 846–860.

[194] Tianyi Zhang et al. ‘Qpytorch: A low-precision arithmetic simulation framework’. In: *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*. IEEE. 2019, pp. 10–13.

[195] Borui Zhao et al. ‘Decoupled knowledge distillation’. In: *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*. 2022, pp. 11953–11962.

[196] J Zhao et al. ‘Low-precision training in logarithmic number system using multiplicative weight update (2021)’. In: *arXiv preprint arXiv:2106.13914* ().

[197] Jiawei Zhao et al. ‘Lns-madam: Low-precision training in logarithmic number system using multiplicative weight update’. In: *IEEE Transactions on Computers* 71.12 (2022), pp. 3179–3190.

[198] Wenlai Zhao et al. ‘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.

[199] Wenjie Zhou et al. ‘FPGA Implementation of N-BEATS for Time Series Forecasting Using Block Minifloat Arithmetic’. In: *2022 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE. 2022, pp. 546–550.

[200] Wenjie Zhou et al. ‘FPGA-based Block Minifloat Training Accelerator for a Time Series Prediction Network’. In: *ACM Transactions on Reconfigurable Technology and Systems* (2024).