Using Mixed Low-Precision Formats in Multiply-Accumulate (MAC) Units for DNN Training

by

Mariko Tatsumi

B.A.Sc., The University of Toronto, 2019

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia

(Vancouver)

April 2022

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Using Mixed Low-Precision Formats in Multiply-Accumulate (MAC) Units for DNN Training

submitted by Mariko Tatsumi in partial fulfillment of the requirements for

the degree of Master of Applied Science

in Electrical and Computer Engineering

Examining Committee:

Guy Lemieux, Professor, Electrical and Computer Engineering, UBC

Supervisor

Renjie Liao, Assistant Professor, Electrical and Computer Engineering, UBC

Supervisory Committee Member

Abstract

Due to limited size, cost and power, embedded devices do not offer the same computational throughput as graphics processing units (GPUs) for training Deep Neural Networks (DNNs). The most compute-intensive stage of multilayer perceptron (MLP) and convolutional neural network (CNN) training is the general matrix multiply (GEMM) kernel which is executed three times per layer in each iteration: once for forward-propagation and twice for back-propagation. To reduce the number of operations, techniques such as distillation (to reduce model size) and pruning (to introduce sparsity) are commonly applied. This thesis considers another technique, where the computational effort of each operation is reduced using low-precision arithmetic.

While the use of small data types is common in DNN inference, this is not yet common in DNN training. Previous work in the area is somewhat limited, sometimes only considering 16-bit floating-point formats or overlooking implementation details, such as the area and accuracy tradeoffs from exact digital multiplier designs.

This thesis considers the use of mixed-precision operations (MPO) within the GEMM kernel for DNN training. To conduct these investigations, we have implemented a complete DNN training framework for embedded sys-

tems, Archimedes-MPO. Starting with the C++ library TinyDNN, we have abstracted each layer to use custom data types and accelerated the GEMM stage with CUDA and Vitis HLS to produce bit-accurate GPU and FPGA implementations. This framework allows us to exactly measure the impact of various multiplier and accumulator designs on area and accuracy.

Compared to 32-bit floating-point multiplier, as few as 2 mantissa bits attain similar accuracy. Accuracy losses are reduced with adaptive loss scaling and the removal of hardware for rounding and not-a-number (NaN) representations. Removal of subnormals saves area as well, but hurts accuracy, so we propose a custom subnormal encoding as a compromise. For accumulation, 12-bit floating-point and 21-bit fixed-point formats work similarly. Fixed-point accumulation seems to have an area advantage, but the impact of a wider output data size can be costly on downstream logic. While precise results depend upon the model and dataset used, the observed trends and framework can help the design of future GEMM-based hardware accelerators for DNNs.

Lay Summary

Machine learning technology is extensively used among the products used in daily life, such as Google Home and Amazon Alexa, self-driving cars and world-class computer chess program. The basic idea of machine learning is tuning a block-box that initially outputs a random guess until it eventually outputs the correct answer using feedback from applying plenty of example data and expected outputs. However, we also need to shrink the black-box size if we want to use it in a small device, such as the aforementioned home assistant devices; otherwise, the box would not be affordable. In this work, we study how to shrink the black box efficiently by using arithmetic with smaller numbers. The arithmetic core is a multiplier-adder, where the multiplier tends to dominate in size. Our results show the multiplier can be shrunk to about the same size as the adder with negligible loss in output accuracy.

Preface

All chapters of this thesis are written by the author, Mariko Tatsumi. In Chapter 3, CUDA code for the GEMM routine is written by Silviu-loan Filip from the National Institute for Research in Digital Science and Technology (INRIA) at the University of Rennes 1, France. In the same chapter, some of the ancillary modules that are outside of the systolic array in the FPGA GEMM kernel are optimized by Caroline White from the University of British Columbia. Professor Guy Lemieux served as an advisor, and helped to develop the framework, analyze the results, and revise the thesis.

Table of Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

Executive Summary. *This thesis investigates the impact of using different precisions and formats to the core of matrix-multiply hardware accelerators used for DNN training. The evaluation requires a new framework, which is GPU- and FPGA-accelerated, to investigate area and accuracy tradeoffs. The work shows that accurate training of CIFAR-10 [25] on ResNet20 [17] and VGG16 [39] is possible with a multiply-accumulate (MAC) unit containing 8-bit floating-point inputs and 12-bit floating-point outputs. An alternative, mixed-data format configuration with 21-bit fixed-point output, shows promise, but requires system-level optimizations due to a wider output. Key observations are that relaxing IEEE-754 rules for floating-point multiplication to save significant area is possible, and that adaptive loss scaling during training is essential to maintain both multiplier and accumulator accuracy.*

1.1 Motivation

Starting with the breakthrough in image classification accuracy by AlexNet [26], deep learning has been attracting attention for the past decade. In recent years, it has been applied to embedded and edge devices such as various "Internet of Things" (IoT) applications as well as autonomous driving. In

such devices, it may be too costly, too slow, or even impossible to offload the computation to a larger system in the cloud, so computation must be done directly on the device.

Compared to the era before deep learning, models and data sets need considerable memory, computational capacity, and energy use for both training and inference. Since these models take much longer to train, acceleration is essential. The problem of training time and energy use is especially evident for embedded devices, where available resources are limited.

To alleviate these problems, various methods such as distillation (to reduce model size) and pruning (to introduce sparsity) can reduce the size of models and their associated computational complexity. In this thesis, we instead focus on low-precision training where the cost of each calculation is reduced by replacing the de facto use of IEEE single-precision floating-point (FP32) with data types and operations that use much fewer bits. A reduction in data size reduces the hardware needed for a single computation as well as improving memory footprint, effective bandwidth, and power.

Using fewer bits to represent a value is sometimes called quantization. Usually, this term is used when converting from a format with more bits into one with fewer bits, where range and/or precision is lost. In this thesis, we are using low-precision floating-point as well as low-precision fixed-point values, but we avoid the term quantization because operations in FPGA GEMM kernels are done directly in the intended low-precision format.

Previous research on low-precision training has tended to adopt three data formats: floating-point [6, 23, 31, 32, 32, 38, 40, 41, 46, 53, 56], fixed-point [2, 14, 15, 37, 48, 52], or block floating-point [7, 11, 13, 24, 51]. How-

ever, these studies have typically only changed the multiplier input precision, but left accumulation inputs and/or outputs unchanged. Few have explored the effect of using different low-precision formats for the multiply and accumulate operations.

In most previous work, the exact methodology used for implementing low-precision calculations is not described. To do so accurately, one would need to avoid standard accelerated libraries and accurately emulate each low-precision calculation, requiring greater effort to code and leading to a significant slowdown of the training process. One quick technique, used by QPyTorch [54], applies a quantization layer after each composite computational layer like convolution or GEMM. In such a scheme, all intermediate calculations use a higher precision, and only the final results are 'downconverted' to the reduced precision. Such techniques do not properly model precision loss, leading to overly optimistic results with inflated training accuracy.

Consider, for example, the validation accuracy of training LeNet5+MNIST shown in Figure 1.1, where all multiplier inputs are quantized to 5 exponent bits and 1 or 2 mantissa bits (E5M1 and E5M2, respectively). With FP32 accumulation followed by quantization (blue/orange curves), both E5My formats look viable, but with low-precision accumulation in the E5M2 format (green curve) accuracy is slightly degraded. Worse yet, the E5M1 format (red curve) result is catastrophic and never converges. This shows the importance of accurately modeling all stages of low-precision arithmetic.

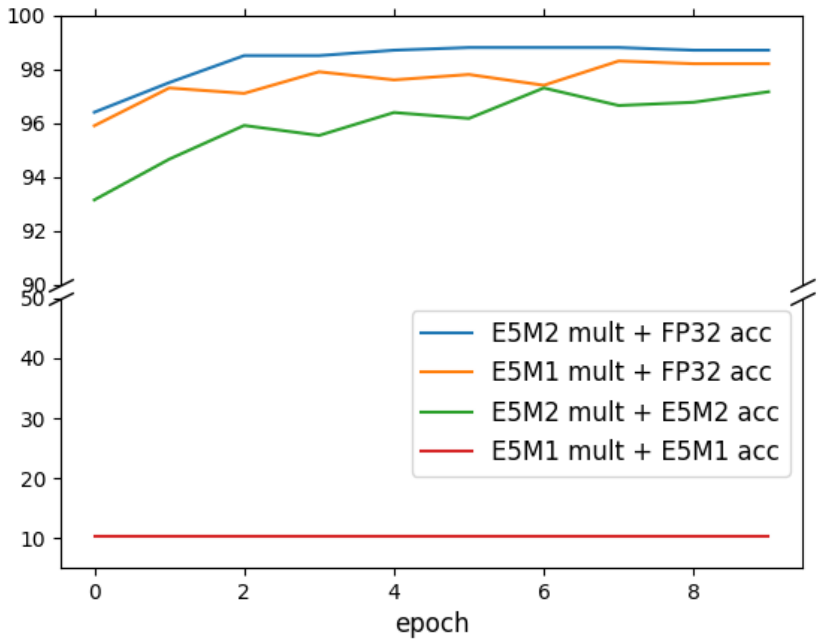


Figure 1.1: Validation accuracy observed using LeNet5+MNIST with low-precision multiplication coupled with FP32 accumulation (blue/orange) and low-precision accumulation (green/red).

In addition, almost all previous work on low-precision training has avoided directly examining hardware implementation complexity. Accordingly, the precise hardware resources saved by replacing the FP32 with a low-precision data type is left ambiguous. Reducing the resource demand per multiply-accumulate (MAC) unit allows one to deploy more computational power within the same hardware area; it is essential to understand the resource cost of each configuration.

In this thesis, to address these ambiguities, we apply different low-precision data formats to the MAC units within a GEMM accelerator used

for DNN training and investigate the impact on training accuracy and hardware resource utilization. In addition, we also explore a variant of floating-point encoding for low-precision multipliers to achieve a better resource-accuracy tradeoff.

1.2 Approach

To investigate both prediction accuracy and hardware resource utilization under various low-precision configurations, we have developed a framework named Archimedes-MPO, where "MPO" stands for mixed-precision operations. The framework carefully models the GEMM computation behavior using customizable low-precision data formats, while providing GPU and FPGA acceleration support to make training practical.

With the FPGA acceleration option, the framework produces a systolic array containing MAC units that use the specific targeted low-precision formats; these targeted precisions cannot be modified at run-time. Since this framework allows one to choose different data formats or precision for each operator within the GEMM kernel, we can test the resource-accuracy tradeoff of a mixed-data format configuration, for example using a floating-point multiplier and a fixed-point accumulator, as well. By implementing a custom hardware kernel, we can measure the resource utilization of the low-precision configurations, and actually deploy the accelerated system onto an FPGA development board.

Moreover, the FPGA acceleration option provides several different implementations for floating-point multipliers used in a GEMM kernel. Each

implementation supports different combinations of exceptional values, and based on the complexity of the internal logic, the corresponding LUT count differs among implementations. By testing the various multiplier implementation, we can study the trend in resource utilization; such an investigation is missing in many prior arts that do not provide hardware implementation details.

On the other hand, the GPU implementation is used for rapid evaluation of the different low-precision configurations, as the number of available FPGA boards is limited whereas our server compute farm has over 100 GPUs available. To maintain consistency, the GPU version framework is bit-accurate with the behavior of the FPGA version. The GPU kernels are written in CUDA.

1.3 Contribution

This work studies low-precision training behavior by considering different data formats and/or precisions for the different operations within a MAC unit. In particular, it investigates resource-efficient multiplier implementations to achieve better resource-accuracy tradeoffs, as well as their impact on a downstream low-precision accumulators that use different data formats.

We evaluate two main training scenarios: a small size scenario using LeNet5 [27] and an MLP with the MNIST [27] dataset, and a medium size scenario using the CIFAR-10 [25] dataset with VGG16 [39] and ResNet20 [17] networks. Time restrictions have precluded the use of a large scale scenario involving the ImageNet [9] dataset, and moreover, such large scenarios may

be inappropriate for embedded edge device training; however, we occasionally use a subset of ImageNet called Imagewoof [18] to test the viability of the observed trend in larger datasets.

The major contributions of the thesis are the following:

- We provide a new extensible framework called Archimedes-MPO to study resource-accuracy tradeoffs when applying different low-precision data types to DNN training; the framework supports various multiplier implementations with different exceptional values support, and different precision or data formats for multipliers and accumulators comprising the matrix-multiplication kernel.

- We show that relaxing floating-point value representation in multiplier beyond IEEE-754 standard floating-point arithmetic is beneficial for area-saving and accuracy; in particular, removing rounding, NaN, and subnormal output supports all reduces resource utilization and improves accuracy, whereas eliminating subnormal input incurs accuracy drop to some extent.

- We observe that simultaneously using a floating-point multiplier and a fixed-point accumulator shows promise of good resource-accuracy tradeoff because it leads to an efficient MAC implementation, but it requires format conversion overhead as well as further system-level optimization due to the wider output length. Since GEMM hardware kernels can differ in how, where and when accumulation is done, there is a rich design space for alternative accumulation strategies which deserves further investigation.

- Fxed-point accumulator size is more sensitive to the target model/task size than the multiplier. Hence, it requires further investigation to apply the mixed configuration in larger tasks.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides the background knowledge regarding data formats, DNN training and inference, and related work on low-precision training. Chapter 3 introduces the Archimedes-MPO framework developed for the experiments and investigates hardware implementation overhead. Chapter 4 demonstrates the training results and discusses the observations. Chapter 5 concludes the thesis and presents ideas for future work.

Chapter 2

Background

In this section, we discuss the background knowledge required to understand low-precision training and related work. First, we introduce frequently used low-precision data types. Second, we look at the GEMM kernel and the FPGA implementation used as the basis for this work. Next, we examine the structure of two common neural networks, MLP and CNN, as well as the difference between inference and training tasks. After that, we look at prior work which uses low-precision data types for neural network training and explain how it impacts the trained model performance. Although those prior arts achieved remarkable results, many of them did not explain the details of the associated hardware implementations. Therefore, we will introduce other works which have implemented low-precision training or inference methodologies on FPGAs in a separate subsection.

2.1 Low-Precision Data Types

There are several low-precision data types used for training and inference. The three most common formats are: floating-point, fixed-point, and block floating-point.

2.1.1 Floating-Point Format

Floating-point data types have three fields: a sign, exponent, and mantissa. Usually, IEEE-754 rules are followed which also specifies the number of bits for the exponent and mantissa set according to the overall size of a few standard data types.

In floating-point formats, a bias term, specific to the target data type, is added to the raw exponent value to represent all possible exponent values as positive integers. Assuming the target floating-point datatype has one *sign* bit, x exponent bits e_i, and y mantissa bits m_i, we can define the exponent value E, the mantissa value M, and a real number `normalized value` as follows:

$$\mathrm{E} = \sum_{i=1}^{x} e_i \times 2^i \qquad (\mathtt{e}_i = 0 \,\mathrm{or}\, 1) \tag{2.1}$$

$$\mathrm{M} = \sum_{i=1}^{y} \mathtt{m}_i \times 2^{i-y-1} \qquad (\mathtt{m}_i = 0 \,\mathrm{or}\, 1) \tag{2.2}$$

$$\mathtt{bias} = 2^{(x-1)} - 1 \tag{2.3}$$

$$\mathtt{normalized\ value} = (-1)^{sign} \times 2^{(E-\mathtt{bias})} \times (1 + M) \tag{2.4}$$

The standard for DNN training is FP32, which uses 32 bits to represent each value. In contrast, bfloat16 and FP16 use only 16 bits, and FP8 uses only 8 bits. Thus, replacing the FP32 values with any of these data types reduces memory use by 50 or 75% (Figure 2.1).

In this work, we express floating-point formats as ExMy, meaning the use of x bits for the exponent field and y bits for the mantissa field.

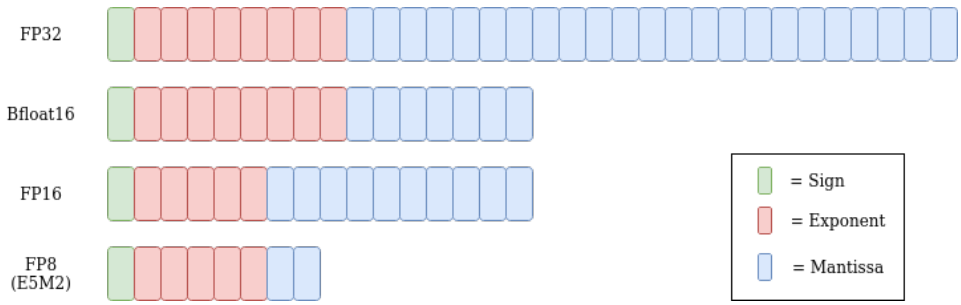


Figure 2.1: Floating-point data type examples: FP32 is comprised of an 8 bits exponent and a 23 bits mantissa, bfloat16 assigns 8 bits for exponent as well but has only 7 bits for mantissa; FP16 has 5 bits exponent and 10 bits for mantissa. FP8 has even less bits in total and the exponent-mantissa breakdown can differ among prior arts. The figure shows an FP8 format assigning 5 bits for exponent and 2 bits for mantissa as an example.

Normalized floating-point numbers represented by Equation 2.4 use values of E ranging from 1 to $2^x - 2$, where x is the number of exponent bits. Reserved E values of 0 (used for subnormals) and $2^x - 1$ (used for infinity and NaNs) are discussed below.

Subnormal Values

An E field value of 0 represents **subnormal** values which are additional numbers between 0 and the smallest normalized value. When the E field is 0, the implied leading digit of the mantissa becomes 0, so the 1 in Equation 2.4 becomes 0 for subnormal values as follows:

$$\texttt{subnormal value} = (-1)^{sign} \times 2^{(E-\texttt{bias})} \times (0 + M) \tag{2.5}$$

The biggest benefit of subnormal values is the ability to represent values very close to 0. Considering the E5M2 data type as an example, the mini-

mum exponent value E5M2 can express is -14 (i.e., $0 - (2^{5-1} - 2)$). Hence, the smallest non-zero normal value of E5M2 is `6.104E-5`. However, using subnormal values, E5M2 can represent `1.526E-5`.

Unfortunately, hardware support for subnormals can be costly because it requires different logic than normalized values.

NaN and Infinity

An E field value of $2^x - 1$ represents either **Infinity** or "Not-a-Number" (**NaN**). An infinity might occur after $1/0$ or $-\log(0)$, while a NaN might occur after $0/0$, $\sqrt{-1}$, or Infinity−Infinity.

In conventional IEEE-754 encoding, a value with the exponent of $2^x - 1$ and the mantissa of all 0s represents Infinity, while all other mantissa values represent NaNs. Since NaNs have little value, a practice called NaN boxing is sometimes used to hide information within the mantissa field (as long as the mantissa is non-zero). In this thesis, since NaNs will never be encountered during GEMM, we will suggest a new encoding for Infinity and NaNs to extend the representable range.

All operations require special treatment when they encounter Infinity or NaN results. Infinities propagate as one might expect, whereas NaNs are "contagious" and almost always infect the output value (with only a few exceptions).

Hardware support for Infinity and NaN values can be costly because it requires different logic than normalized values.

2.1.2 Floating-Point Arithmetic

This section will introduce the floating-point arithmetic operations used in matrix multiplication: multiplication and addition. First, we cover floating-point multiplier logic in the following subsection and then floating-point adder logic in the subsequent subsection.

Multiplier

Floating-point values represented by Equation 2.4 (or Equation 2.5 for sub-normal numbers) are multiplied as follows. We denote the sign, exponent, and mantissa of operand-A/operand-B/product as $S_a/S_b/S_p$, $E_a/E_b/E_p$, and $M_a/M_b/M_p$ respectively.

$$\begin{aligned}\texttt{product} &= (-1)^{S_a} \times 2^{(E_a - \texttt{bias})} \times (1 + M_a) \\ &\quad \times (-1)^{S_b} \times 2^{(E_b - \texttt{bias})} \times (1 + M_b) \\ &= (-1)^{S_a \oplus S_b} \times 2^{(E_a + E_b - 2 \times \texttt{bias})} \times (1 + M_a) \times (1 + M_b) \end{aligned} \tag{2.6}$$

$$S_p = S_a \oplus S_b \tag{2.7}$$

$$\begin{aligned} E_p &= E_a + E_b - 2 \times \texttt{bias} + \texttt{bias} \\ &= E_a + E_b - \texttt{bias} \end{aligned} \tag{2.8}$$

$$(1 + M_p) = (1 + M_a) \times (1 + M_b) \tag{2.9}$$

As shown in Equation 2.6, the sign of the product is computed by `XOR`-ing the sign of two values. Since the scaling values of both operands use the

same base of 2, the exponent of the product can be calculated by summing their exponent values. More precisely, the unbiased exponent value of the product is defined as the summation of unbiased exponent values (i.e., $(E_a - \texttt{bias}) + (E_b - \texttt{bias})$); the final biased exponent value, which we need for representing the value in the floating-point data format, is determined by Equation 2.8.

Meanwhile, we can compute the mantissa of the product by multiplying the mantissa of two values, including a leading 1 for normal values and a leading 0 for subnormal values. The product in Equation 2.9 yields about twice as many bits as can be stored in M_p, so rounding is necessary.

Adder

Unlike multiplication, the addition of two mantissa values requires them to be on the same scale. Hence, if two operands have different exponent values, the mantissa corresponding to the smaller exponent value needs to be shifted to the right by the exponent difference so that the addition between them is computed with aligned exponents as shown in Figure 2.2.

Consider the addition of `1.5626` (operand-A) and `0.4375` (operand-B) as an example in Figure 2.2. In this example, we assume the target data type is E5M4. The value of operand-A, `1.5625`, has the unbiased exponent (i.e., $E - \texttt{bias}$) of `0` and mantissa of `b1001`. Given that E is not zero, this value is normalized so the leading bit of the mantissa is set and the entire significand is defined as `1.1001`. Similarly, the value of operand-B, `0.4375`, has the unbiased exponent of `-2` and the entire significand of `1.1100`. Since the two values have different exponent values (`0` vs. `-2`), the adder shifts

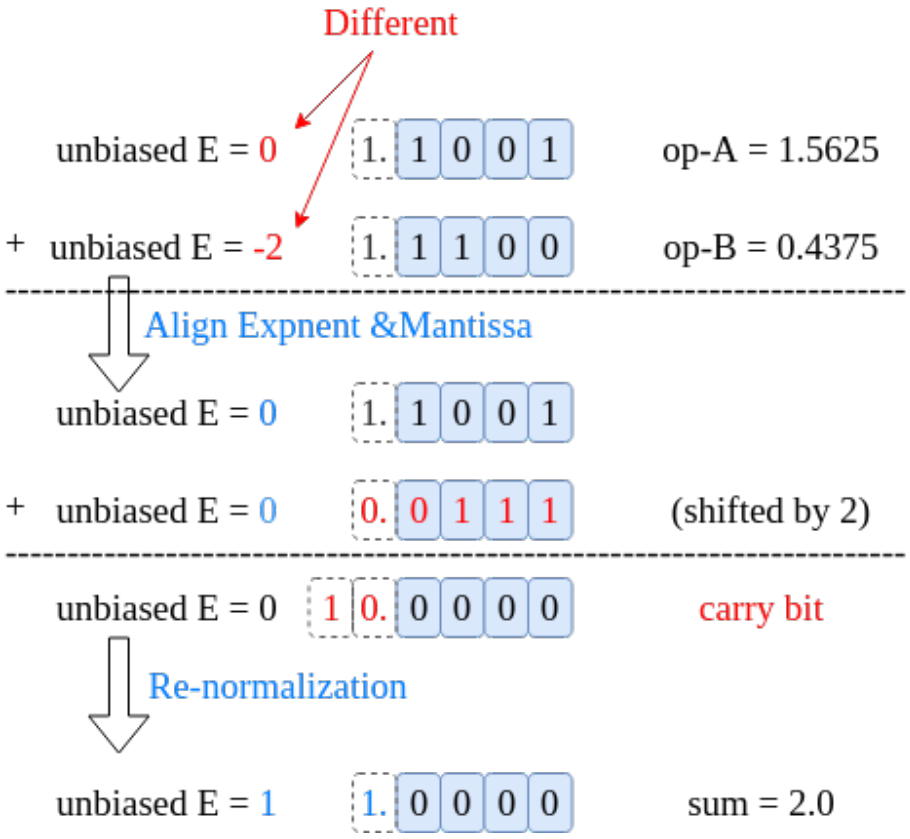


Figure 2.2: Mantissa addition including alignment before the summation and re-normalization of the sum value.

the mantissa of operand-B to the right by $0-(-2)=2$ bits. Then the adder sums up the significand of operand-A, `1.1001`, and the shifted significand of operand-B, `0.0111`. The addition `1.1001+0.0111` generates the temporary sum of `10.0000`, so the adder renormalizes the value such that the leading bit will be the left-most set bit.

Note that renormalization, caused by small differences in the two significands, may result in much larger shift. This requires a barrel shifter, which is a fairly large hardware structure. In some FPGA implementations, this

barrel shifter may be realized using a hard multiplier.

2.1.3 Fixed-Point Format

Another data format to introduce is the fixed-point data type. A format called Q-notation is used to specify how many bits of the fixed-point data type is used for representing integer/fraction part of the data type. Assuming the target precision assigns i bits for the integer part and f bits for the fraction part, we express the precision as Q$i.f$ as shown in Figure 2.3. The primitive fixed-point data types, such as `uint8`, can be also represented in Q-notation as Q8.0, because no bits are assigned to represent the fraction field.

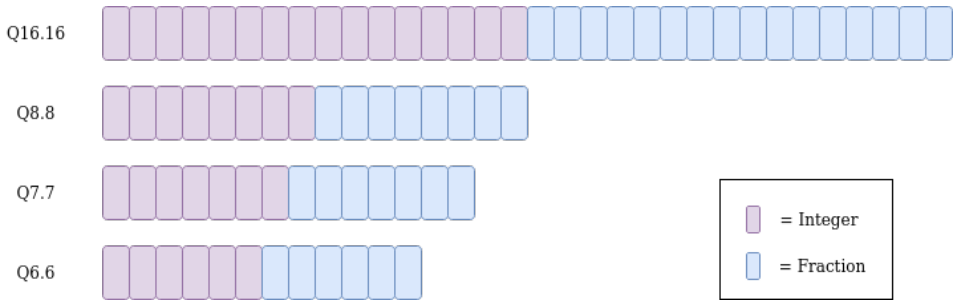


Figure 2.3: Q-notation Examples

The Q-notation values do not have an exponent field to scale the mantissa value. Hence, when we convert a floating-point value into Q$i.f$ format, we need to shift the entire mantissa value based on its exponent value. Assuming that a floating-point value has a mantissa value of M and an exponent value of E, then we must first restore the value to its non-normalized form by left-shifting the bits in (1+M) by an amount equal to $E - \texttt{bias}$ (or right-shifting if the shift amount is negative). Then, we capture i bits to the

left of the decimal, and f bits to the right.

2.1.4 Fixed-Point Arithmetic

Due to the lack of an exponent field, the Q-notation data types sacrifice the representation range compared to the floating-point data types of the same bit length. However, this simplicity enables us to implement the Q-notation arithmetic without any alignment nor subnormal handling logic and apply the computations in the same way we add integers.

Multiplier

Since Q-notation values are essentially shifted integer values, we can calculate products using simple integer multiplication, but some truncation or rounding afterward is required. For example, we can calculate the product of two Q3.3 values, `001.100` and `000.100`, as `001.100` $\times$ `000.100` = `000000.110000`. The multiplication of two Q$i.f$ values generates a product in Q$(i \times 2).(f \times 2)$ format. To convert this to Q$i.f$, we must shift out the least significant f bits (i.e., 3 bits for Q3.3) and cut off the most significant i bits so that the output value will be in Q$i.f$ format as well (Figure 2.4). Bits shifted out represent lost precision which can be mitigated by rounding; bits cut off represent overflow.

Adder

The addition of values that use the same Q-format is much simpler than floating-point addition. The adder does not need any alignment or renormalization after the summation, so it directly computes the sum of input

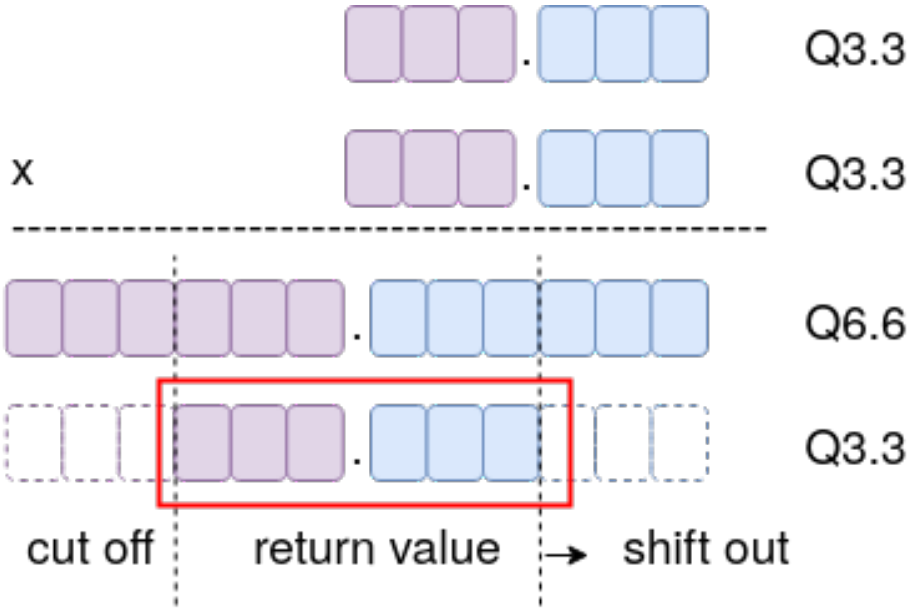


Figure 2.4: Multiplication of Q-notation Values

values. On overflow, the output may be saturated or simply wrap around.

2.1.5 Block Floating-Point Format

Both floating-point and fixed-point formats have strong and weak points. Floating-point data types can represent a broader range of values because of its exponent field. In contrast, fixed-point addition is always exact and requires no shifting or rounding hardware, and fixed-point multiplication doesn't need to consider adjusting exponent fields.

The block floating-point format was introduced with the motivation to take the strengths of those two data types at the same time. In block floating-point format, a block or group of floating-point values located within the same tensor are factored to share one exponent value, and each value's

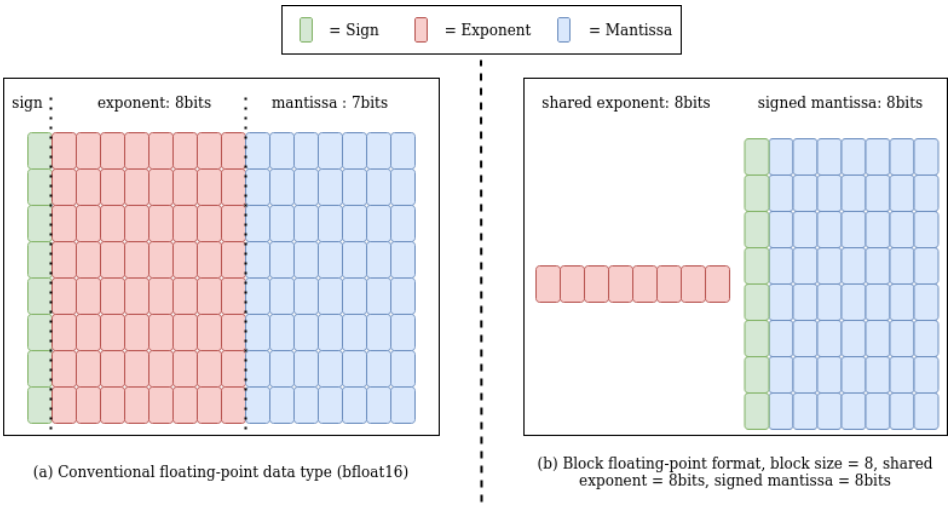


Figure 2.5: Difference between floating-point data type (bfloat16) and block floating-point data type

original significand is rescaled based on the shared exponent value (Figure 2.5). The block size depends upon the implementation, but it may contain 8 values, for example.

The most significant advantage of the block floating-point data type is the efficient implementation of the accumulator. When we use floating-point data types for matrix multiplication, the mantissas of the multiplier output and accumulator output need to be aligned to use the same exponent. In contrast, summation using block floating-point does not need realignment. It can calculate the block output as a fixed-point operation since all values within the same block are guaranteed to have the same exponent value. Alignment is only necessary when the sum of this block is added to the sum of another block that has a different shared exponent value.

However, block floating-point data types also suffer from potential infor-

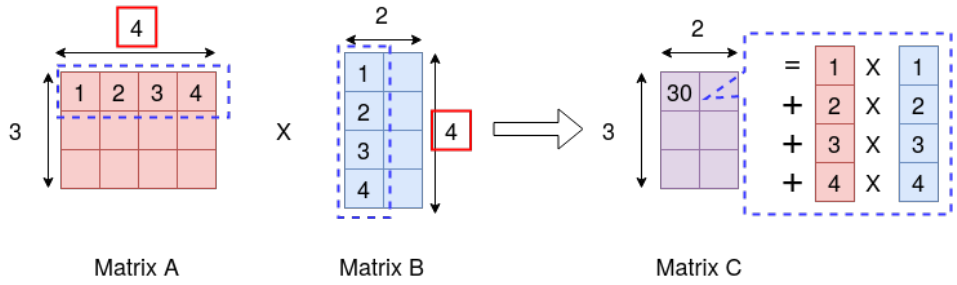


Figure 2.6: General Matrix Multiplication

mation loss. Since all values assigned to the same block are rescaled to have the same exponent value, a large difference between the shared exponent value and original exponent value can cause truncation of the mantissa.

2.2 General Matrix Multiplication (GEMM)

The primary computation in DNN training is general matrix multiplication (GEMM). GEMM takes two inputs, matrix A and B, and computes the dot product between each row of matrix A and each column of matrix B; therefore, the number of columns in matrix A and the number of rows in matrix B need to be the same. Assuming that matrix A has a dimension of (3×4) and matrix B has a dimension of (4×2), the GEMM computation to calculate the product matrix C (3×2) is depicted as Figure 2.6.

In GEMM computation, we can reuse the input data to generate multiple output values. For example, we need the 1st row of matrix A and the 1st column of matrix B to compute the value located at $(1, 1)$ of the output matrix C. The computation of $(2, 1)$ and $(3, 1)$ values of matrix C can re-use the 1st column of matrix B. Hence, we can decrease the number of memory

reads from three to one if we schedule the computation appropriately. The dot product itself requires a MAC unit to compute the four products and accumulate them into a single result.

We use the GEMM-HLS project [8] as a base of our FPGA GEMM kernel. This implementation employs a linear array of PEs to implement GEMM efficiently. Each PE contains more than one multiply-accumulate (MAC) unit. As shown in Figure 2.7, only the first PE in the systolic array is connected with the data reading/writing module; all data transferring is done through this first PE. Each PE takes the matrix A input value only if the index of the value corresponds to the PE's index, and sends the value to the subsequent PE otherwise. In contrast, all PEs reuse the same value for matrix B inputs. After the computation completes, the output values from each MAC unit are sent through the linear array until they reach the memory writing module.

The GEMM-HLS kernel tiles the input values when the target input/output matrices are larger than the systolic array size, but each MAC unit in the GEMM kernel adds up all input values corresponding to a certain index of the output matrix before reusing the MAC unit for another index of the output matrix; hence, all the accumulation is performed within the GEMM kernel and the implementation does not require a software wrapper to apply further summation outside of the kernel.

In this thesis, the GEMM-HLS kernel is heavily modified to support different multiplier and accumulator precisions, and to convert between FP32 and the intended precision. Also, we made several optimizations, such as removing all hardware multipliers that are inferred outside of the PEs.

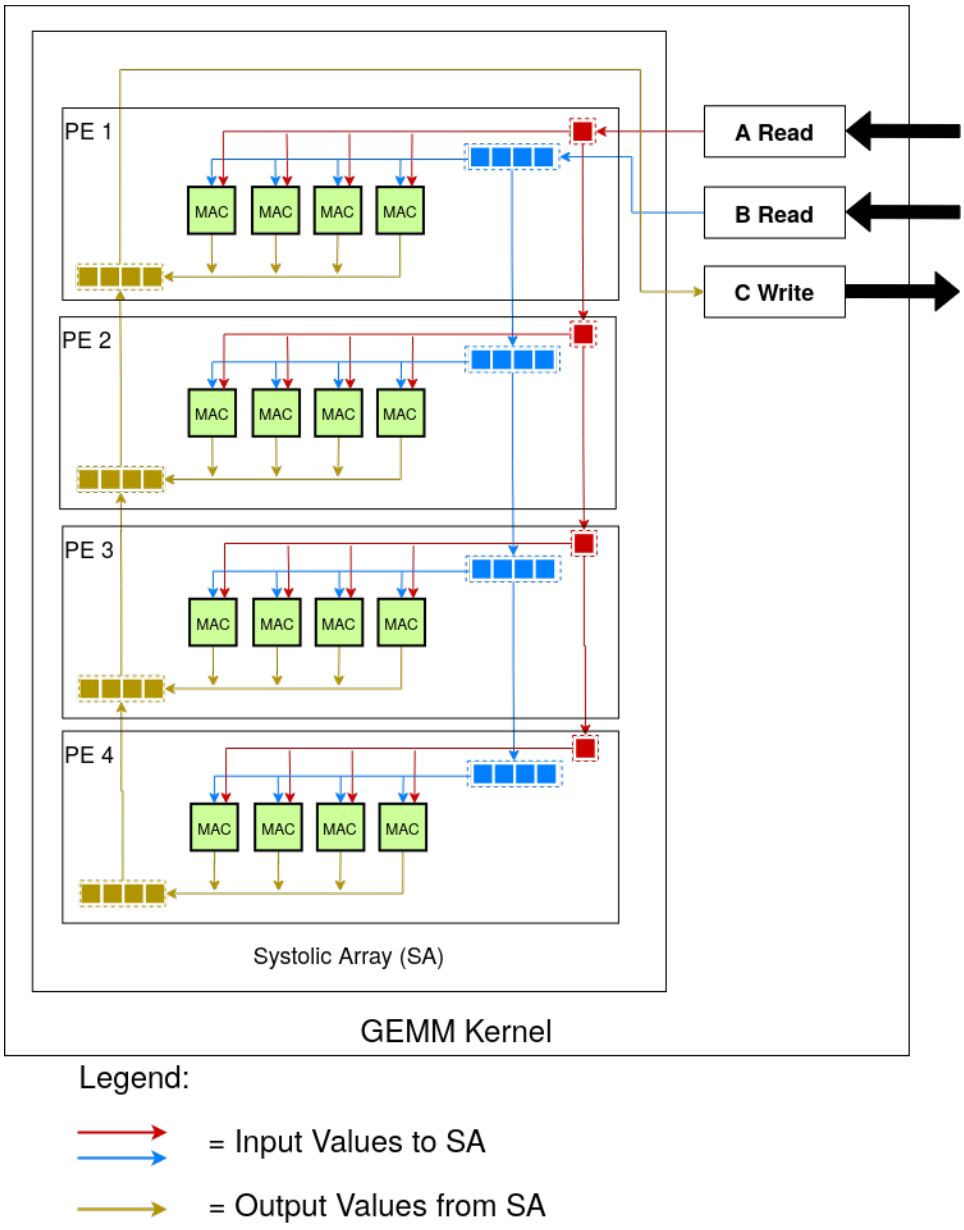


Figure 2.7: Overview of GEMM implementation for FPGA (4 PEs and each PE has 4 MAC units)

2.3 Deep Neural Networks

In this section, we briefly review two types of DNNs, differences between inference and training, and a training technique known as loss scaling.

2.3.1 Two DNN Model Types

Neural networks have been developed by mimicking the structure of the actual brain. Similar to neurons constructing brain networks, artificial neural networks also consist of a unit called a perceptron. Each perceptron holds a weight value w to scale input values, and the output value y is calculated as $y = w \cdot x$ assuming input x. Utilizing multiple perceptrons allows models to approximate more complex multivariable equations.

However, simple perceptron models cannot learn patterns based on linearly non-separable equations, such as the exclusive-OR operation. To address this problem, activation functions are implemented to inject nonlinearity between the different models. Sigmoid, Tanh, and Rectifier Linear Unit (ReLU) are the most frequently used activation functions. The ReLU function is prevalent in recent research due to its stability and light computation. ReLU forces negative input values to zero, while allowing positive values to pass unchanged.

MLP Models

One of the most basic models, the multilayer perceptron (MLP), consists of multiple perceptron layers. MLP models applied to image processing tasks usually have activation functions inserted between layers to generalize better

for image recognition tasks. MLP models can theoretically approximate any continuous function arbitrary well, but the model size expands quickly when the input and output dimensions of the model increase. Therefore, training vanilla MLP models targeting large datasets such as ImageNet is not straightforward from multiple perspectives, particularly exploring training hyperparameters and the computational effort required.

The main computation applied in batch training of MLPs is matrix multiplication. It forms an input matrix containing one input vector from each batch item, and multiplies this by a weight matrix to produce an output matrix containing one vector per batch item for the next layer. Therefore, the GEMM computation module mentioned earlier fits this model well.

CNN Models

The convolutional layer was inspired by the network structure of the visual cortex and can handle computer vision tasks with fewer weights than MLP. The layer type used in MLP is also known as a fully-connected layer and connects all input perceptron values to each output perceptron. On the other hand, the convolutional layer correlates a patch of the input values to each output value (Figure 2.8). Therefore, the convolutional layer does not connect all input values to all of its output values, and this sparsity vastly reduces the number of compute operations. This structure provides translational invariance, so input image patterns can shift left/right/up/down and still be recognized.

There are several ways to compute a convolution. The simplest method is directly computing the convolution by iterating through the input value

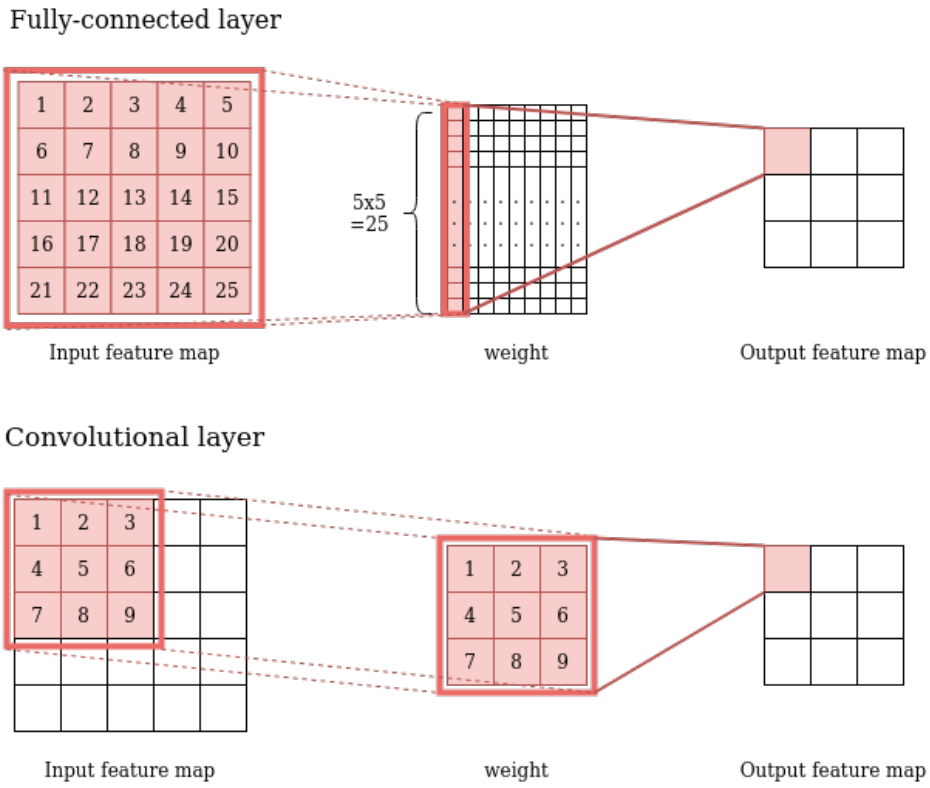


Figure 2.8: Connection of Fully-Connected and Convolutional Layers

matrix with a sliding window that matches the desired convolutional kernel size (Figure 2.9 upper). However, this technique cannot easily employ highly optimized software matrix multiplication libraries such as BLAS [3].

A method that decouples the multiply-add operations from memory data layout is proposed to address this issue. This method adopts two data transformation functions, namely `im2col` and `col2im`, along with the GEMM computation. First it applies the `im2col` function to the input and transforms the input feature map so that the values convolved together will be located in the same column (Figure 2.9, lower top). Hence, the dimension of

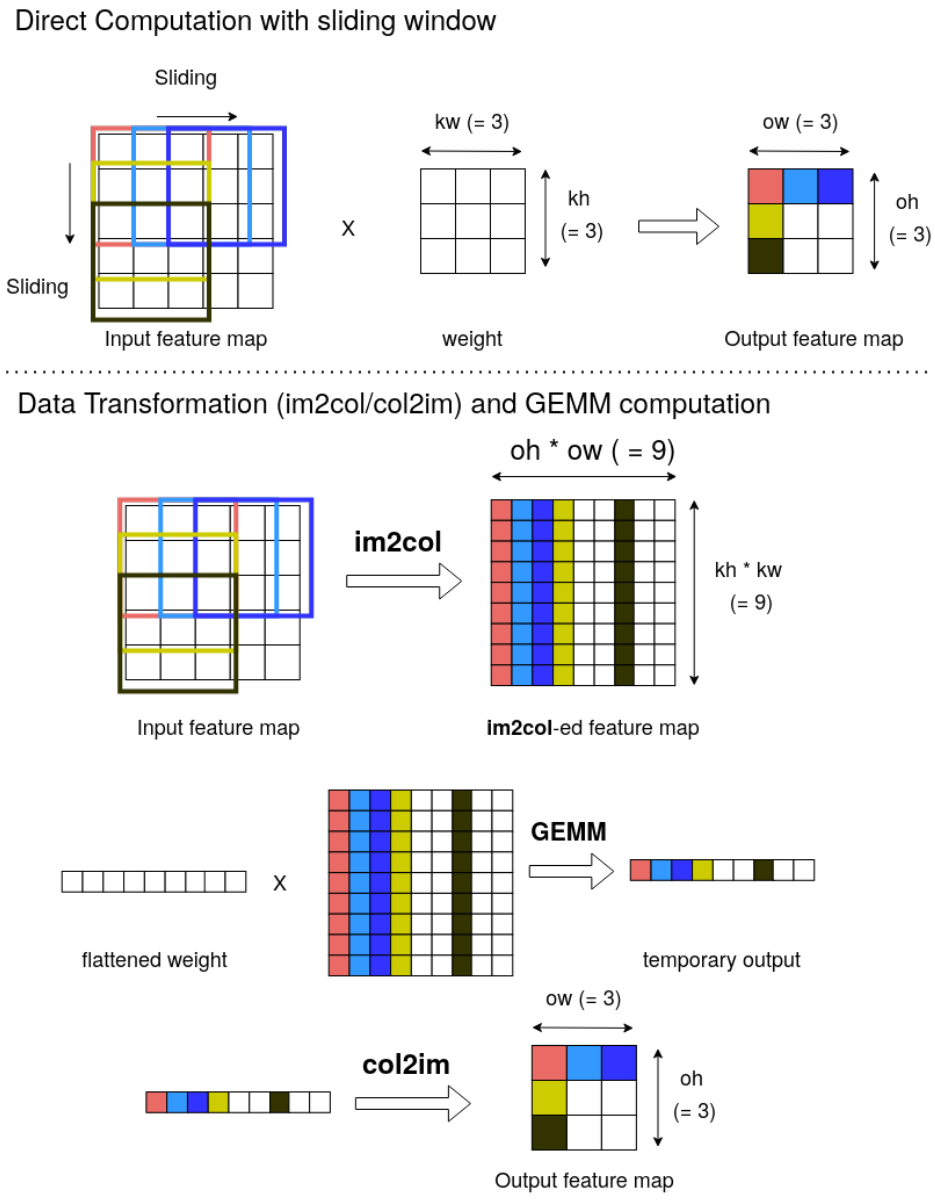


Figure 2.9: Different methods to compute convolution: direct convolution in a sliding window manner (upper), and the combination of im2col/col2im and GEMM (lower)

the transformed matrix would be (output weight × output height) × (kernel width × kernel height). We can compute the convolution as the GEMM of a flattened weight matrix and the transformed matrix, and then restore the output feature map by reshaping the temporary output with the `col2im` function.

Since the transformed matrix generated by `im2col` function includes replicated values, the memory footprint increases compared to direct convolution. However, the benefit of using the same GEMM routine as the fully-connected layer is still significant even considering this drawback.

The `im2col` and `col2im` functions involve data movement which may take a significant proportion of the overall runtime. With some effort, it is possible to generate the output of `im2col` on the fly and immediately consume the values without resorting to storing the entire transformed matrix in memory. This thesis does not include such an optimization.

CNN models frequently include downsampling layers and normalization layers. However, since both layers are generally less compute-intensive than fully-connected and convolutional layers, they do not benefit much from acceleration. Additionally, because they are structured very differently from convolutional and fully connected layers, they cannot re-use the existing accelerated GEMM kernel. Therefore, this thesis only focuses on accelerating GEMM for fully-connected and convolutional layers.

2.3.2 Inference and Training

To utilize DNN models in a new application, we first need to train the model from scratch and then deploy the trained model onto the device where the

application runs. This requires two distinct but related forms of computation: one for inference (deployment), and one for training.

Inference uses pre-trained models with fixed weight values to make a prediction regarding given inputs. This calculation involves a forward pass through the model with one GEMM calculation per convolutional layer, for example. In contrast, training must determine the weights used in a target model using input data paired with expected outcome labels indicating the ground truth values. During training, weight values in the model are updated through a technique known as back-propagation, which reduces the error between the prediction values calculated by the current model and ground truth values (Program 2.1). As part of back-propagation, the model needs to compute each layer's gradient, the delta values used for weight updates (Figure 2.10). The gradient computation for each layer consists of two extra GEMM computations, one for calculating weight gradient and another for computing gradient of the input feature map, which causes the elapsed run-time to be approximately three times longer than the equivalent inference task.

Due to several differences, low-precision or quantized training is more challenging than doing the same for inference. First, in inference tasks, we can quickly estimate the value range of both input and weight values. Since the weight values in pre-trained models are fixed, we can use profiling to estimate the value range. Similarly, the value range of inference input values can be estimated by taking a small number of random crops from the training set. On the other hand, the value range that appears during model training is hard to predict and changes as the training proceeds. Besides this

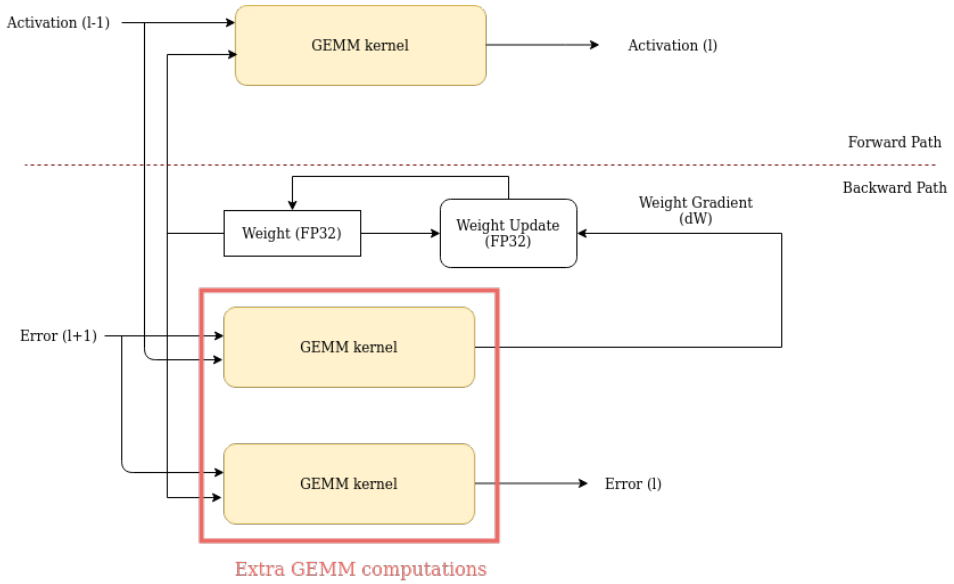


Figure 2.10: Forward and backward propagation paths of a single layer l.

problem, recent work has shown that back-propagation computation requires more expression range compared to the forward-propagation path [40].

2.3.3 Loss Scaling

Micikevicius et al. [32] explored the distribution of activation gradients and determined that they are biased to the values close to 0 compared to the activation distribution. Due to the bias, many of the gradient values are below the minimum value that low-precision data types can represent, thereby leaving most of its representation range unused (Figure 2.11).

Based on this observation, Micikevicius et al. [32] presented a **loss scaling** technique that scales the loss values with a particular factor (e.g., 1024) so that gradient values better reside within the representable range of a low-precision data type. As a result, shifting the gradient value range towards

Program 2.1 Training Algorithm

Input: randomly initialized model `M` (having `L` layers), image data $\texttt{D}_{image}$, label data $\texttt{D}_{label}$

Output: trained model `M` (having `L` layers)

```
1: for each epoch do
2:   for each batch do
3:     A_L ← M.forward_propagation(D_image[batch])
4:     loss ← CrossEntropyLoss(A_L, D_label[batch])
5:     W_delta ← M.backward_propagation(loss)
6:     M.weight_update(W_delta)
7:   end for
8: end for
```

infinity prevents many gradient values from being truncated down to 0 and improves low-precision model accuracy.

Adaptive Loss Scaling

A problem with the loss scaling technique is how to pick the best loss-scaling factor during training. As shown in Figure 2.12, training behaves differently depending upon the loss-scaling factor. Choosing an appropriate factor is essential for some low-precision training (e.g., E4M1 and E4M2), but searching for the best parameter takes extra time.

To address this problem, NVIDIA introduced an adaptive loss-scaling technique [35]. The basic idea is to start the training with a large loss-scaling value S. If no overflow is detected within N iterations, the value of S is doubled. In contrast, if any computation causes overflow during the iteration, the weight update for this iteration is skipped and S is halved. The pseudo-code of the process is provided in Program 2.2.

In this thesis, we apply this adaptive loss-scaling technique with a start-

Program 2.2 Model training with adaptive loss-scaling (adapted from [35]). Adaptive loss-scaling specific lines are shown in light blue.

Input: randomly initialized model `M` (having `L` layers), image data $\texttt{D}_{image}$, label data $\texttt{D}_{label}$, initial loss-scaling factor `S`, threshold `N`

Output: trained model `M` (having `L` layers)

for each *epoch* **do**
  for each *batch* **do**
    $A_L \leftarrow$ M.forward_propagation($\texttt{D}_{image}[batch]$)
    loss $\leftarrow$ `CrossEntropyLoss`($A_L, \texttt{D}_{label}[batch]$)
    loss_scaled $\leftarrow$ loss $\times$ `S`
    $W_{delta} \leftarrow$ M.backward_propagation(loss_scaled)
    has_Inf_NaN $\leftarrow$ `CheckInfNaN`(W_{delta})
    if has_Inf_NaN is **true** **then**
      `S` $\leftarrow$ `S` $\times$ 0.5
      continue
    else
      $W_{delta} \leftarrow W_{delta}$ / `S`
      M.weight_update(W_{delta})
      if has_Inf_NaN is **false** for `N` iterations **then**
        `S` $\leftarrow$ `S` $\times$ 2
      end if
    end if
  end for
end for

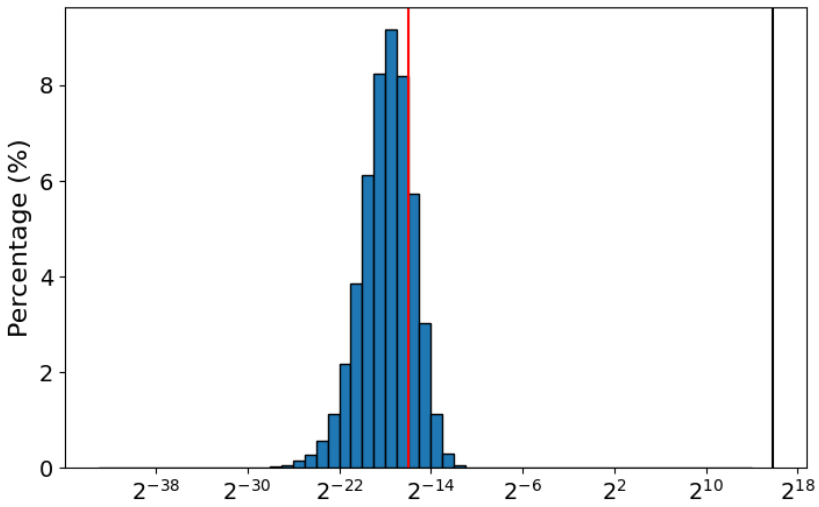


Figure 2.11: The value distribution of activation gradient collected from all layers comprising a ResNet20 model. The red/black line shows the minimum/maximum value that E5M2 can express. Note that 49.68% of the gradient values are 0.

ing value S of **1024** and the count threshold N of **200**.

2.4 Related Works

A snapshot comparing some key previous work related to low-precision deep learning is provided in Table 2.1. Below we will summarize related work that explores low-precision training. We note that most papers are vague about exact details. For example, in some cases only the storage format for the weights and activations use a smaller data size, but actual calculations are done at a higher precision. Most work does not indicate how it treats overflow conditions. Sometimes, the exact training frameworks, and

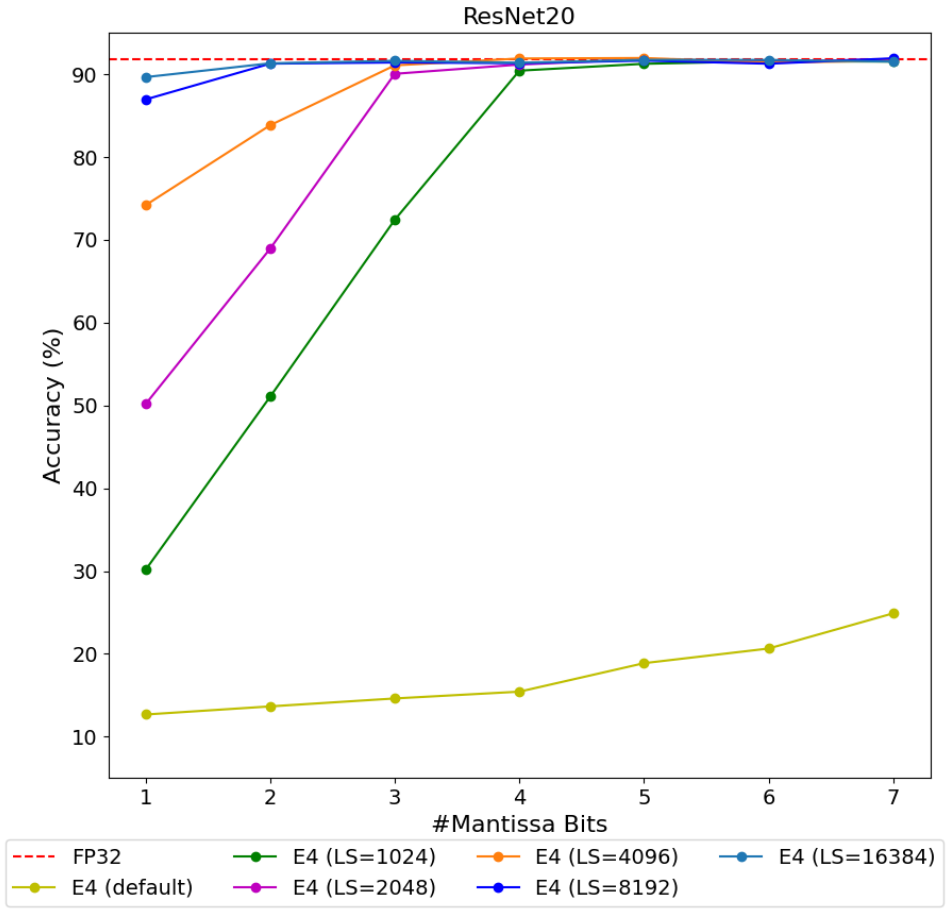


Figure 2.12: The best prediction accuracy achieved by the training using E4My multipliers with different loss scaling factors (1024, 2048, 4096, 8192, or 16384).

Table 2.1: Related Work Highlights

paper	forward		backward		task	HW
	mult	add	mult	add	type	detail
Micikevicius et al. [32]	FP16	FP32	(same)	(same)	training	NO
Kalamkar et al. [23]	bfloat16	FP32	(same)	(same)	training	NO
Wang et al. [46]	E5M2	E6M9	(same)	(same)	training	NO
Sun et al. [40]	E4M3	-	E5M2	-	training	NO
Sun et al. [41]	INT4	E6M9	FP4	(same)	training	NO
Banner et al. [2]	INT8	INT32	(same)	(same)	training	NO
Fu et al. [14]	INT4-8	-	INT6-8	-	training	NO
Xilinx.com [50]	E3M3	INT21/INT14	N/A		inference	YES
Wu et al. [47]	E3M4	INT23-32	N/A		inference	YES
Fox et al. [12]	BFP(E8M8)	32b(INT?)	(same)	(same)	training	YES
Fox et al. [13]	BM6(E2M3)/-	INT20	(same)	(same)	training	YES

how they are modified to provide (possibly accelerated) training, are not described. Few, if any, works make a distinction between data storage formats and arithmetic operator precision or consider the precise area costs and accuracy trade-offs related to a hardware implementation of the arithmetic.

Floating-point training

The low-precision training exploration started with the bfloat16 [23, 53] and IEEE-754 standard FP16 [32, 56] data types. Both use 16 total bits, but FP16 is E5M10 and bfloat16 is E8M7. The logic behind bfloat16 is to keep the exponent field the same size as FP32 and truncate only the mantissa; this makes it easier to use bfloat16 as a storage format to save memory, but use existing FP32 hardware for calculations. Hence, bfloat16 has similar dynamic range but lower precision than FP32. In DNN training, dynamic range is usually deemed more important.

Micikevicius et al. [32] altered an FP32 GEMM kernel to use FP16 for multiplication but keep accumulation as FP32. This work showed some

accuracy loss, which was fully recovered using (non-adaptive) loss scaling.

Kalamkar et al. [23] explored the use of bfloat16 instead of FP32. Similar to [32], the work also replaced GEMM multiplication with bfloat16, leaving the GEMM accumulator as FP32. This work showed that bfloat16 achieves similar results as FP32 without the need for loss scaling or any hyperparameter re-tuning.

Some recent works have further reduced data types to various 8-bit floating-point formats which we'll collectively call FP8 [31, 38, 40, 46]. Wang et al. [46] showed that using an FP8 (E5M2) multiplier and a custom FP16 (E6M9) accumulator can achieve the results close to the FP32 baseline. To avoid catastrophic accuracy loss, they use a chunk-based accumulation strategy with loss scaling and stochastic rounding.

Sun et al. [40] also investigated the use of an FP8 GEMM multiplier and showed that using different exponent-mantissa breakdowns for forward and backward propagation is effective. In the experiments, the configuration using FP8 (E4M3) for the forward GEMM multiplication and FP8 (E5M2) for the backward GEMM multiplication achieved the best prediction-bitwidths tradeoff. The results indicate forward propagation requires more precision resolution, whereas the backward propagation needs a wider range.

Chmiel et al. [6] determined that input gradients calculated during backpropagation follow the log-normal distribution, whereas activation values in the forward path and weight gradients follow the normal distribution. Based on this observation, their work determines the best exponent-mantissa bit breakdown for a given total data size applied to backpropagation and successfully decreased input gradient data size to 6 bits.

Meanwhile, other work by Sun et al. [41] reduced the data size of the activation values to INT4 while using FP4 for the data representation of the gradient values. The work used FP16 for accumulation, and the data type used during the weight update is unclear.

Fixed-point training

Aside from low-precision floating-point training, other works have investigated the impact of replacing FP32 arithmetic with fixed-point datatypes [2, 14, 15, 37, 48, 52]. Fixed-point operators can be implemented more area- and power-efficiently, allowing us to pack more processing elements (PEs) into the same space (cost) to improve the overall throughput of the model.

Banner et al. [2] uses INT8 as a storage format for weights, activation values, and input gradients while maintaining INT16 for weight gradient representation and FP32 for weight update. That work uses Google's GEMMLOWP library which performs GEMM accumulation in INT32. Later work by Yang et al. [52] attempted INT16 accumulation, but it suffered from prediction accuracy degradation.

Recently, Fu et al. [14] proposed a method to adjust the data size of each layer at a different phase of training using Neural Architecture Search techniques. The work observed that the models learn coarse features at the beginning of the training and are more tolerant to perturbation; therefore, we can assign fewer bits at the initial phase of the training and gradually increase the data type size as training proceeds. Based on the observation, the work showed that gradually increasing data size from INT4 to INT8 for the forward propagation and from INT6 to INT8 for the backward propa-

gation can achieve FP32-comparable results, while the data type used for accumulation is not clearly stated.

Block floating-point training

The use of block floating-point format in DNN training is also actively studied [7, 11, 13, 24, 51]. Drumond et al. [11], and Das et al. [7] utilize a shared exponent of 8 bits and successfully reduced the mantissa to 8 bits and 16 bits, respectively, while maintaining FP32 comparable prediction accuracy. Drummond et al. [11] also observed that using 6-bit exponents incurs some accuracy loss compared to an FP32 baseline.

Flexpoint proposed by Köster et al. [24] employed a 5 bit shared exponent and showed that training converges well with a mantissa of 16 bits. SWALP [51] was motivated by weight averaging [20] and reached convergence with an 8 bit shared exponent and 8 bit mantissa.

Recently, Fox et al. [13] proposed a variant of block floating-point called Block Minifloat (BM); it uses low-precision floating-point data types for the mantissa representation of the block-floating point format. BM has a shared exponent bias within the same block, which scales the floating-point mantissa values similarly to the shared exponent in block floating-point. The work reduced the BM mantissa width to 6 bits while keeping FP32-comparable results, although the size of the shared bias is not clearly mentioned.

Some prior works pick a relatively long 8 bit mantissa empirically [11], or follow the result of preceding work [51] as their conclusion. However, it is important to consider very smaller mantissas because the area required

for a digital multiplier scales quadratically with the mantissa size.

Deployment to FPGA

The efficient implementation of quantized DNN models is also widely studied in the FPGA community [4, 30, 47, 50]. A whitepaper by Xilinx [50] and paper by Wu et al. [47] both combined floating-point multipliers and fixed-point accumulators for inference tasks. They determined that this combination has the potential to achieve a good prediction accuracy-resource utilization tradeoff. In this thesis, we consider applying the combination to DNN training tasks. In addition, this thesis has some important differences compared to the Xilinx work which consider training accuracy versus cost:

- Xilinx drops NaN, infinity and subnormal support which reduces accuracy; we retain NaN infinity, repurpose NaNs, and modify subnormal support to reduce hardware and limit any lost accuracy;
- Xilinx changes the exponent bias term to $2^{(x-1)}$ to avoid multiplication overflows by reducing the dynamic range with unknown accuracy impact; and
- Xilinx truncates the mantissa after multiplication to fit the desired storage format, which we show degrades accuracy during training.

Gupta et al. [15] deployed DNN training on an FPGA and demonstrated that FP32 data representations can be replaced by 16-bit fixed-point without much accuracy degradation, as long as the values are stochastically rounded. The advantage of 16-bit fixed-point is with a systolic array implementation.

Fox et al. [12] successfully trained VGG16 and PreResNet-20 on an embedded FPGA using the SWALP [51] algorithm. This work replaces the GEMM multipliers with 8-bit block floating-point operators, but still uses 32-bit integer accumulators; the impact of reducing accumulator width is not investigated. On the other hand, we modify both multiplier and accumulator sizes.

Low-precision training frameworks

As low-precision training becomes a popular research topic, frameworks supporting non-FP32 operation are being developed. QPyTorch [54] extends PyTorch [36] by adding quantizing modules before and after each computation module, such as convolutional layers and linear layers. The quanitizing modules simulate low-precision arithmetic by converting high-precision results into fewer bits. This allows quick testing of ideas, but internal operations within a composite operation are still in FP32 precision, which is unrealistic as it does not capture overflows of intermediate results. As a result, QPyTorch gives optimistic results. To properly capture the effect of low-precision arithmetic operations, we developed Archimedes-MPO.

Vink et al. proposed a framework called Barista [45], which extends the Caffe library [21] by adding FPGA acceleration support. One advantage of our work over the Barista paper is that we are using training results measured by using an actual FPGA kernel, whereas they use a performance model to report the same.

Chapter 3

Framework: Archimedes-MPO

In this section, we discuss the design of our framework, Archimedes-MPO (**M**ixed-**P**recision **O**peration), implemented for exploring mixed-precision training. Archimedes-MPO supports a range of CPU, GPU, and FPGA devices. The FPGA support is implemented to evaluate the resource utilization of the target low-precision kernel in detail so that we can understand the trade-off between resource-saving and prediction accuracy. However, we note that the purpose of the FPGA support is to study the trend in the trade-off; the implementation generated by the framework is not intended to be an accelerator for use in end applications, rather it is used as a research vehicle to explore various mixed-precision configurations.

On the other hand, the GPU support, which is bit-accurate with the FPGA version, is added to allow concurrent batch-mode training on a cluster of workstations. We have access to a compute farm having more than 100 GPU compatible nodes while there are only three FPGA boards available in the lab. Thus, GPUs allow us to explore the training convergence behavior of multiple configurations more quickly.

In both GPU and FPGA versions, we implement an accelerated GEMM kernel since these are the core operations in both convolution and fully connected layers. All other network components, including ReLU activation

Table 3.1: Major Modifications to Prior Arts

prior arts	modification	Before	After
TinyDNN [34]	Data Type	float/double	+LPFP,+FXP
	HW Support	CPU	+FPGA,+GPU
	Convolution	direct	`im2col/col2im`+GEMM
GEMM-HLS [8]	Data Type	float/double/uint8 etc.	+LPFP,+FXP
	Conversion	-	FP32 ⇔ LPFP/FXP
	Transpose	Support either T/non-T inputs	Support both T/non-T inputs

and pooling layers, are performed in software on the host CPU using FP32. We have limited our exploration to just the GEMM operations for expedience because they amount to most computations and are the most likely layer to exhibit precision problems.

3.1 CPU-side Implementation

For maximum flexibility, Archimedes-MPO uses a hybrid software/hardware approach that incorporates TinyDNN [34] as well as GPU- or FPGA-accelerated GEMM kernels. The FPGA-accelerated version, shown in Figure 3.1, uses PetaLinux, 4 ARM cores, and an HLS-based GEMM accelerator on a Xilinx MPSoC FPGA. The GPU-accelerated version uses standard Linux on x86 with NVIDIA GPUs. A summary of the changes that make up Archimedes-MPO is given in Table 3.1.

The TinyDNN framework is written in C++14 and consists of a header-only, dependency-free, templated and optimized implementation capable of multi-threaded SIMD execution on x86 and ARM. Table 3.1 summarizes the major modifications to TinyDNN. In detail, Archimedes-MPO modifies TinyDNN in three key ways:

- it adds more supported data types and fully customizes the data types

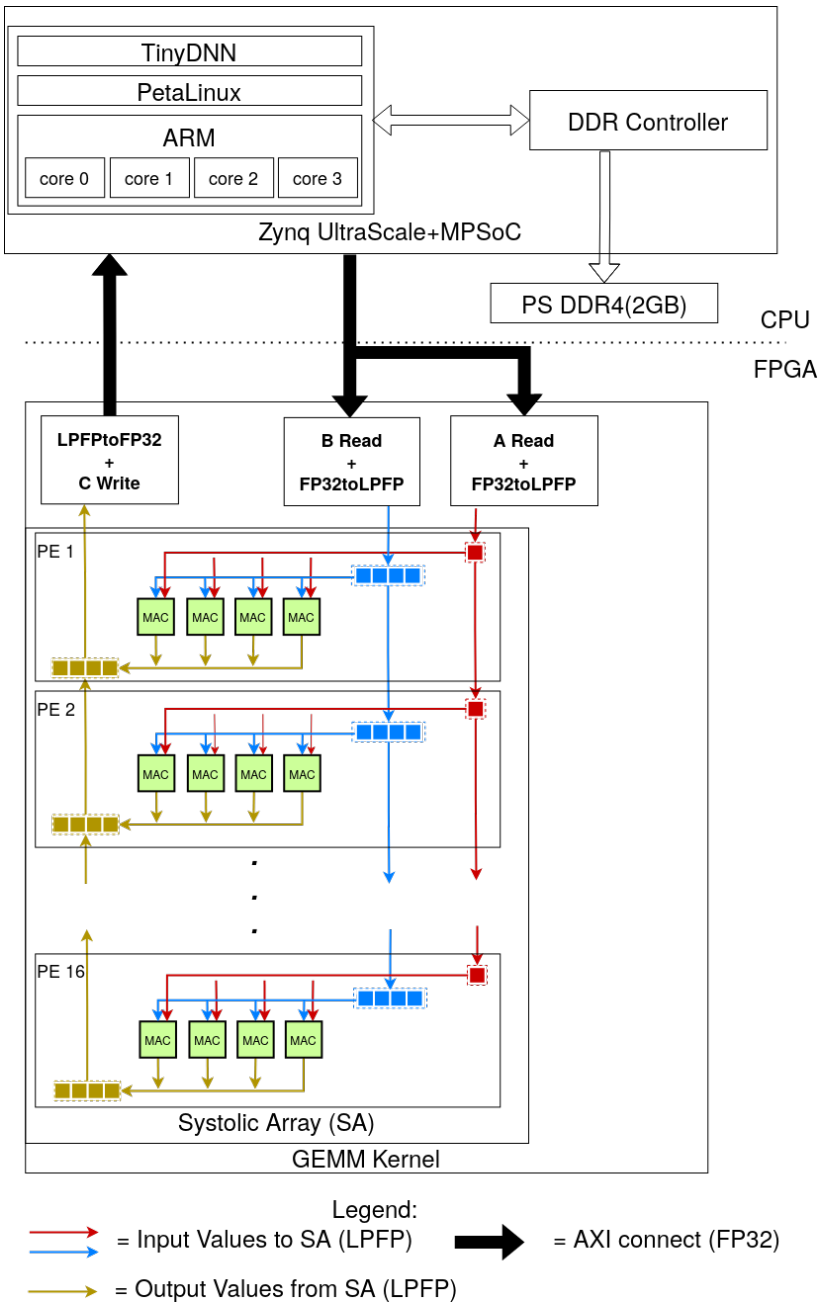


Figure 3.1: Archimedes-MPO Block Diagram for FPGAs

used for GEMM multiplication and GEMM accumulation of each layer (each computational kernel) with templates;

- it employs custom low-precision GPU or FPGA acceleration for the GEMM kernel; and
- it implements `col2im` and `im2col` transforms so it can offload convolutional layers for the FPGA GEMM kernel.

Since each layer is a C++ instance, this framework is capable of modelling networks with a different precision in each layer. In some layer types, such as fully-connected layers, different precisions are specified for multiplier inputs, multiplier outputs, and accumulation.

Using templates and overloading, Archimedes-MPO can create networks with a different precision or arithmetic type in each layer. However, all layer instances that use the FPGA accelerator hardware must use the same set of precisions because only a single hardware kernel is synthesized.

In this thesis, we investigate the utilization of FP32, low-precision floating-point, and fixed-point data types on GEMM computation and consistently use FP32 for other layers because the other layers are not computational bottlenecks. This partitioned strategy simplifies the FPGA implementation, since all data storage on the software side can use the FP32 format, and the hardware converts/quantizes to/from the desired format during data reading/writing. Although the data conversion could be done in software, it is much more efficiently done on-the-fly in FPGA logic. While a low-precision data storage format would improve memory footprint, bandwidth, and power, those are not important objectives for this thesis.

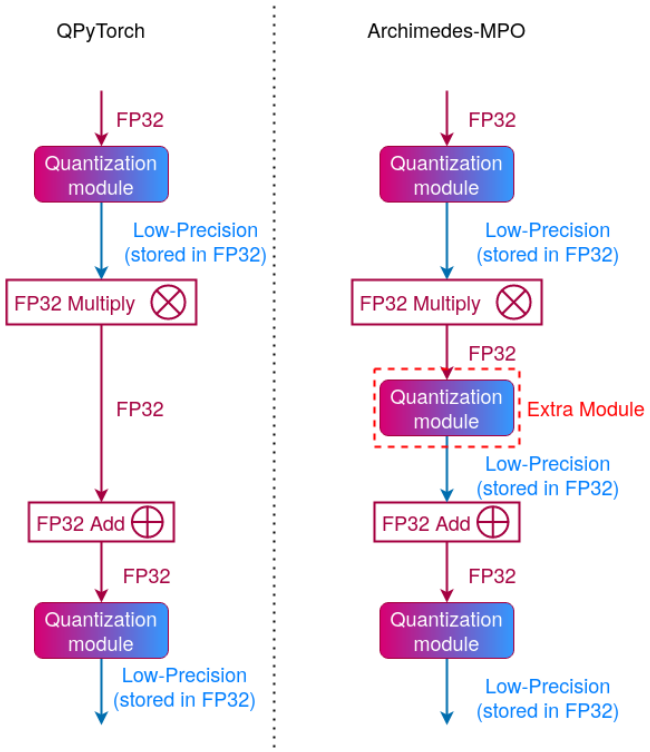


Figure 3.2: Low-Precision GEMM Computation using QPyTorch (left) and Archimedes-MPO (right)

Given that all operations performed in software remain in FP32, we note that the trained models generated have FP32 weight values, and that weight updates applied during training is done in FP32. If memory capacity or bandwidth are important, these are the next critical areas to consider for reduced precision.

3.2 GPU CUDA Implementation

The GPU version of Archimedes-MPO enables fast batch-mode training on x86/GPU servers running Linux. It also uses the same modified TinyDNN

as the FPGA implementation so it is easier to track changes between the two accelerator platforms, e.g., if we modify training strategies. However, it uses CUDA to implement the GEMM kernel by emulating low-precision arithmetic in a way that is bit-accurate with the FPGA accelerator. To decrease the overhead caused by data transfer between layers, all operations (other than the image data loading at the beginning of each iteration) are implemented as CUDA kernels.

Compared to QPyTorch [54], the GPU version of Archimedes-MPO inserts a quantization module before and after each operation as shown in Figure 3.2. The strength of Archimedes-MPO over QPyTorch is the quantization of both GEMM multiplication and accumulation. QPyTorch adds extra quantization modules but does not modify the core PyTorch GEMM routines; hence, it can only quantize the input and output values from the composite GEMM operation. Accordingly, QPyTorch can quantize the input to GEMM multiplication and the output from GEMM accumulator, but it cannot quantize the intermediate values between them.

The GPU experiments in this thesis are executed on a compute farm where each node runs CentOS Linux 7 on an Intel Xeon Silver 4116 processor. Each processor contains 12 cores (supporting 24 threads) and 4 NVIDIA Tesla V100 SXM2 GPUs (each containing 5,120 CUDA cores and 640 tensor cores). However, training uses only 1 GPU. The Archimedes-MPO GPU implementation is not as fast as QPyTorch, but it accurately emulates low-precision arithmetic for the entire GEMM operation.

3.3 FPGA Implementation

For the FPGA GEMM implementation, we customized an open source GEMM hardware kernel [8] written using Xilinx Vitis HLS. We used the Xilinx Vitis framework to provide PetaLinux on ARM as well as an FPGA logic shell called the Vitis Embedded Platform. A shell is predesigned logic that simplifies connection of custom accelerators by providing common services (typically memory access).

3.3.1 Target Platform

The platform used to evaluate Archimedes-MPO is a Xilinx ZCU104 development board. It contains a Zynq UltraScale+ MPSoC device (ZCU7EV) and one 2GB bank of 64-bit DDR4 soldered to the ARM hard processor system. The GEMM kernel is synthesized into the FPGA logic using Vitis HLS version 2020.2.

The software portion of our framework utilizes all four ARM Cortex-A53 cores running at 1.2GHz. Compiler auto-vectorization is enabled to use NEON instructions.

In this thesis, we follow the notation of [8], where a PE includes more than one multiply-accumulate (MAC) unit as shown in Figure 3.3. The FPGA GEMM kernel uses a linear array of 16 PEs, with each PE containing 4 MAC units, for a total of 64 parallel MAC units. Overall performance is impacted by clock speed and functional unit utilization caused by fragmentation and overhead when tiling the given network model onto the fixed-size PE array. While other configurations are possible and to achieve different

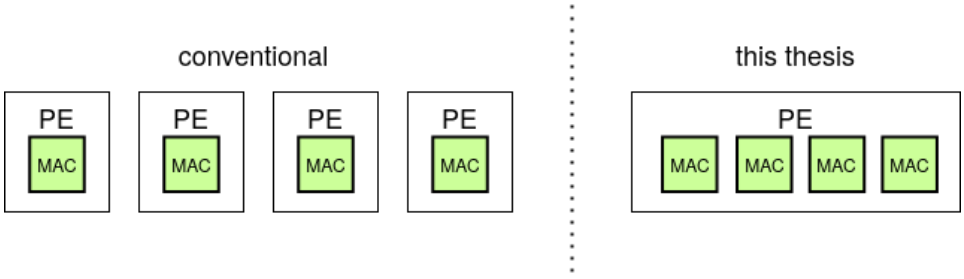


Figure 3.3: PE Definition in this Thesis

levels of performance, it is not the focus of this thesis.

3.3.2 Dataflow

Since the FPGA GEMM kernel handles all data conversions between FP32 and the low-precision format, data traversing the AXI4 Interconnect modules are always in FP32. The GEMM kernel has three ports: two for reading the input values corresponding to two input matrices, and a third for sending back computation results to DDR4. The modules for reading and writing the values in DDR4 are also in charge of data type conversion.

3.3.3 Modifications to GEMM Kernel

The lower rows of Table 3.1 summarizes the major modifications from the GEMM-HLS kernel [8]. The detail of each modification is provided as follows.

The original GEMM-HLS kernel supports only standard-width primitive data types (e.g., float and uint8). We modified it to support custom low-precision data types, such as FP8 with different exponent-mantissa breakdowns. In addition, we modified data reading/writing modules in the

GEMM kernel to convert GEMM input/output data between FP32 and low-precision. Allowing the input data to be FP32 reduces the software overhead of conversion and allows TinyDNN to work exclusively with FP32, keeping it simple and efficient.

We also modified the base GEMM-HLS kernel [8] to read both transposed ($k \times n$) and non-transposed ($n \times k$) matrix data. This does not affect Fmax, and the kernel throughput still sustains over 97% of peak GOPS when tested with sufficiently large (2048×2048) transposed and non-transposed matrices. This allowed us to remove the redundant input data transpose when performing back-propagation of convolutional and fully-connected layers.

Although there are plenty of possible GEMM implementation alternatives, we have not attempted to investigate this. The purpose of the thesis work is to study the trend in resource-accuracy tradeoffs at the MAC level.

3.3.4 Saturation

While converting FP32 values to lower precision floating-point or fixed-point data types, the converter might run into larger or smaller values than its representation range limit. The major approach to address the issue is saturating values that exceed a threshold to preset values; for example, the largest value the data type can express, or infinity.

Considering the compatibility with the adaptive loss-scaling algorithm, which needs to detect the overflow incurred in GEMM arithmetic, the data type converter saturates all values larger/smaller than the representation limit values to positive/negative infinity. If a value is saturated to infinity values at a certain point of GEMM arithmetic, all subsequent operators

taking the value as input will output infinity values.

Similar to the data type converter, multiplier and accumulator operators also saturate the output value to infinity when overflow occurs during computation.

3.3.5 Multiplier

We have implemented multipliers for low-precision floating-point data types and fixed-point data types to investigate the resource-accuracy trade-offs. Both input/output values are stored in `ap_uint` in floating-point multiplier, `ap_int` data type in fixed-point multiplier. These are the Vitis HLS data types that allow us to access each bit of the value separately.

Implementation

Conventionally, a floating-point product is computed by adding the exponents of two operands and multiplying their mantissa (Section 2.1), then the output mantissa is rounded and re-normalized. With large mantissa fields, the product is normally computed using multiplier blocks in the FPGA known as DSP blocks. Since we are working with small mantissa fields, we have coded the multiplication logic in C++ to use only LUTs. Since FPGAs typically use 6-input LUTs, a single LUT can compute a single output bit of a 3×3 multiplier. For example, in a Xilinx UltraScale+ VUP13 FPGA, there are 140 6-LUTs for every DSP block [50]; these LUTs are sufficient to implement 23 independent 3×3 multipliers. The implementation also provides optional support for input or output subnormals, rounding, renormalizing, NaN recoding, and infinity saturation.

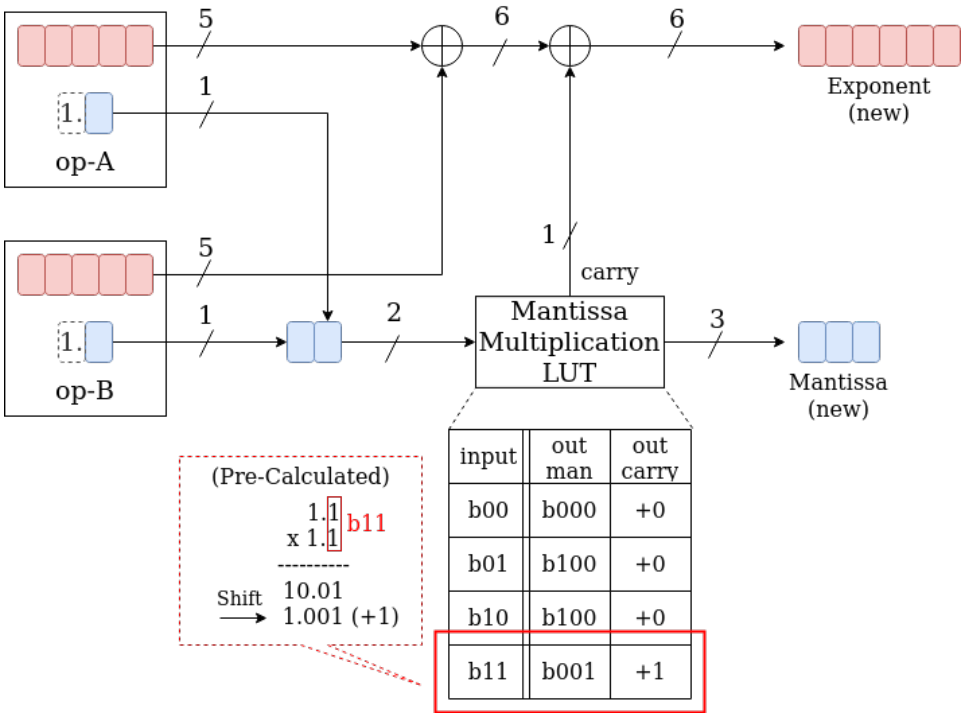


Figure 3.4: The implementation of floating-point multiplier using lookup table for mantissa multiplication

Figure 3.4 shows how an E5M1 product is computed without output rounding. It produces an exact E6M3 result which cannot overflow and suffers no rounding loss. (The bias term adjustment is omitted for clarity.)

We also implemented a fixed-point multiplier following normal conventions: the multiplier calculates the product of two integer values and rounds the result back to the input values' data type. This implementation is synthesized to use DSP blocks because the fixed-point values are much wider than the small floating-point mantissas.

Table 3.2: Best Resource Utilization of a Multiplier

precision	LUTs	FFs	DSPs
FP32 sub (no DSP)	987	185	0
FP32 sub	374	169	2
FP32 (no DSP)	573	658	0
FP32	102	188	2
FP16 (no DSP)	330	127	0
FP16	195	107	1
bfloat16 (no DSP)	291	117	0
bfloat16	180	122	1
E6M3	115	67	0
E5M3	86	55	0
E4M3	78	50	0
E6M2	68	55	0
E5M2	67	51	0
E4M2	65	52	0
E6M1	53	43	0
E5M1	47	39	0
E4M1	40	35	0
Q16.16	279	189	4
Q8.8	106	98	1
Q7.7	93	87	1
Q6.6	81	74	1

Resource Utilization

Table 3.2 shows the resources used by a single multiplier of different data types. We employ E4M1 to E6M3 for low-precision floating-point multiplier and Q6.6, Q7.7, Q8.8, and Q16.16 for the fixed-point multipliers. The format of the target Q-notation is empirically chosen based on the MNIST training accuracy (Section 4.1.2).

To see the reduction in resource utilization, Table 3.2 also provides the resource utilization of the higher precision multipliers: FP32, FP16, and bfloat16. Vitis HLS utilizes LogiCORE Floating-Point Operator IP [49] to

implement an FP32 multiplier by default; however, the IP core does not support subnormal inputs and outputs. To make the comparison fair, we have implemented and reported the resource utilization of a custom FP32 multiplier that supports subnormals (shown as `FP32 sub` in the table), given that low-precision multipliers also support them. Adding subnormal value support increases the LUT utilization of an FP32 multiplier from 102 to 374 when we allow the HLS compiler to use two DSP blocks; it increases from 573 to 987 for the implementation without DSP blocks.

Bfloat16 and FP16 have the same total length of 16 bits, but bfloat16 is implemented more efficiently. Both FP16 and bfloat16 multiplier use 1 DSP, but the FP16 multiplier requires 195 LUTs whereas bfloat16 requires 185 LUTs. The implementation forcing the FP16 multiplier to only use LUTs takes 330 LUTs, while the bfloat16 multiplier uses only 291 LUTs; bfloat16 still takes less resources. The difference in LUT count arises from the longer mantissa length in FP16. We will see the same trend with low-precision formats: it takes more LUTs to increase the mantissa by 1 bit than to increase the exponent by 1 bit.

The low-precision floating-point multipliers save considerable resources. Similar to the 16-bit multiplier comparison, the mantissa length more strongly affects area than the exponent length (e.g., E5M2 vs. E5M1 and E5M2 vs. E6M2). The result is reasonable because having one more mantissa bit for input values adds two input bits to the lookup table producing the product, producing a table 4 times larger. In contrast, one more bit in the exponent value merely increases the exponent adder width by one bit.

As shown in Table 3.2, the fixed-point multipliers tend to use more LUT

resources than ExMy multipliers, plus the fixed-point multipliers all need DSP blocks. Given that ExMy needs only 6 to 10 bits while Q$i.f$ needs 12 to 32 bits to represent values, it seems that Q$i.f$ multipliers will also need more system support for wider buffering (storage) and interconnect than ExMy. Given their promise, we will further consider ExMy multiplier variants in the next section.

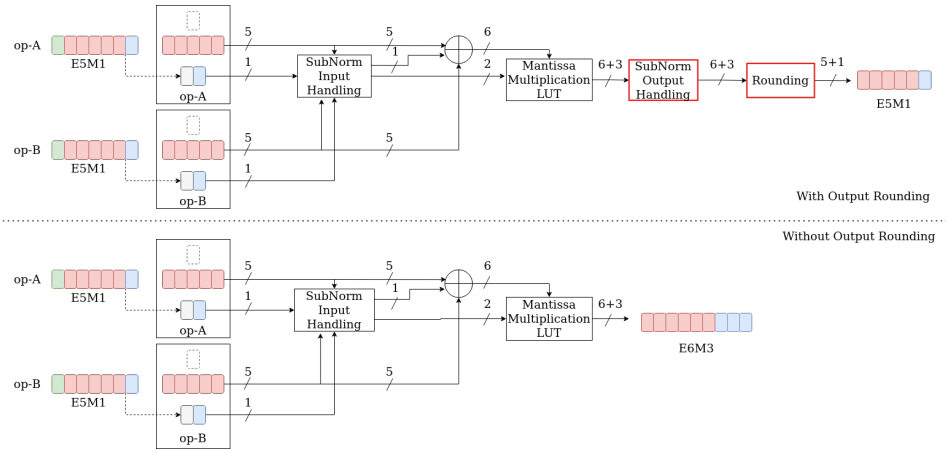


Figure 3.5: Multiplier with (upper) and without (lower) Post-Processing to Output Values.

3.3.6 Multiplier Variants

The default floating-point multiplier (`CFG-1`) discussed in the previous section has ancillary logic to handle exceptional values: subnormal input/output handling, output rounding, and NaN value handling as shown in Figure 3.5. In this section, we investigate the impact of removing them incrementally on resource utilization.

CFG-2: Removal of Subnormal Outputs

The first multiplier variation cuts off the subnormal support for multiplier output values. The modification removes the shifter used for handling the mantissa of the subnormal output values. It saves several LUTs, yet it leads to information loss as outputs that used to be conveyed as subnormal values are now all truncated to 0.

CFG-3: Removal of Output Rounding

The second multiplier variation does not round its output. As shown in Figure 3.5, the product of two input values, each with $y_i + 1$ mantissa bits after including the implicit bit, generates at most $2y_i + 2$ output bits. These must be rounded to the required output precision, which would normally be $y_o + 1$ bits including the implicit bit.

In the rounding logic, the lowest mantissa bits are removed, and any overflow saturates to the Infinity representation of the target precision. The rounding process causes information loss, but it also requires special logic resources for the additional shifting and rounding. Therefore, we hypothesize that the multiplier without any output rounding (Figure 3.5 lower) might improve both resource utilization and prediction accuracy at the same time.

A multiplier without the output rounding returns the full mantissa product directly as the multiplier's output. For example, if the multiplier takes two E4M1 (or E6M2) operands, the output would be E5M3 (or E7M5). As depicted in Figure 3.6, having one more bit in the exponent field doubles the representation range. Product values that were previously in the sub-

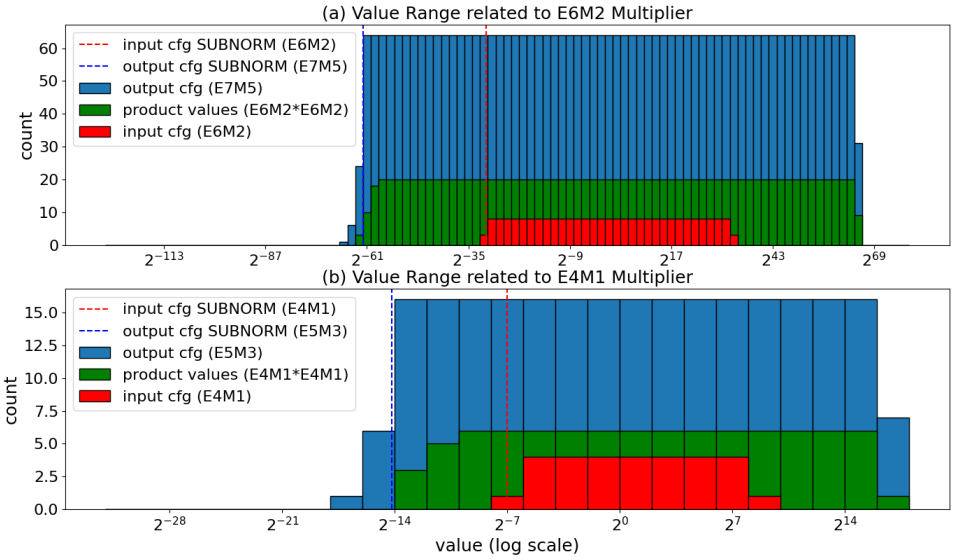


Figure 3.6: Representation range of the low-precision data type used for input (red) and output (blue) with no multiplier output rounding. The green values show the distribution of all product values that the multiplication can generate. The red dashed line indicates a boundary of the input data type: values left of the boundary are subnormals, while values to the right are normalized. The blue dashed line indicates the same boundary, but for the output data type of the no-rounding multiplier. The upper/lower subplot shows the representation range of the multiplier taking E4M1/E6M2 input values.

normal range of E4M1/E6M2 can be represented as normal values within E5M3/E7M5, which allows us to remove the post-processing logic regarding subnormal handling as well.

CFG-4: Removal of NaN Support

The third multiplier variation stops preserving NaN values because it wastes some encoding range. With a data type having a broad representation range (for example, FP32), the loss is not significant; however, when considering

Table 3.3: Conventional and Customized Encoding Table when Exponent is the Largest Value (e.g., E5M2)

mantissa	conventional	custom
max. normal value	57,344	57,344
b00	INF	65,536
b01	NaN	81,920
b10	NaN	98,304
b11	NaN	INF

low-precision data types such as E5M1, preserving NaN values can result in losing more than 1% of the encoding range.

In addition, in the GEMM computation used for DNN training, the importance of distinguishing NaN and infinity values is low. The condition to generate NaN values in GEMM multiplication and addition requires either of the operands to be infinity, which is good enough to flag an exceptional value and indicate that training is not converging. Based on this observation, we removed NaN support from the baseline multiplier. Instead, the multiplier can use the recovered encoding space to represent more large numbers.

By treating these as normalized values, the new encoding can be supported without any special additional logic. This requires remapping infinity to use a mantissa value of all ones, which is also more logical and consistent. Table 3.3 shows the expanded range this encoding provides for E5M2.

CFG-5: Alternative Subnormal Input Support

The fourth multiplier variation attempts to save more LUTs: rather than adding special logic to handle denormalized input mantissas when the exponent field $E = 0$, this variation treats all subnormals as though they are

Table 3.4: Conventional (left) and Customized (middle) Encoding Table when Exponent is the Smallest Value (e.g., E5M2). The Rightmost Column Shows the Case Where All Subnormal Values are Truncated Down to 0.

mantissa	conventional	custom	truncate
b00	0	0	0
b01	1.53E-5	3.81E-5	0
b10	3.05E-5	4.58E-5	0
b11	4.58E-5	5.34E-5	0

normalized, i.e., the implicit bit is still 1 rather than 0. This provides some consistency to the logic and preserves a small fraction of the subnormal input range.

As shown in Figure 3.7, a CFG-5 multiplier can still represent half of the conventional subnormal numbers that use the largest subnormal exponent value (e.g., $-(2^{5-1}-2)-1=-15$ for E5My) using the custom encoding. However, the multiplier loses other subnormal values with even smaller exponent values. Since the representation range of the subnormal values relies on the mantissa bit count, the encoding range loss is more severe when the mantissa is wider, e.g., consider E5M2 vs. E5M4 in Figure 3.7. Conversely, the range loss is less severe with the low-precision data types than FP32.

CFG-6: Removal of Subnormal Input Support

The fifth multiplier variation attempts to save more LUTs by removing all subnormals and truncating the values down to zero. This is expected to produce the most savings, but it will likely negatively impact training accuracy.

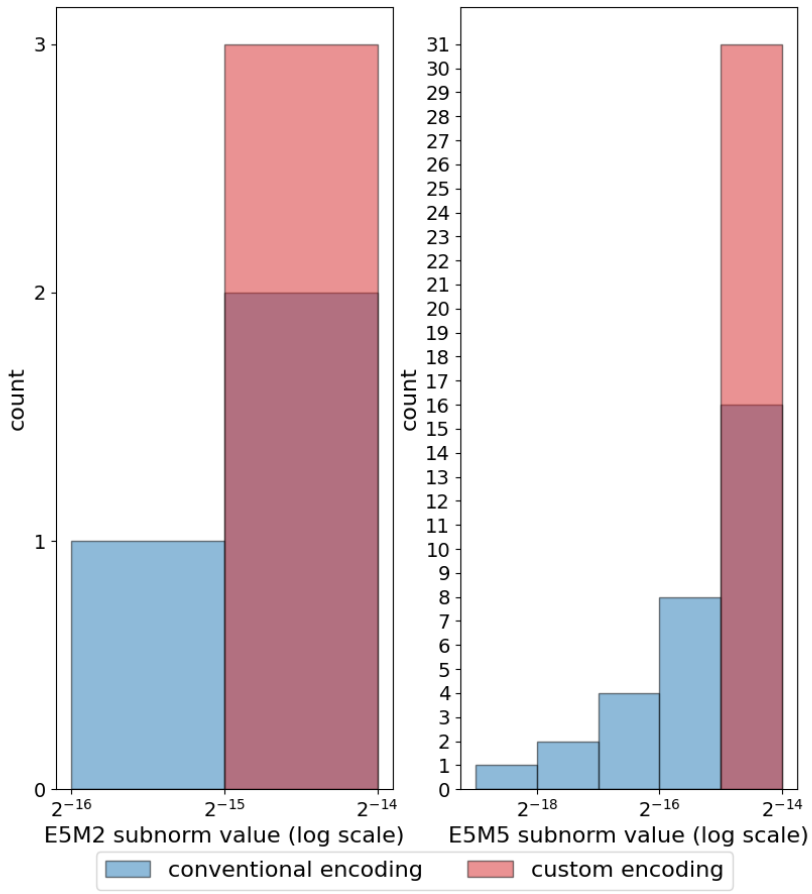


Figure 3.7: The figure shows the E5M2/E5M4 subnormal representation range of the conventional encoding (blue) and custom encoding (red).

Table 3.5: Exceptional Values Supported by Different Configurations of Low-Precision Floating-Point Multiplier

cfg name	Output SubNorm	Rounding	NaN/INF	Input SubNorm
`CFG-1`	YES	YES	NaN/INF	YES
`CFG-2`	NO	YES	NaN/INF	YES
`CFG-3`	NO	NO	NaN/INF	YES
`CFG-4`	NO	NO	INF only	YES
`CFG-5`	NO	NO	INF only	Custom
`CFG-6`	NO	NO	INF only	NO

Resource Utilization

In this subsection, we discuss the resource utilization of the different multiplier variants. Table 3.5 gives a summary of which configuration has support for which exceptional values. `CFG-1` in the first row indicates the default multiplier, which handles all subnormal input/output values and NaN values, and applies the rounding to its output values. The following configuration `CFG-2` removes the support for subnormal output values and truncates them to 0. Then the 3rd configuration (`CFG-3`) considers the additional impact of removing output rounding. We investigate the additional effect of removing NaN handling logic in `CFG-4`. Finally, `CFG-5` considers the customized encoding of subnormal inputs, while `CFG-6` truncates all the conventional subnormal input values to 0.

Figure 3.8 shows the LUT counts used to implement the 6 multiplier variants across multiple data types: E6M3, E5M3, E6M2, E5M2, E6M1, and E5M1. For all data types, the resource utilization of a single multiplier decreases from `CFG-1` to `CFG-6`, as expected.

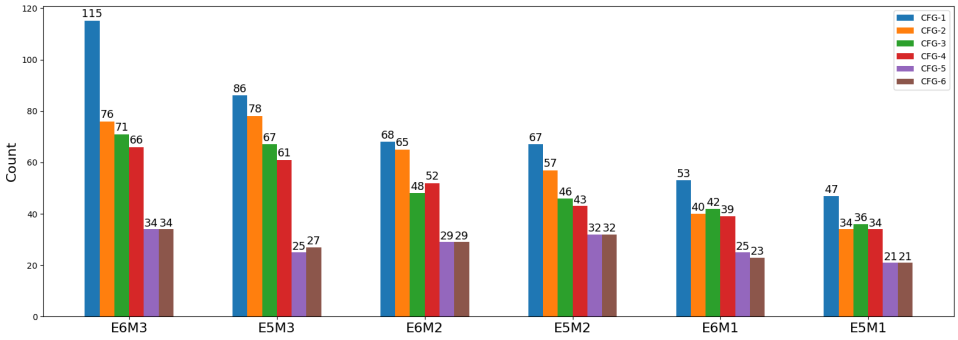


Figure 3.8: LUT Count of a Single Multiplier with Different Configurations

The first thing we notice is a considerable LUT count reduction when removing the logic for handling subnormal output values (i.e., `CFG-1` vs. `CFG-2`). The multipliers of all data types benefit from the removal, although the reduction amount differs among data types, with E6M3 saving the most.

Similarly, the reduction that arises from the removal of output rounding is shown when comparing `CFG-2` to `CFG-3`). Interestingly, when the multiplier benefits a lot from the removal of the subnormal output handling logic, for example saving more than 10 LUTs, its LUT count does not differ significantly between `CFG-2` and `CFG-3` (e.g., E6M3, E6M1, and E5M1). On the other hand, when the reduction amount by the first modification is not remarkable, the multipliers tend to benefit more from removing the output rounding logic (e.g., E5M3 and E6M2). In either way, multipliers having the same mantissa bits settle down to the similar LUT count after removing both subnormal outputs and the rounding. We believe this variation is caused by the sometimes inconsistent success of HLS synthesis optimizations.

In contrast, removing the extra shifters required by subnormal input values from `CFG-4` to `CFG-5` consistently decreases the LUT count by 10 to

13. As this removes a barrel shifter, it is reasonable to observe the significant drop in LUT count. The multiplier with our custom subnormal encoding (`CFG-5`) requires the same LUT count as `CFG-6`. Since `CFG-5` makes use of these encodings for essentially no cost, and expect it to provide higher training accuracy than `CFG-6`, it is the most promising multiplier variant.

3.3.7 Accumulator

Similar to the multiplier investigation, we have studied the resource utilization of an accumulator of low-precision floating-point and fixed-point data types. Both floating-point/fixed-point accumulators are implemented following the explanation in Section 2.1, with additional implementation details provided below.

As briefly explained in the background section, the core steps of a floating-point adder are: (1) align the mantissa of the value with a smaller exponent value, (2) apply addition or subtraction based on the sign of two operands, (3) re-normalize, and (4) apply final rounding.

The first step, alignment, forces the adder to lose information conveyed by the shifted mantissa bits. The adder employs a guard bit, round bit, and sticky bit to mitigate the information loss. The sticky bit is set to 1 if any of the shifted out bits passing the sticky bit position has a value of 1, so the adder needs to track the value of the shifted portion of the mantissa. Accordingly, the adder assigns $(1 + \texttt{MAN\_BITS} * 2 + 3)$ bits for the alignment input: 1 bit for the implicit bit, `MAN_BITS` for the original mantissa value, 3 bits for the extra bits, and another `MAN_BITS` for tracking the shifted mantissa value. For example, the adder targeting E6M3 input values allocates

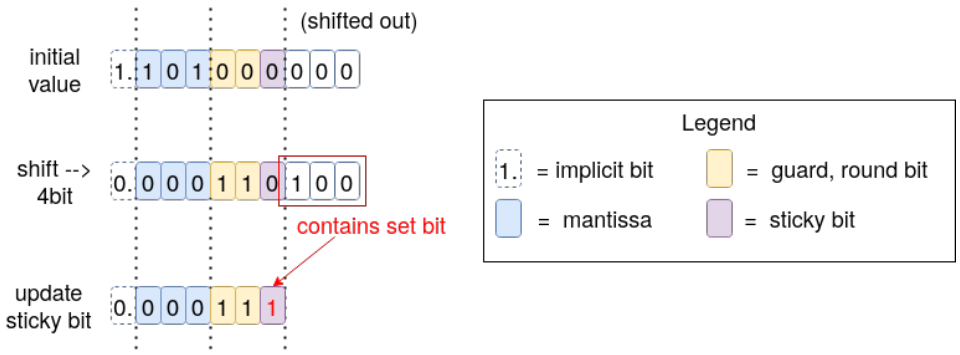


Figure 3.9: FP Accumulator Alignment and Extra Bits

$1 + 3 + 3 + 3 = 10$ bits for the alignment, and based on the shifted-out portion of the mantissa, it updates the sticky bit as shown in Figure 3.9. Besides the shifter used for the alignment itself, the first step includes other operators, which are three adders of size `EXP_BITS`. One of those adders is used for computing the shift count needed for the alignment by taking the difference between exponent values of the two operands. The other two adders are used for updating the exponent values of the two operand values, respectively, if the corresponding value is in subnormal range.

Following the alignment, the second step applies mantissa addition or subtraction of `MAN_BITS` $+ 1 + 3$ bits based on the sign of the two operands; if both operands have the same sign, the adder adds up two mantissa values, otherwise it subtracts the smaller mantissa from the larger mantissa.

As the third step, the adder re-normalizes the summation/subtraction result, but thisp requires the adder to calculate the shift count. To obtain the shift count, the `__builtin_clz` HLS function locates the leftmost set bit and calculates the difference of the index from the supposed decimal point position with a 6-bit subtractor. The re-normalization itself requires

a `MAN_BITS` + 1 + 3 + 1-bit shifter, and the exponent update in accordance with the shift requires another adder of `EXP_BITS` bits.

The fourth and final step applies rounding to the re-normalized value. Considering the carry during the rounding, the computation takes the concatenation of the new exponent and mantissa values as the input, so the addition takes `MAN_BITS` + 1 + 3 + 1 bits.

In summary, a floating-point adder taking ExMy input values contains:

- $(1 + y * 2 + 3)$-bit shifter
- four x-bit adders
- $y + 1 + 3$-bit adder/subtractor
- `__builtin_clz`
- 6-bit subtractor
- $(y + 1 + 3 + 1)$-bit shifter

It is possible that the adder is suboptimal in its logic design and HLS writing style, but we leave the further adder optimization to future work.

Resource Utilization

Table 3.6 shows the resources utilized by an accumulator of different data types. For floating-point accumulators, we also show the expected input type for the floating-point multipliers with or without intermediate rounding. A range of E4M1 to E7M7 accumulators covers all possible output data types of floating-point multipliers listed in Table 3.2.

Table 3.6: Resource Utilization of an Accumulator

FP-mult configuration	acc dtype	LUTs	FFs	DSPs
FP32	FP32	189	403	2
E6M3(w/o round)	E7M7	255	200	0
E6M2(w/o round)	E7M5	185	171	0
E6M1(w/o round)	E7M3	187	154	0
E5M3(w/o round)	E6M7	242	191	0
E5M2(w/o round)	E6M5	187	174	0
E6M3(w/ round) E5M1(w/o round)	E6M3	165	145	0
E6M2(w/ round)	E6M2	136	139	0
E6M1(w/ round)	E6M1	138	121	0
E4M3(w/o round)	E5M7	222	180	0
E4M2(w/o round)	E5M5	172	165	0
E5M3(w/ round) E4M1(w/o round)	E5M3	148	136	0
E5M2(w/ round)	E5M2	121	127	0
E5M1(w/ round)	E5M1	113	110	0
E4M3(w/ round)	E4M3	121	127	0
E4M2(w/ round)	E4M2	116	117	0
E4M1(w/ round)	E4M1	84	105	0
-	Q16.16	52	0	0
-	Q8.8	34	0	0
-	Q7.7	25	0	0
-	Q6.6	26	0	0

Table 3.7: LUT Count of E6M2 Accumulator when Using/Not Using Selected Single-Line If Statements

default	w/o the if statements
136	77

The fixed-point adder uses fewer LUTs and FFs than FP32 and even E5My accumulators. The major reason for the difference is that the fixed-point accumulator does not require any input/output alignments before/after the summation nor does it require output rounding.

The floating-point accumulator uses considerable area. To investigate whether there may be opportunity for optimization in its design, we removed seemingly simple steps in the accumulator that each require a single-line "if" statement in HLS: a compare-and-swap of input operands to find the smaller exponent, and exceptional value checking of the accumulators inputs and outputs. Table 3.7 shows the LUT savings of an E6M2 accumulator before and after this optimization. While the optimization investigated here renders the accumulator to generate incorrect results, we hope this motivates further research into possible accumulator optimizations.

Based on the current LUT count of multipliers and accumulators, we hypothesize that simultaneously using a low-precision multiplier and a fixed-point accumulator might produce the lowest LUT count.

Data Type Conversion from FP to FXP

To employ a floating-point multiplier and fixed-point accumulator at the same time, the multiplier output in floating-point data format needs to be converted to fixed-point data format before the value is fed into a subsequent accumulator. Although this is a simple table lookup operation when used with narrow multiplier outputs, the table size doubles for each additional output bit in the multiplier; the extra cost is considerable for the variants without output rounding. Instead, we implemented a logic-based

Table 3.8: Resource Utilization of FP(E4M1 to E7M7) to FXP(Q8.13) Value Converter

input data type	output data type	LUTs	FFs
E7M7	Q8.13	116	42
E7M5		103	40
E7M3		72	38
E6M7		97	41
E6M5		81	39
E6M3		67	37
E6M2		56	36
E6M1		55	35
E5M7		94	40
E5M5		76	38
E5M3		61	36
E5M2		55	35
E5M1		56	34
E4M3		47	0
E4M2		44	0
E4M1		42	0

data conversion module.

Table 3.8 shows the LUT and FF utilization of the data conversion module. We assume all possible output data types of the floating-point multiplier listed in Table 3.2 as the input of the data type converter, and always set the output fixed-point data type to Q8.13 based on the training accuracy results shown in Section 4.2.2. As shown in the table, with shorter input types, the data type converter is smaller than the Q8.13 accumulator (i.e., 55 LUTs), but it can grow to approximately twice the accumulator size for the largest input data types. Nevertheless, the sum of the converter and fixed-point accumulator areas are still smaller than the corresponding floating-point accumulator. For example, E6M5 accumulator takes 187 LUTs whereas the

Table 3.9: PE-level Resource Utilization of Floating-Point Accumulator and Data Converter+Fixed-Point Accumulator

mult input dtype	cfg name	mult output dtype	acc dtype	LUTs	FFs
E5M2	`CFG-1`	E5M2	E5M2	1,564	1,762
			Q8.13	1,639	1,633
	`CFG-5`	E6M5	E6M5	1,679	1,932
			Q8.13	1,654	1,646

combination of the E6M5 to Q8.13 converter and the Q8.13 accumualtor takes $81 + 55 = 136$ LUTs in total. There are potentially more optimal implementations for the converter as well, but based on these observations we consider the use of a fixed-point accumulator with a floating-point multiplier to be a good combination for area.

3.3.8 PE-level Resource Utilization

This section considers logic usage of an entire PE with four MACs. For these experiments, we use E5M2 for multiplier inputs, two multiplier variants `CFG-1` and `CFG-5`, and two accumulator data types for each multiplier variant based on the best accuracy configurations reported in Section 4.2.2.

The results are shown in Table 3.9. They indicate that the fixed-point accumulators are roughly the same total area in LUTs as the floating-pont accumulators, which is unexpected. Upon further inspection, we realize the larger output width of the fixed-point accumulators requires additional support logic for routing, multiplexing and buffering results. This suggests it is also important to consider the effects on area of downstream logic.

Table 3.10: Archimedes-MPO Resource Utilization with 16 × 4 PE Array (Zynq UltraScale+ ZU7EV). The Default Multiplier (`CFG-5`) is Used for All Implementations.

precision	LUTs	FFs	BRAMs	DSPs
Device	230,400	460,800	312	1,728
Vitis shell				
FP32	22,308	30,044	41	3
Archimedes-MPO GEMM kernel (excluding Vitis shell)				
FP32	58,420	102,908	23	320
E6M3 + E7M7	46,153	59,633	23	0
E5M3 + E6M7	44,754	58,321	23	0
E4M3 + E5M7	42,761	56,583	23	0
E6M2 + E7M5	43,074	57,761	23	0
E5M2 + E6M5	42,640	56,925	23	0
E4M2 + E5M5	41,111	55,530	23	0
E6M1 + E7M3	40,937	53,680	23	0
E5M1 + E6M3	39,091	52,582	23	0
E4M1 + E5M3	37,416	51,290	23	0
E5M2 + Q8.13	43,371	55,209	23	0
E5M1 + Q16.16	48,583	59,627	23	0
E5M1 + Q8.8	37,383	50,240	23	0
E5M1 + Q7.7	35,154	48,102	23	0
E5M1 + Q6.6	34,636	47,818	23	0

3.3.9 System-level Resource Utilization

This section considers logic usage of the entire GEMM accelerator. Table 3.10 shows the resource utilization for different data types; it also shows the total device capacity and the overhead of the Vitis shell. The systolic array size is set to 16 PEs, each with 4 MACs, in all implementations. These are synthesized for the Xilinx ZCU104 board with a target clock frequency of 280 MHz. Following the results in Section 3.3.6, we selected `CFG-5` for all low-precision floating-point multipliers discussed in this section.

The low-precision implementations reduce all LUT, FF, and DSP utilization compared to the FP32 baseline. In particular, the DSP count falls from 320 to 0 when we replace the FP32 operators with the low-precision floating-point datatypes. This is expected, as the FP32 version uses 5 DSPs per MAC (3 for the multiplier and 2 for the adder).

The LUT count also decreases monotonically as the data size reduces for both floating-point and fixed-point data types. The area of floating-point implementations is more sensitive to mantissa size than exponent size (e.g., E5M1 vs. E6M1 and E5M1 vs. E5M2). This trend is also observed in the resource utilization per multiplier (Table 3.2), suggesting that reducing mantissa bits is key to reducing LUT count.

The mixed-precision implementations, which use a floating-point multiplier and a fixed-point accumulator simultaneously, sometimes achieve lower resource utilization than the floating-point-only counterparts. Due to the resource-efficient accumulator, the mixed-precision implementations consume fewer resources than floating-point counterparts when the fixed-point data length is not too long (e.g., E5M1 vs. E5M1 + Q7.7/Q6.6). However, the benefit cancels out with the larger internal FIFO and other ancillary modules when the fixed-point data length increases further (e.g., E5M1 + Q16.16).

BRAM count is fixed at 23 for all implementations since those BRAM are only used for implementing global AXI buffers. All internal FIFOs and storage are implemented with LUTs and FFs.

3.4 Summary

This section has introduced our framework Archimedes-MPO, and the implementation details of its GEMM accelerator. In particular, we explored the implementation of resource-efficient low-precision floating-point multipliers by removing exceptional value support, and a mixed-precision approach that employs a floating-point multiplier with a fixed-point accumulator.

Figure 3.8 provides a summary of the multiplier investigation. It shows that customizing number representations used in the low-precision floating-point multiplier is beneficial to saving LUTs; the IEEE-754 standard requires multipliers to handle exceptional values such as subnormals, NaN, and infinity as well as the proper rounding of the output. However, our custom multiplier eliminated the subnormal outputs, NaNs and rounding, and used custom encoding for subnormal inputs and infinity to save area.

Table 3.9 indicates that PEs using a mixed-precision floating-point multiplier and fixed-point accumulator simultaneously can be just as efficient as using only with floating-point. However, the wider fixed-point accumulator output can enlarge downstream logic. As a result, the floating-point only implementations achieve lower LUT counts at the system level (Table 3.10).

There is a large design space worthwhile to investigate regarding the implementation GEMM systolic array, floating-point accumulator, and data type conversion between a floating-point multiplier and fixed-point accumulator. The PE-level and operator-level trends in resource utilization we observed in the section would be useful to help guide such a design.

Chapter 4

Training Results

In this chapter, we employ the Archimedes-MPO framework from the previous chapter and evaluate training with different low-precision configurations. The GEMM computation in these experiments will consider a floating-point multiplier and fixed-point or floating-point accumulator. The investigation aims to observe the trade-offs between resource utilization and prediction accuracy among GEMM operator data type variations. We start the investigation with the training of small models that target a small dataset, MNIST [27], and then explore the viability of the trend in larger models targeting larger datasets such as CIFAR-10 [25]. The MNIST experiments were all run on the FPGA board, but evaluation of larger models (e.g., ResNet20 and VGG16 targeting the CIFAR-10 dataset) is done using GPU acceleration.

4.1 Experiment 1: MNIST

This section explores the MNIST dataset [27] using MLP and LeNet5 models. Training was performed with Archimedes-MPO using FPGA acceleration.

Table 4.1: Network Structure Used for MNIST and CIFAR-10 Testing

layer name		kernel size	stride	output channel	output size
LeNet5 (MNIST) input size: (32×32×1, padded with 0s)					
conv1		5	1	6	(28×28)
conv2		5	1	16	(10×10)
fc1		-	-	128	-
fc2		-	-	84	-
fc3		-	-	10	-
MLP (MNIST) input size: $28 \times 28 \times 1 = 784$					
fc1		-	-	128	-
fc2		-	-	96	-
fc3		-	-	10	-
ResNet20 (CIFAR10) input size: $(32 \times 32 \times 3)$					
conv0		3	1	16	(32×32)
conv1-block	c1-x-1	3	1	16	(32×32)
(×3)	c1-x-2	3	1	16	(32×32)
	c2-1-1	3	2	32	(16×16)
conv2-1-block	c2-1-2	3	1	32	(16×16)
	c2-1-d	3	2	32	(16×16)
conv2-block	c2-x-1	3	1	32	(16×16)
(×2)	c2-x-2	3	1	32	(16×16)
	c3-1-1	3	2	64	(8×8)
conv3-1-block	c3-1-2	3	1	64	(8×8)
	c3-1-d	3	2	64	(8×8)
conv3-block	c3-x-1	3	1	64	(8×8)
(×2)	c3-x-2	3	1	64	(8×8)
fc		-	-	10	-
VGG16 (CIFAR10) input size: $(32 \times 32 \times 3)$					
conv1-x (×2)		3	1	64	(32×32)
conv2-x (×2)		3	1	128	(16×16)
conv3-x (×3)		3	1	256	(8×8)
conv4-x (×3)		3	1	512	(4×4)
conv5-x (×3)		3	1	512	(2×2)
fc1		-	-	512	-
fc2		-	-	512	-
fc3		-	-	10	-

Table 4.2: Network Structure of ResNet50

layer name		kernel size	stride	output channel	output size
ResNet50 (Imagewoof) input size: $(224 \times 224 \times 3)$					
conv0		7	2	64	(112×112)
conv1-1-block	c1-1-1	1	1	64	(56×56)
	c1-1-2	3	1	64	(56×56)
	c1-1-3	1	1	256	(56×56)
	c1-1-d	1	1	256	(56×56)
conv1-block ($\times 2$)	c1-x-1	1	1	64	(56×56)
	c1-x-2	3	1	64	(56×56)
	c1-x-3	1	1	256	(56×56)
conv2-1-block	c2-1-1	1	1	128	(56×56)
	c2-1-2	3	2	128	(28×28)
	c2-1-3	1	1	512	(28×16)
	c2-1-d	1	2	512	(28×28)
conv2-block ($\times 3$)	c2-x-1	1	1	128	(28×28)
	c2-x-2	3	1	128	(28×28)
	c2-x-3	1	1	512	(28×28)
conv3-1-block	c3-1-1	1	1	256	(28×28)
	c3-1-2	3	2	256	(14×14)
	c3-1-3	1	1	1024	(14×14)
	c3-1-d	1	2	1024	(14×14)
conv3-block ($\times 5$)	c3-x-1	1	1	256	(14×14)
	c3-x-2	3	1	256	(14×14)
	c3-x-3	1	1	1024	(14×14)
conv4-1-block	c4-1-1	1	1	512	(14×14)
	c4-1-2	3	2	512	(7×7)
	c4-1-3	1	1	2048	(7×7)
	c4-1-d	1	2	2048	(7×7)
conv4-block ($\times 2$)	c4-x-1	1	1	512	(7×7)
	c4-x-2	3	1	512	(7×7)
	c4-x-3	1	1	2048	(7×7)
fc		-	-	10	-

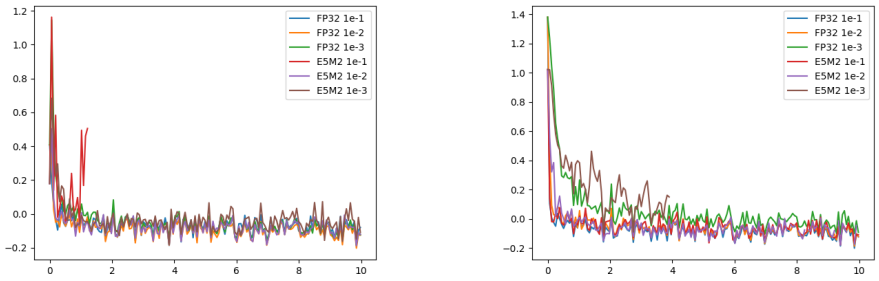
4.1.1 Network Models and Training Settings

Training with MNIST uses the MLP and LeNet5 [27] network models described at the top of Table 4.1. The MLP model contains three fully connected layers, while LeNet5 contains 2 convolution layers and 3 subsequent fully connected layers. Each convolutional layer is followed by ReLU activation and max-pooling layers; similarly, a ReLU activation appears at the output of all fully connected layers except the last. We employ the cross-entropy loss function to calculate loss values for both models.

Stochastic-gradient descent with a momentum value ($\mu = 0.9$) is used for all experiments. As for the learning rate, we have investigated the training stability of LeNet5 and MLP models in FP32 and E5M2 using different learning rate values: 0.1, 0.01, and 0.001, as shown in Figure 4.1. Given that training loss using a learning rate of 0.01 stably and quickly decreases with both data types, we decided to use a learning rate of 0.01 for both models. Similarly, we stopped training both models after the 10th epoch because the loss values plateau after the 2nd epoch.

The MNIST image data is of `uint8` data type, ranging from 0 to 255. It is re-scaled by dividing the original pixel values by 255, and stored as FP32, so that the input values are in the range of 0 to 1. Note that information stored in the least significant bits will be lost when the GEMM kernel converts these input values to low-precision floating-point in the first convolutional layer.

The weights of the LeNet5 model are initialized following the implementation of `kaiming_normal_` function in PyTorch [36], and the weights in the MLP model are initialized following `xavier_uniform_` initialization function



(a) LeNet5 (b) MLP

Figure 4.1: Training loss of FP32/E5M2 LeNet5 and MLP models with different learning rate. Each data point is sampled after every 50 batch-iteration.

of the same library. We divided the 60,000 MNIST training set images into 50,000 images for training and 10,000 images for validation. The batch size is set to 64.

4.1.2 Accuracy

Table 4.3 shows the final validation/test accuracy of LeNet5/MLP training using different GEMM multiplier and accumulator data type combinations. The leftmost column indicates the datatype(s) used for the multiplier and accumulator: the row with a single data type on the leftmost column shows indicates the same data type for both multiplier and accumulator, whereas the rows with $x + y$ notation indicates using data type x for the multiplier and data type y for the accumulator.

We evaluated training using FP32, E6M3, E6M2, E6M1, E5M3, E5M2, E5M1, and bfloat16 for floating-point data types, and Q16.16, Q8.8, Q7.7, and Q6.6 for fixed-point data types. Inspired by [50], we have also investi-

Table 4.3: MNIST Validation/Test Accuracy (%), Best After 10 Epochs, All Using `CFG-1` Multiplier

	QPytorch		Archimedes-MPO			
	LeNet5	MLP	LeNet5		MLP	
precision	test	test	valid	test	valid	test
FP32	98.8	97.7	98.7	98.8	97.5	97.7
E6M3	99.1	97.9	98.5	98.5	95.3	95.3
E5M3	99.0	97.8	98.0	97.9	96.0	96.1
E6M2	98.7	97.5	96.4	96.8	93.5	93.5
E5M2	98.6	97.8	97.2	97.5	94.3	94.2
E6M1	98.6	96.7	**11.4**	**9.8**	**10.2**	**9.8**
E5M1	98.1	97.2	**10.2**	**9.8**	**10.2**	**9.8**
Q16.16	99.0	98.0	98.8	98.8	97.5	97.6
Q8.8	98.9	97.9	98.6	98.7	97.1	97.5
Q7.7	99.1	98.0	98.5	98.6	97.2	97.4
Q6.6	11.4	9.8	11.4	11.3	10.9	10.9
bfloat16	98.9	97.8	98.6	98.7	97.4	97.6
E5M1 + E5M2			95.7	95.6	94.6	94.7
E5M1 + Q16.16	*not*		98.3	98.4	97.1	97.1
E5M1 + Q8.8			98.3	98.5	96.9	97.0
E5M1 + Q7.7	*supported*		98.3	98.5	96.7	97.0
E5M1 + Q6.6			11.4	11.4	96.7	96.7

gated the performance of mixed-format training with different combinations; we employ the E5M1 data type for the multiplier and consider accumulator types of E5M2/Q16.16/Q8.8/Q7.7/Q6.6, respectively.

The second and third left columns show the test accuracy achieved by applying the same data type configuration in QPytorch [54], and the right-most four columns show the training results sampled using our framework, Archimedes-MPO. Since QPytorch performs all GEMM calculations using FP32 precision, we cannot evaluate the performance of mixed-precision training with QPyTorch; the corresponding rows are filled with "*not sup-*

ported". Due to the FP32 accumulation, the final prediction accuracy achieved by QPyTorch training is higher than that of Archimedes-MPO in all configurations except the FP32 baseline. In particular, the ExM1 precisions show excellent accuracy with QPyTorch but fail to converge with Archimedes-MPO. This underscores the importance of adding exact MPO support to the training framework even for small network models that are easy to train.

Based on the results, it is clear that the LeNet5/MLP training requires a low-precision floating-point accumulator to use at least 2 bits for the mantissa field. The configurations using ExM2 or ExM3 accumulator still converge, although we observe some accuracy degradation compared to the QPyTorch and FP32 results.

On the other hand, fixed-point low-precision data type training converges well with lower-precision, where we can decrease the bit-width till Q7.7 without a remarkable accuracy drop compared to FP32 results. Unfortunately, training with the Q6.6 accumulator causes the network to diverge completely, which was also predicted by QPyTorch. As already shown in other studies [23] bfloat16 achieves results very close to FP32 baseline; note that our work use bfloat16 on accumulation as well, whereas prior work uses FP32 accumulation.

Both LeNet5 and MLP training converge for all mixed-format combinations except LeNet5 training using the E5M1 multiplier with the Q6.6 accumulator. The fixed-point accumulator performs better than the E5M2 counterparts because the fixed-point accumulators never suffer from the information loss caused by the alignment shifting during summation.

Table 4.4: Average Training Time per Epoch of Archimedes-MPO (second)

	ARM + FPGA		ARM only	
precision	LeNet5	MLP	LeNet5	MLP
FP32	110.6	9.5	325.2	74.2
E6M2	112.5	9.9	1051.8	262.1
E5M2	112.4	9.7	1057.5	265.9
E6M1	112.0	9.6	1087.3	267.9
E5M1	112.0	9.6	1088.7	255.4

The mixed-format training using E5M1+Q6.6 diverges for the LeNet5 model, but it converges for the MLP model. The difference is most likely caused by the difference in accumulation length required by each model. For the LeNet5 model, accumulators need to be capable of representing the result of at most 64 * 16 * 16 = 16,384 consecutive accumulations during the weight gradient computation, whereas the maximum accumulation length of the MLP model is only 784.

4.1.3 Runtime

This section considers the impact of FPGA acceleration on low-precision training. While acceleration isn't the goal, we wish to validate that the bulk of computation is in the GEMM stages, and we wish to show that exact low-precision modelling suffers a slowdown in a pure CPU implementation, but does not suffer any slowdown with an FPGA implementation.

Overall Training Time

Table 4.4 summarizes the average runtime elapsed for a single epoch during LeNet5 and MLP training. Training data is preloaded from a microSD card

on the ZCU104 board, and this loading time is excluded from overall run-time calculations. We have measured the runtime of the training using five different data types (FP32, E6M2, E5M2, E6M1, and E5M1) in two different implementations: the one using only ARM (no FPGA acceleration) and the other using the combined ARM + FPGA-based GEMM kernel. With both network models, the ARM-only runtime of low-precision training increases by a factor of three. In contrast, the ARM+FPGA runtime shows a negligible increase in run-time. The large increase in the ARM-only runtime is due to the specialized code required to implement low-precision arithmetic correctly using 32-bit CPU instructions. In the ARM+FPGA implementation, these steps are efficiently implemented in the kernel such that there is no runtime overhead. These results confirm that an FPGA kernel allows us to execute low-precision (and full-precision) training more quickly on an embedded device.

Despite the successful speedup, there is a bottleneck to resolve in future work. The speedup of ARM + FPGA versus ARM-only is lower for LeNet5 training (around 9.4×), compared to the MLP training (around 27.4×). The gap in speedup is caused by the convolution-specific data transform functions: `col2im` and `im2col`. Neither of them has been FPGA-accelerated yet in our framework. Hence, the framework first transforms the data using the ARM CPU before the accelerated GEMM kernel. The transformed data is larger than the original data and incurs significant overhead in memory footprint and bandwidth. A similar issue is observed in prior art as well [12]. We suggest moving these functions to the FPGA kernel to reduce this overhead in future work.

Table 4.5: Layer-Wise Breakdown of E5M2 LeNet5 Training Runtime Average per Batch-Iteration (s)

	ARM only(s)		ARM + FPGA(s)		speedup	
layer	fwd	bwd	fwd	bwd	fwd	bwd
whole	1057.5		112.4		9.39	
conv1	1.12e-1	2.48e-1	2.31e-2	5.11e-2	4.85	4.85
conv2	1.71e-1	4.0e-1	9.46e-3	2.07e-2	18.12	19.32
fc1	3.20e-2	8.08e-2	1.76e-3	1.84e-3	18.17	43.91
fc2	7.74e-3	1.80e-2	6.93e-4	9.18e-4	11.16	19.64
fc3	1.07e-3	1.94e-3	4.84e-4	7.58e-4	2.22	2.57

Layer-wise Runtime Breakdown

In this section, we compare the layer-wise runtime of ARM-only and ARM + FPGA implementations. Fig.4.2 and Table 4.5 show the runtime breakdown of the computationally intensive layers during E5M2 training on the LeNet5 model. The runtime speedup of convolution layers is relatively moderate compared to the fully-connected layers; this difference is due to the aforementioned overhead of `col2im` and `im2col` transformations, which are not yet FPGA accelerated.

The last fully-connected layer does not benefit much from GEMM acceleration because of fragmentation and under-utilized PEs; the required GEMM dimensions are realtively small compared to the tile size used in the GEMM kernel. In contrast, the GEMM kernel achieved 43 times speedup in backward propagation of the 1st fully-connected layer. This is because it does not suffer from any data transform overhead or PE underutilization.

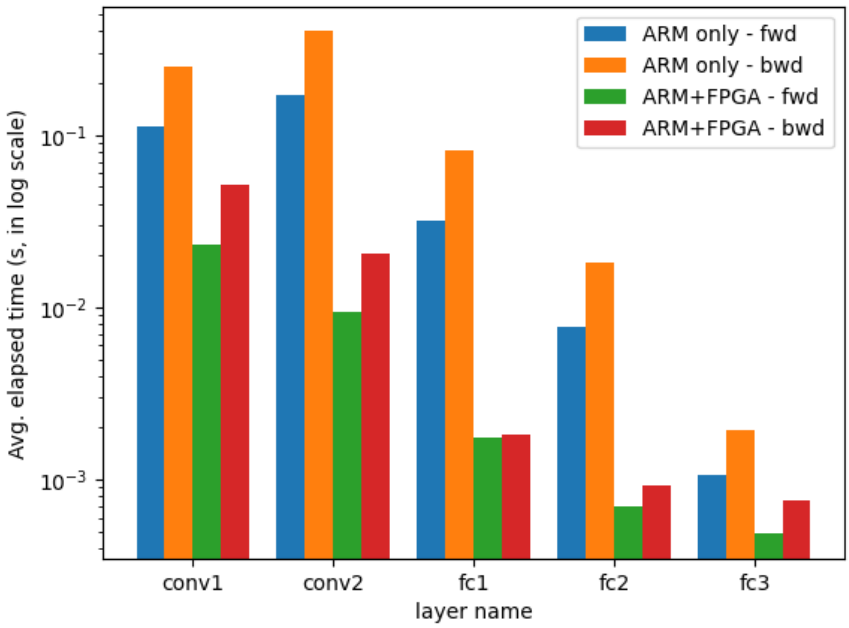


Figure 4.2: The layer-wise runtime breakdown of the computationally heavy layers (convolution and fully-connected) of LeNet5. The x-axis shows the layer name, and the y-axis shows the average elapsed timed of each layer. For ease of comparison, the y-axis is plotted in a logarithmic scale.

Summary

The purpose of this subsection is to verify that the acceleration by the FPGA GEMM kernel is performing as expected. As shown in Table 4.4, using FPGA acceleration in low-precision training leads to a negligible increase in runtime, whereas the same ARM-only implementation suffers from the overhead of emulating low-precision arithmetic. The FPGA acceleration option speeds up DNN training as expected and verifies the validity of the approach. Hence, runtime will be omitted in subsequent experiments.

4.2 Experiment 2: CIFAR-10 and Imagewoof

This section primarily explores the CIFAR-10 dataset [25] using VGG16 and ResNet20 models. Also, to test the viability of the observed trends in larger datasets, we occasionally use a subset of ImageNet called Imagewoof [18] with ResNet50 model (Table 4.2).

The larger dataset, Imagewoof, takes 10 dog-breed classes from the 1,000 classes of ImageNet. Although the number of classes is the same as the CIFAR-10, it is harder to generalize models to the dataset for three reasons. First, the input image size enlarges to $(224 \times 224 \times 3)$, and the target model, ResNet50, becomes deeper and larger as well. Second, the classes comprising CIFAR-10 are completely different objects (e.g., dog vs. truck). However, the images contained in the Imagewoof are all dogs of different breeds (e.g., Beagle vs. Golden Retriever); therefore, the model trained for the dataset is required to distinguish more subtle differences than the one trained for CIFAR-10. Third, the training/validation split of Imagewoof is set to 70/30 and some images that were originally used in the training set of ImageNet are used in validation set instead.

Training results were run on the GPU version of Archimedes-MPO. We always report the validation/test accuracy for the comparison of results, not the accuracy obtained during training.

4.2.1 Network Models and Training Settings

Following the first results on the MNIST dataset, we tested low-precision training with the CIFAR-10 dataset. Since the images of CIFAR-10 have

a larger dimension ($32 \times 32 \times 3$) compared to MNIST ($28 \times 28 \times 1$), the models used for the experiments are also replaced with larger and deeper ones: ResNet20 [17] and VGG16 [39]. As shown in Table 4.1, ResNet20 consists of 19 convolution layers and a single fully-connected layer, whereas VGG16 consists of 13 convolution layers and 3 fully-connected layers. In both models, each convolution layer is followed by a batch-normalization layer [19] and a ReLU layer.

Similar to the MNIST experiments, we have used stochastic-gradient descent with a momentum value of 0.9 for all CIFAR-10 experiments. The initial learning rate and weight decay are set to 0.1/0.01 and 0.0001/5e-4 for ResNet20 and VGG16 training, respectively, following the experimental settings of the original papers. All experiments employ the cosine annealing scheduler to decay the learning rate as the training proceeds. The batch size is always set to 128, and we have trained the ResNet20 model for 165 epochs and the VGG16 model for 200 epochs; both epoch counts are calculated based on the training iteration count reported in the original papers.

All Imagewoof experiments also used stochastic-gradient descent with a momentum value of 0.9 and initial learning rate is set to 0.01. The batch size is set to 64 and the ResNet50 model is trained for 100 epochs. It is common to apply data augmentation techniques such as random resizing and random cropping to ImageNet image data [17]. However, in this thesis we omit this to mitigate the runtime overhead. Since the purpose of the work is to study the trend in resource-accuracy tradeoff when changing the data type of GEMM operators to low-precision, removing data augmentation does not compromise the validity of the experiments.

As for image-preprocessing, we re-scaled the original pixel values to FP32 in the same manner as the MNIST image data, and normalized the values by subtracting the mean values of the CIFAR-10/ImageNet dataset and dividing it by the standard deviation. Similar to the MNIST experiments, also note that the information stored in least significant bits are lost when the GEMM kernel converts the input values to low-precision floating-point in the first convolutional layer. The work by Wang et al. [46] observed that this information loss leads to an accuracy drop on ImageNet training, and instead adopted FP16 for the first convolutional layer.

4.2.2 Accuracy

To investigate the limitation of low-precision training in detail, we evaluated the impact of replacing the FP32 GEMM multiplier and accumulator with the low-precision alternatives separately. Initially, the data type of either multiplier or accumulator is replaced with low-precision, while the other operator remains in FP32. After investigating the limitation of each operator, we employ the low-precision data types for both GEMM multiplier and accumulator at the same time and investigate the accuracy impact further.

Multiplier Investigation

First, we trained ResNet20 model by changing the GEMM multiplier data type to ExMy, where x ranges from 4 to 8 and y ranges from 1 to 7. Hence, the shortest data type in the experiments is E4M1, whereas the longest data type is E8M7. The left plot of Figure 4.3 shows the best test accuracy (across all epochs) achieved by each data type configuration. The results of

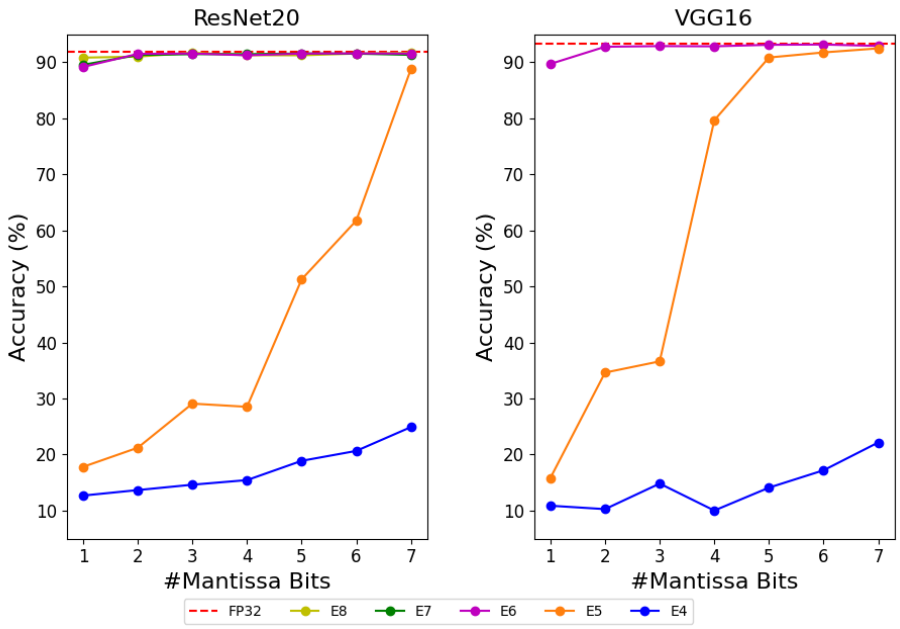


Figure 4.3: Best test accuracy achieved by different low-precision floating point data types (E4M1 to E8M7 for ResNet20, and E4M1 to E6M7 for VGG16) with training two different models (left: ResNet20, right: VGG16). The x-axis indicates the mantissa bit count and y-axis indicates the accuracy in percentage. The yellow/green/pink/orange/blue line shows the best prediction accuracy of E8My/E7My/E6My/E5My/E4My, respectively. The dashed red line shows the best accuracy achieved by FP32 baseline training.

data-types sharing the same exponent bit count are shown using the same trend line. The training using E8My/E7My/E6My achieves FP32 baseline comparable results in ResNet20 model. In contrast, the training with E5My or E4My GEMM multiplier fails to reach a similar baseline accuracy.

Following the investigation on ResNet20 model, we conducted the same investigation on VGG16 by changing x from 4 to 6 and y from 1 to 7; since we had already observed that the training using multipliers with more than 5 exponent bits converges to FP32-equivalent results, we decreased the max-

imum x value from 8 to 6. As depicted in the right plot of Figure 4.3, similar to the ResNet20 investigation, training using E6My multipliers achieves a test accuracy close to FP32 baseline, whereas the training with E5My or E4My GEMM multiplier fails to reach FP32-comparable accuracy.

Based on the results, we applied adaptive loss-scaling techniques [35] to training with E5My and E4My multipliers. As shown in Figure 4.4, the method contributes to saving gradient values from truncation, and the training using E5My and E4My (except E4M2 and E4M1) successfully attains an accuracy close to the FP32 baseline. The E5M2 multiplier with adaptive loss scaling is almost as good as FP32, and E5M1 with just 1 mantissa bit is good as well.

We also conducted the same multiplier investigation on the Imagewoof dataset on ResNet50. As depicted in Figure 4.5, training using E6My and E5My multipliers achieves FP32 comparable results, whereas training with E4My multiplier suffer from the severe accuracy loss. Replacing FP32 multipliers with E4My multipliers significantly degrades the test accuracy, which is unlike the ResNet20/VGG16 results on CIFAR-10 which tolerated E4My provided $y \geq 2$. However, the trend that training behavior is primarily affected by exponent field width, and training converges as long as multipliers have enough representation range is unchanged. Therefore, we conjecture that there is likely no trend which is specific to Imagewoof, or the full ImageNet, and use CIFAR-10 dataset for the rest of the investigation to expedite data collection. However, we will revisit Imagewoof again at the end.

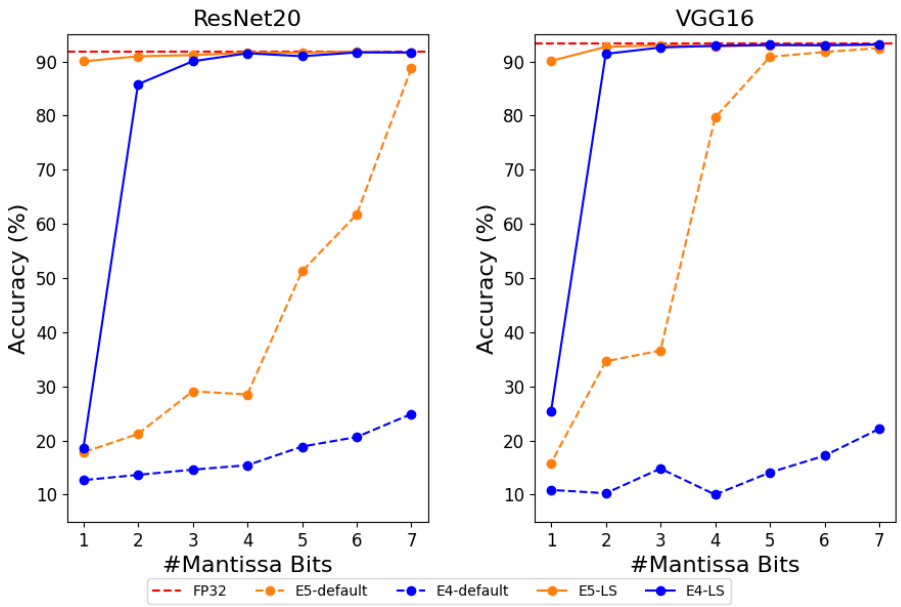


Figure 4.4: The graph shows the best test accuracy achieved with/without loss-scaling technique training on two models (left: ResNet20, right: VGG16). Similarly to the previous figure, the dashed red line shows the FP32 baseline result. The dashed orange/blue lines indicate the best accuracy achieved by E5My/E4My training without loss scaling, and the solid orange/blue lines indicate the results of the same configuration, but with loss scaling.

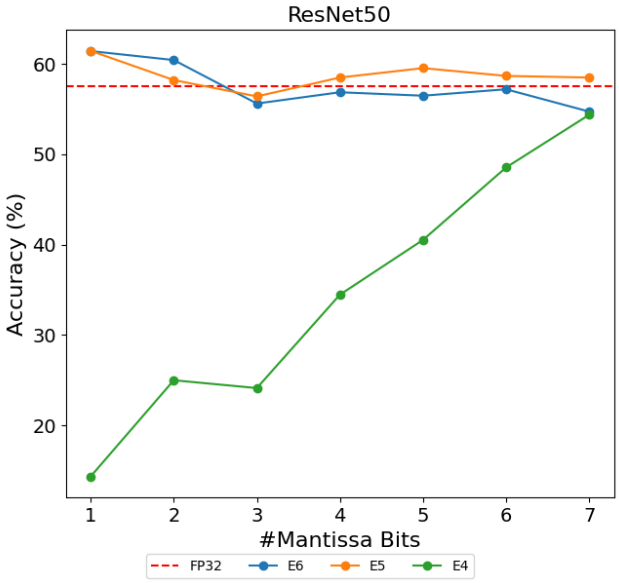


Figure 4.5: The graph shows the best test accuracy achieved by the ResNet50 model using E6My/E5My/E4My multiplier for Imagewoof dataset training with adaptive loss scaling. The blue/orange/green line shows the training results using E6My/E5My/E4My multiplier, respectively.

Multiplier Variants Investigation

After the testing with the default multipliers, we have investigated the impact of using custom multipliers which remove the exceptional-value support so that we can decrease the LUT count further, as discussed in Section 3.3.6. To recap briefly, the default multiplier which handles all input/output subnormal values, output rounding, and NaN values is denoted as `CFG-1`. Removal of subnormal output handling, output rounding, NaN support, and subnormal input support incrementally are denoted as `CFG-2`, `CFG-3`, `CFG-4`, `CFG-5`, and `CFG-6`, respectively (see Table 3.5).

In the first series of experiments, we compared the impact of subnormal output support on accuracy (`CFG-1` vs `CFG-2`). As shown in Figure 4.6, the accuracy of E5My stably reaches the FP32 baseline results regardless of the presence/absence of subnormal output logic; the E5My multiplier output values do not significantly reside within the subnormal range for both ResNet20 and VGG16 training. On the other hand, the E4My training results are significantly degraded after removing subnormal output support.

To regain the accuracy of E4My training while keeping subnormal output logic removed, the next series of experiments compares the multiplier variant with removed subnormal output logic (`CFG-2`) to the one that removes both the subnormal output and rounding logic (`CFG-3`). Figure 4.7 shows that removal of rounding (which requires assigning one more bit to exponent and `MAN_BITS+1` more bits to mantissa) prevents results that were in subnormal range from being truncated down to 0 (Figure 3.6) and pushes the E4My results back to being comparable with FP32.

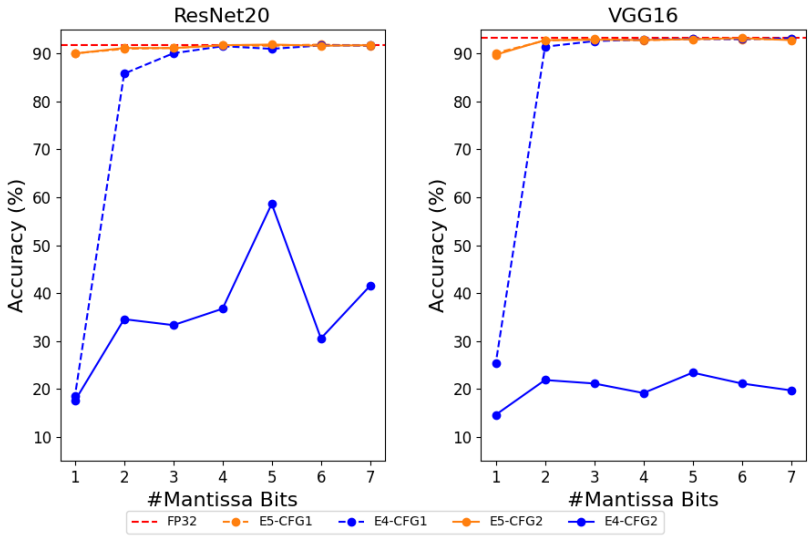


Figure 4.6: The graph shows the test accuracy of the training using E5My and E4My `CFG-1` multiplier (dashed line) and `CFG-2` multiplier (solid line) variants. The loss-scaling technique is employed in all experiments.

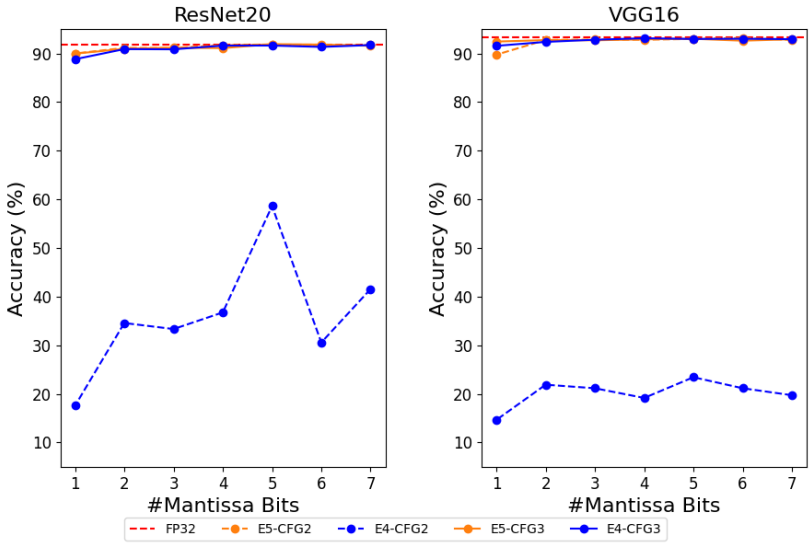


Figure 4.7: Test accuracy of the training using E5My and E4My `CFG-2` multiplier (dashed line) and `CFG-3` multiplier (solid line).

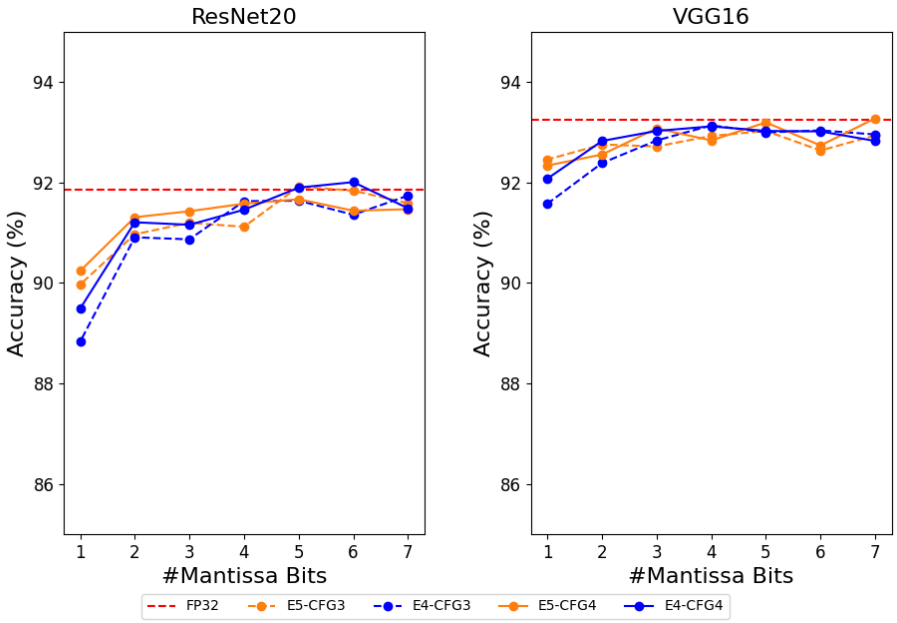


Figure 4.8: Test accuracy of the training using E5My and E4My `CFG-3` multiplier (dashed line) and `CFG-4` multiplier (solid line).

In the next experiment, we have explored removing NaN value support to assign more encoding range to usable values. As shown in Figure 4.8, there is a slight increase in accuracy, with E4My appearing to benefit more than E5My. This is reasonable because the narrow exponent range benefits more from the large values made available after NaN removal. Also, if the configuration has more mantissa bits, the more encoding values are reclaimed. Hence, NaN removal is beneficial to low-precision training.

Next, we have investigated the impact of removing subnormal value support in the multiplier inputs. As described in section 3.3.6, we applied custom subnormal encoding to input values of the multiplier (`CFG-5`) and compared with a variant having conventional subnormal support (`CFG-4`)

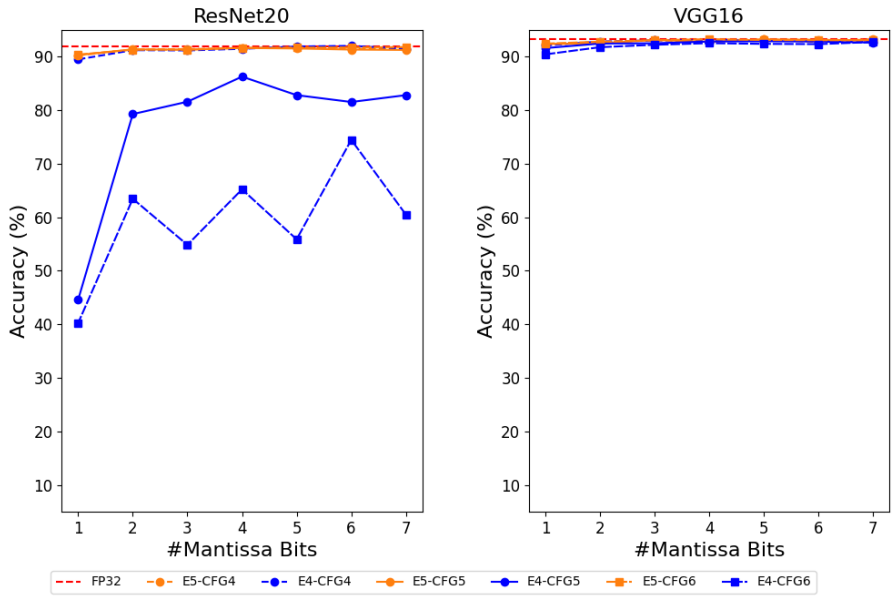


Figure 4.9: Test accuracy of the training using E5My and E4My `CFG-4` (dashed line, circle marker), `CFG-5` (solid line), and `CFG-6` multiplier (dashed line, square marker).

and a variant that truncates all subnormal input values to 0 (`CFG-6`).

As depicted in Figure 4.9, the results using E5My multipliers are not affected by the changes in either model. On the other hand, the E4My training results using different methods to handle subnormal inputs behave differently on ResNet20 and VGG16. While E4My with VGG16 attains FP32-comparable results regardless of the multiplier configuration, the removal of subnormal input handling logic on ResNet20 significantly damages the accuracy. The proposed custom encoding of subnormal numbers also suffers from the accuracy degradation of around 10% compared to the conventional one (i.e., `CFG-4` vs. `CFG-5`). However, it is much better than truncating all subnormal values to 0 (CFG-6). Recall the LUT count for

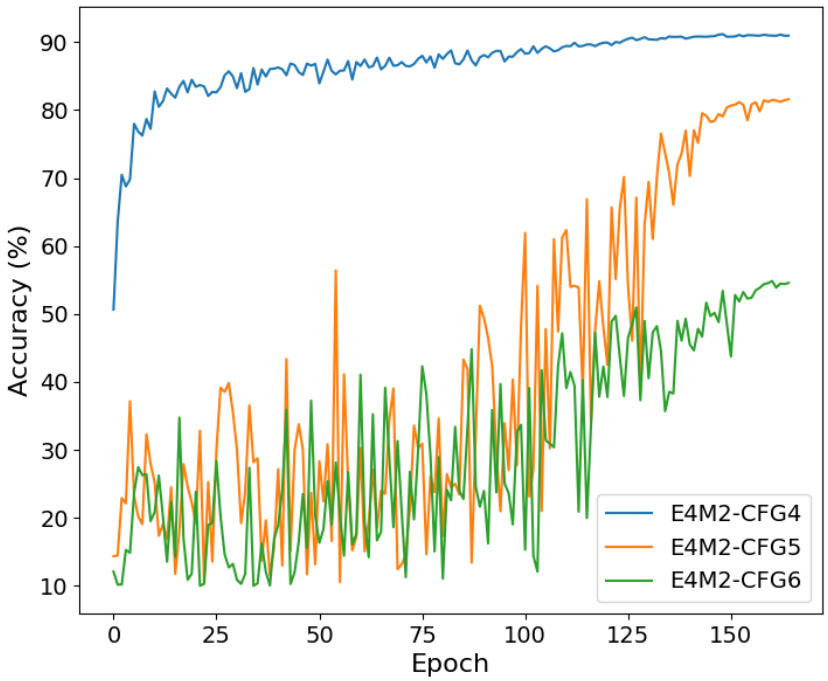


Figure 4.10: Transient of test accuracy when using E4M2 multiplier of different configurations on ResNet20: The blue/orange/green line shows the results of `CFG-4`/`CFG-5`/`CFG-6` multiplier respectively.

`CFG-5` and `CFG-6` are similar and much lower than `CFG-4` (see Figure 3.8).

To explore the potential of the custom subnormal encoding method, we have dug into the cause of the accuracy drop of `CFG-5` multiplier compared to `CFG-4` multiplier. Figure 4.10 shows the transient of test accuracy of E4M2 multiplier variants on ResNet20. As shown, the accuracy of `CFG-5` (in orange) and `CFG-6` multiplier (in green) monotonically increases; however, the accuracy value fluctuates significantly at the beginning of the training and stabilizes around the 150th epoch. Since we have replaced the default

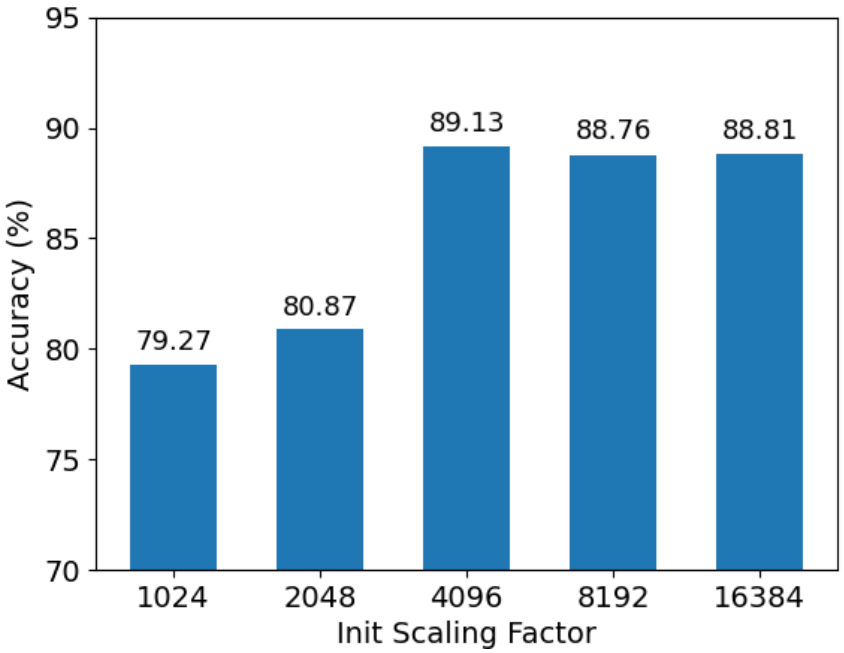


Figure 4.11: The best test accuracy achieved by E4M2 `CFG-5` multiplier when applying different initial loss-scaling factor.

subnormal encoding with a custom one for multiplier **inputs**, the possible cause of the degradation is either the activation, weight, or gradient values.

Considering the trend that the training stabilizes as the training proceeds, it is possible that the initial loss-scaling hyper-parameter tuned earlier is a slow-starter and impacts the initial phase of the training. Based on the observation, we increased the initial loss-scaling factor from 1024 to 2048, 4096, 8192, and 16384 (Figure 4.11). The best test accuracy of E4M2 `CFG-5` multiplier improved from 79.29% to 89.13% by changing the initial loss-scaling factor to 4096.

Figure 4.12 shows the best results when using the E4My multiplier with

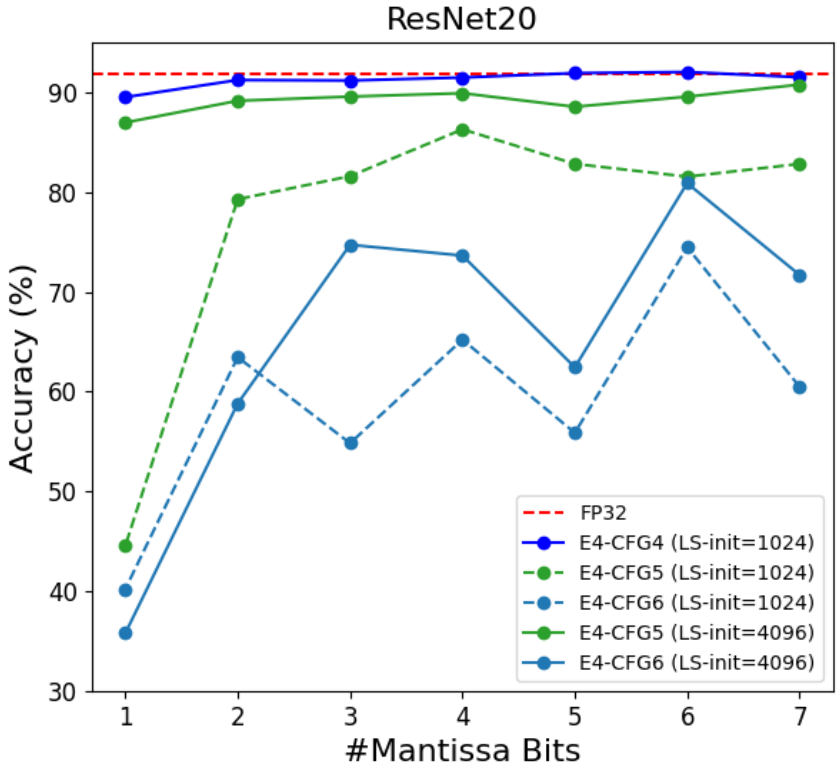


Figure 4.12: The best test accuracy achieved by E4My multiplier with different configurations and different initial loss-scaling factor. All the experiments are targeting ResNet20 model and the dark blue/green/light blue indicates the results of multiplier of `CFG-4`/`CFG-5`/`CFG-6`. The results with dashed/solid line are obtained by using initial loss-scaling factor of 1024/4096 respectively.

different configurations but setting the initial loss-scaling factor to 4096 instead of 1024. As shown, after resolving the issue of the slow-starter loss-scaling factor, the E4My multiplier with custom subnormal encoding (`CFG-5`) regains accuracy close to that of conventional subnormal encoding (`CFG-4`). Given that the multiplier with completely removed subnormal support (`CFG-6`) still suffers from the accuracy drop, the proposed custom encoding is effective.

Accumulator Investigation

In the subsequent experiments, we have investigated the impact of replacing FP32 accumulators with low-precision operators. Similar to the previous multiplier investigations, each experiment changes the data type used for GEMM accumulation, but the GEMM multiplier still uses FP32. The experiments change accumulators to ExMy floating-point accumulators, where y ranges from 1 to 7. The x ranges from 4 to 7 for both ResNet20 and VGG16. In addition to the floating-point accumulator, we also consider a fixed-point accumulator: the integer bit counts are always fixed to 8, and the fraction bit count is changed from 8 to 23.

Figure 4.13 shows the best prediction accuracy achieved by each floating-point data type accumulator. The E7y/E6y accumulators with more than a 4-bit mantissa stably achieve FP32-comparable accuracy in both models, and the accuracy starts to drop when the mantissa bit count becomes 4. The degradation becomes notable as the mantissa bits decrease further. On the other hand, E5My accumulators are less accurate than E6My while having the same trend, and E4My accumulators fail to converge at all.

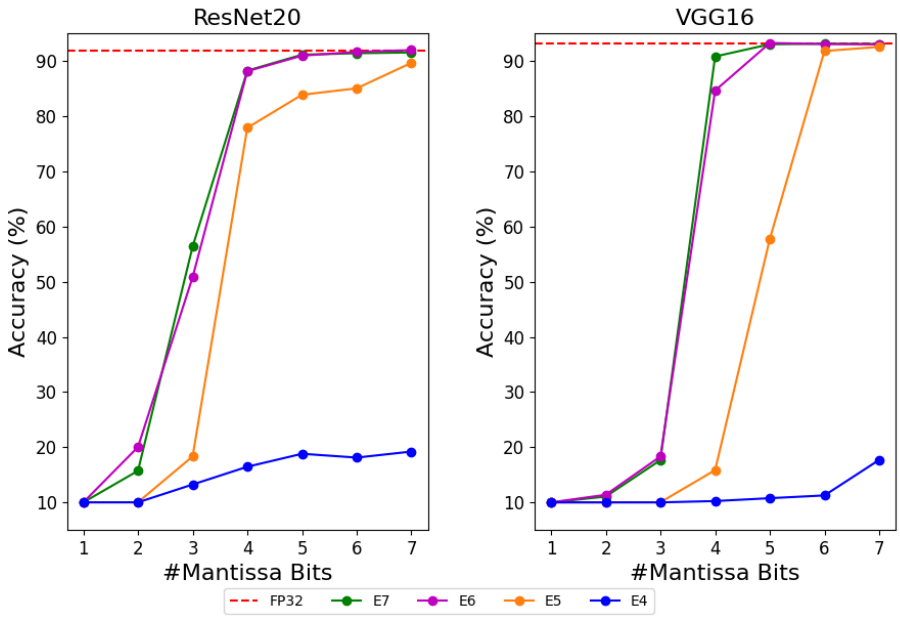


Figure 4.13: The graph shows the best test accuracy achieved by each low-precision floating-point accumulator configuration (multiplier data type is fixed to FP32). The x-axis indicates the exponent bits count, the y-axis indicates the mantissa bits count, and the z-axis indicates the prediction accuracy in percentage.

The observations of the result are two-fold. First, the floating-point GEMM accumulator seems to require more than 4 bits mantissa to avoid swamping. Second, the floating-point GEMM accumulator without any ancillary technique requires more than 6 bits for the exponent to guarantee accuracy. The second condition is also observed in the multiplier investigation; running into the same phenomena is reasonable given that input values of the GEMM accumulators are the output values of the GEMM multipliers.

To address the accuracy degradation of the floating-point accumulators, we have applied the adaptive loss-scaling technique, and Figure 4.14 shows

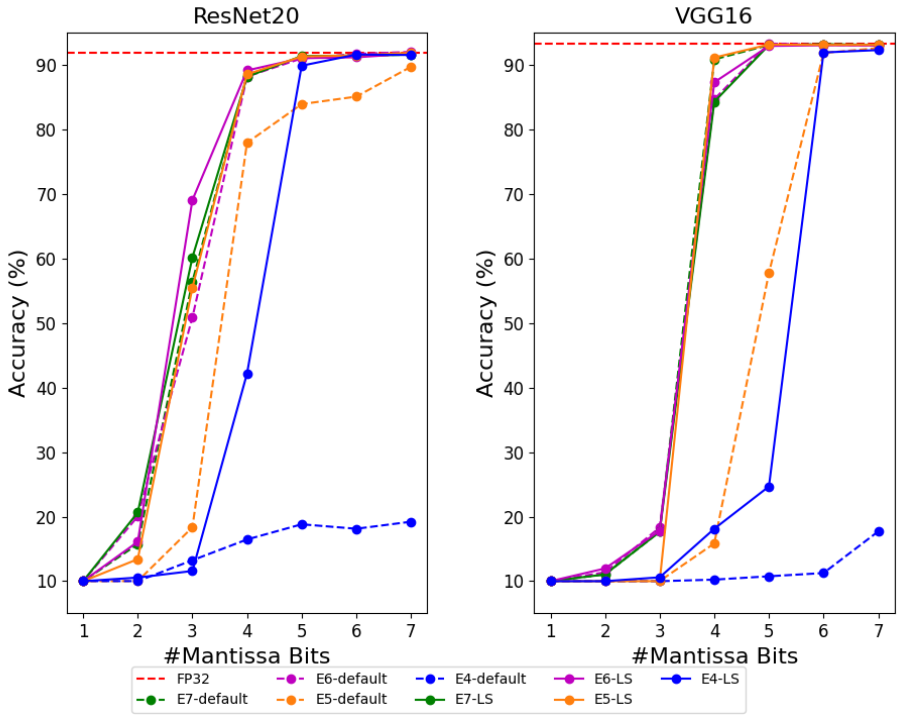


Figure 4.14: The graph shows the best test accuracy achieved by each low-precision floating-point accumulator configuration with or without adaptive loss scaling (multiplier data type is fixed to FP32). The dashed lines show the result of training without loss scaling, and the solid lines show the training result with adaptive loss scaling.

the results. The E5My accumulators with more than 5 bits mantissa regain the accuracy close to FP32 baseline, and E4My accumulators with more than 4/5 bits mantissa on ResNet20/VGG16 training also achieves FP32 comparable results after loss scaling. However, the trend of accuracy degradation observed for the accumulators using less than or equal to 4-bit mantissa still exists. Although the loss scaling technique can resolve the representation issue arising from gradient values, it cannot address the swamping error.

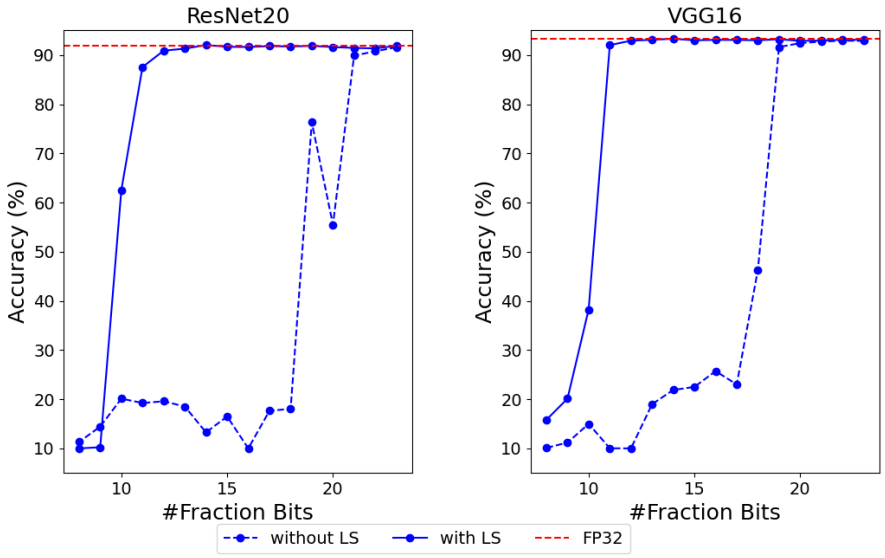


Figure 4.15: The graph shows the best test accuracy performed by each Q-notation fixed-point accumulator configuration (multiplier is fixed to FP32), while keeping the bit length of integer part to 8 bits. The blue solid line indicates the result with adaptive loss-scaling technique, and the blue dashed line indicates the results without the loss-scaling technique.

Assuming that the trend with few mantissa bits is correlated with the error, it is logical to observe the loss-scaling factor does not change the tendency.

Figure 4.15 shows the best accuracy achieved by fixed-point GEMM accumulators. The fixed-point accumulators also initially suffer from the representation range issue (dashed line), but the problem is resolved by employing the adaptive-loss-scaling technique (solid line). In both ResNet20 and VGG16 models, we can reduce the fixed-point accumulator length to Q8.12 (i.e., 20bits) while restraining the accuracy drop from FP32 baseline to be within 1%.

MAC Investigation

Based on the individual operator investigation results, we explored replacing both GEMM multiplier and accumulator with the low-precision operators. For the investigation, we employ the floating-point multipliers of `CFG-1` (default), `CFG-4` (without output rounding nor NaN support), and `CFG-5` (custom subnormal input encoding) given that training with those multipliers achieve a better accuracy compared to the others (Figure 4.9).

We use E5M2 data type for the `CFG-1` multiplier variant and E4M2 data type for the `CFG-4` and `CFG-5` variants, because those configurations have good accuracy with the fewest mantissa bits (Figure 4.4 and Figure 4.12). Based on Section 4.2.2, the loss-scaling factor is initialized to 4096.

In addition, we choose E5M5 for the floating-point accumulators and Q8.12 for the fixed-point accumulators. Since using even fewer bits ends up with degraded accuracy (Figure 4.14), the accumulator should be ExM5 at least. Q8.12 is chosen because it is the smallest fixed-point data type achieving FP32-comparable results.

Figure 4.16 shows the best accuracy achieved by training when using one of E5M2(`CFG-1`)/E4M2(`CFG-4`)/E4M2(`CFG-5`) with either an E5M5 or Q8.12 accumulator. As shown, the training with E5M5 accumulator still converges with accuracy degradation to some extent, whereas the training using the Q8.12 fixed-point accumulator completely diverges for the training using E4M2 multipliers. Observing that the adaptive loss-scaling factor of those failed trainings is set to 1 at the end of training, we hypothesize the failure might be due to a mismatch in desired loss-scaling factor for

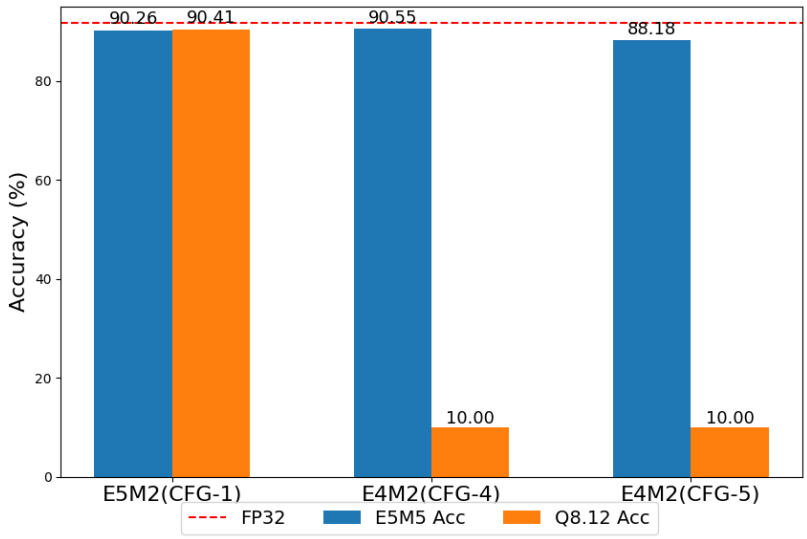


Figure 4.16: The graph shows the best test accuracy obtained by different combinations of low-precision floating-point multipliers and floating-point/fixed-point accumulators. The blue bars show the result using the floating-point (E5M5) accumulators, and the orange bars show the result using fixed-point accumulators.

the E4M2 multiplier and Q8.12 accumulators. Following the adaptive loss-scaling algorithm, the scaling factor is halved down when any of the GEMM computations in the iteration incur overflow. The loss-scaling factor of 1 at the end of the training indicates that overflow is detected several times. While decreasing the factor based on the Q8.12 range, the E4M2 multiplier cannot represent the small values due to its small exponent value range compared to E5M2.

Following the hypothesis, we evaluated E5M2 multiplier using `CFG-4` /`CFG-5` variants with the Q8.12 fixed-point accumulators. Since both `CFG-4` and `CFG-5` variants do not apply rounding, the output of those E5M2 mul-

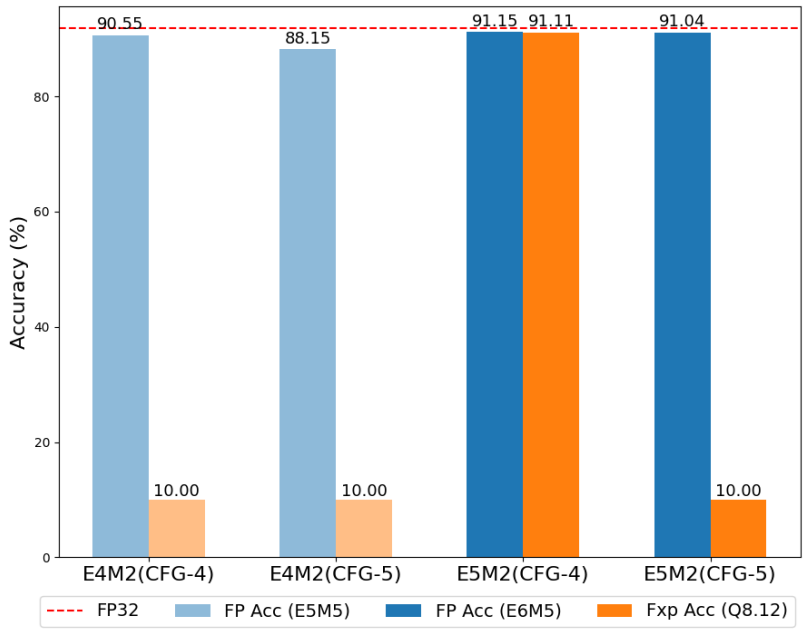


Figure 4.17: The graph shows the best test accuracy achieved by the combinations of the multiplier without output rounding (`CFG-4` and `CFG-5`) in E4M2/E5M2 data type, and the floating-point accumulator (E5M5/E6M5 for E4M2/E5M2 multiplier) and fixed-point accumulator (Q8.12). The blue bars show the result using the floating-point accumulators, and the orange bars show the result using fixed-point accumulators.

tipliers is in the E6M5 data type. Accordingly, we measured the training configurations with E5M2 multipliers and E6M5 accumulators.

Figure 4.17 shows the training results using E5M2 multiplier of `CFG-4` or `CFG-5` and the floating-point accumulator (E6M5) and the fixed-point accumulator (Q8.12). After replacing the E4M2 `CFG-4` multipliers with the E5M2 ones, the mixed-format training regains the prediction accuracy close to the FP32 baseline (the degradation is within 1%). A similar training

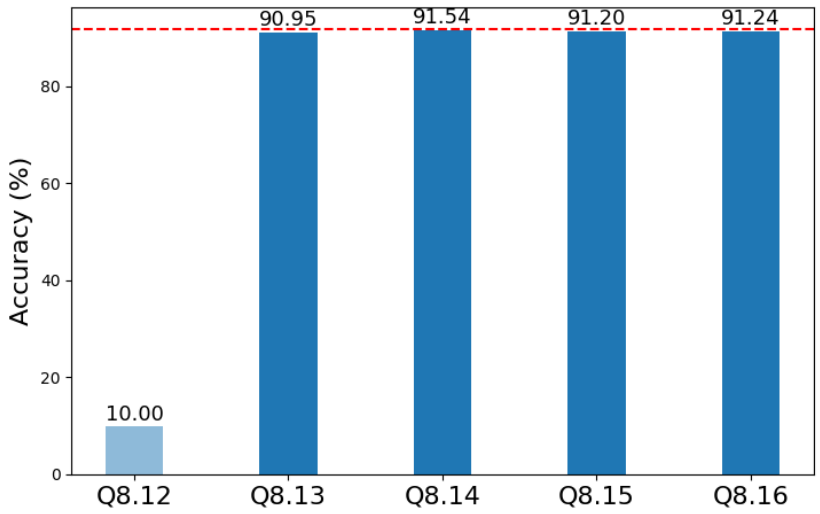


Figure 4.18: The test accuracy achieved by the configurations using E5M2 multiplier of `CFG-5` and different fixed-point data type accumulators (Q8.12, Q8.13, Q8.14, Q8.15, Q8.16).

accuracy improvement is observed for the floating-point-only configurations. In contrast, the mixed-format training employing E5M2 `CFG-5` still diverges; given that the floating-point-only configuration that uses the same multiplier converges, we conjecture that the result is due to the representation range of the fixed-point accumulator.

As shown in Figure 4.18, after adding one or more bits to the fractional part of the fixed-point accumulators, the accuracy of the mixed-format training using E5M2 `CFG-5` multipliers also recovers to FP32 baseline result (the degradation is also within 1%).

In this section, we observed that there are two configurations achieving FP32 baseline result with E5M2 `CFG-5` multiplier: E5M2(`CFG-5`) + E6M5,

Table 4.6: Kernel LUT Count and Accuracy of the Configurations that Obtained FP32-Comparable Results

configurations	LUTs	DSPs	Test Acc. (%)
FP32	58,420	320	91.85
E5M2(`CFG-5`) + E6M5	42,640	0	91.04
E5M2(`CFG-5`) + `Q8.13`	43,471	0	90.95

and E5M2(`CFG-5`) + Q8.13. Table 4.6 summarizes the resource utilization and predication accuracy.

The configurations applying the fixed-point accumulator achieve slightly better PE-wise resource utilization, but the total LUT count of the same configuration exceeds that of the floating-point-only operations. As mentioned briefly in Section 3.3.9, the increase in LUT count probably arises from the internal buffering that transfers the output values from the systolic array to the data write module. The output value of the floating-point-only implementation only requires 12 bits for the buffer, whereas the mixed-data format counterpart needs 21 bits in total. If we can decrease the bit-length for fixed-point outputs, while we keep using Q8.13 for the accumulation, the resource utilization can be further reduced; this investigation is left to future work.

To investigate the viability of the proposed configurations, they are evaluated on Imagewoof dataset using the ResNet50 model. As shown in Table 4.7, the floating-point only configuration using E5M2 for multiplier and E6M5 for accumulator still achieves FP32-comparable result on Imagewoof experiments. On the other hand, the proposed mixed-data format configuration using Q8.13 for the accumulator failed to converge.

The results indicate that bit counts required for the fixed-point accumu-

Table 4.7: Kernel LUT Count and Accuracy of the Same Configurations on Imagewoof Training

configurations	LUTs	DSPs	Test Acc. (%)
FP32	58,420	320	57.5719
E5M2(`CFG-5`) + E6M5	42,640	0	59.7862
E5M2(`CFG-5`) + Q8.13	43,471	0	10.9188

lator is more sensitive to the model and target task sizes compared to the floating-point counterpart. The difference probably arises from the presence/absence of the exponent field; as the model size increases and accumulation length increases, the intermediate sum value grows up as well. A floating-point accumulator, which has an exponent field, can broaden representation range exponentially by adding 1 bit, but a fixed-point accumulator, which does not have an exponent field, can extend the range linearly by adding the same 1 bit. Hence, a fixed-point accumulator needs more bits to secure the same representation range as a floating-point accumulator; it is reasonable to observe that training using a fixed-point accumulator is more sensitive to the model size than the one using a floating-point accumulator, given that adaptive loss-scaling algorithm forces the model to skip the weight update if any GEMM computation incurs overflow.

In addition, it is also unclear at which computation overflow occurs. The overflow incurred during the GEMM computation in forward propagation also impacts training results, and adaptive loss scaling cannot help the representation range issue regarding activation or weights; further investigation regarding these ambiguities is preferred, but left as future work.

4.3 Summary

This chapter has investigated low-precision training behavior using our framework, Archimedes-MPO. In the first experiment targeting the small MNIST data set, we reduced the precision of GEMM arithmetic to E6M3, Q7.7 or a combination of E5M1 multiplier and Q7.7 accumulator for both LeNet5 and MLP models without significant accuracy drop (Table 4.3). The low-precision training on MLP achieves good accuracy, but it is a bit short of the FP32-equivalent accuracy.

Referring to the resource utilization reported in Table 3.10, the most resource-efficient configuration which can achieve FP32-comparable training results with the MNIST data set is a mixed-precision combination of E5M1 multipliers and Q7.7 accumulators; the LUT count is lower than not only FP32 but also other homogeneous low-precision implementations with similar accuracy.

In a subsequent series of experiments targeting the larger CIFAR-10 dataset, we decreased the precision of GEMM arithmetic to E5M2 multiplication and either E6M5 or Q8.13 accumulation while maintaining FP32-comparable accuracy. Due to the GEMM design, the mixed-data format implementation using a Q8.13 accumulator is not as area-efficient as a floating-point-only implementation (Table 4.6) because of buffering and downstream logic required by the wider fixed-point accumulator (21 versus 12 bits for floating-point accumulator).

The CIFAR-10 experiments also verified good results from custom low-precision floating-point multipliers that get rid of the subnormal output,

rounding, NaN values, and adopt custom subnormal input encoding. Adaptive loss scaling is essential as well.

The best configurations determined for CIFAR-10 were also evaluated on Imagewoof dataset. We reduced the precision of the GEMM multiplier and floating-point accumulator to E5M2 and E6M5 respectively to achieve similar results to the FP32 baseline. However, the 21 bit fixed-point accumulator alternative suffers from significant accuracy drop; the difference probably arises due to limited range. Applying the mixed-data format configuration on larger models requires further investigation.

Chapter 5

Conclusion

In this thesis, we consider the use of mixed-precision operations in the GEMM kernel for DNN training. We explored resource-efficient multiplier configurations that achieve a good resource-accuracy tradeoff. To conduct these investigations, we have developed a framework, Archimedes-MPO. The framework allows each layer to use custom precisions, and also accelerates the GEMM computation with GPU and FPGA kernels.

Using the framework, we measure the exact impact of various multiplier and accumulator designs on resource utilization and training accuracy. We first evaluate the mixed-precision low-precision training with MNIST dataset using LeNet5 and MLP models, and extend the study to CIFAR-10 dataset using ResNet20 and VGG16 models with adaptive loss scaling. After the CIFAR-10 investigation, we evaluated the viability of the best configurations on a larger dataset, Imagewoof, using the larger ResNet50 model.

The results primarily indicate three things. First, multiplier area is heavily affected by mantissa width; with low-precision implementations, reducing mantissa is essential to suppress resource utilization, even if doing so requires an increase to exponent width.

Second, customizing number representations used in low-precision floating-point multiplier is beneficial to both area and accuracy. The de-facto IEEE-

754 standard requires multipliers to handle exceptional values such as subnormals, NaN, and infinity as well as the proper rounding of the output. However, our custom multiplier eliminated subnormal outputs, NaN and rounding to save area and improve prediction accuracy at the same time. In the process of removing NaNs, we altered the encoding for Infinity as well. Removal of subnormal input saves area further, but degrades test accuracy slightly. Instead, an alternative to subnormals is proposed that incurs no area overhead and restores some of the lost accuracy.

Third, using low-precision multiplier and fixed-point accumulator simultaneously enables us to implement each MAC efficiently. However, converting between format types and downstream logic area increases because fixed-point accumulators are necessarily wider. As a result, the low-precision floating-point only implementation can achieve better resource utilization as a whole. Following that, the mixed-data format implementation achieves the best resource-accuracy tradeoff for the MNIST dataset with E5M1 multiplier and Q7.7 accumulator (14bits), but in CIFAR-10 experiments, the floating-point only configuration achieves a slightly better resource tradeoff compared the mixed-data format counterpart that employs Q8.13 accumulator (21 bits) for the accumulator. With Imagewoof, the Q8.13 accumulator failed to converge. Since GEMM hardware kernels can differ in how, where and when accumulation is done, there is a rich design space for alternative accumulation strategies which deserve further investigation. Further investigation and verification in larger datasets and more elaborate models are needed.

5.1 Future Work

Several extensions and investigations are left for future work in different aspects: framework, low-precision data formats, and targeting tasks/models.

Framework

Presently all layers apart from convolution and fully-connected layers use FP32 to isolate the impact of mixed-precision GEMM arithmetic. Using low-precision on other layers, such as the batch-normalization layer or during weight updates, and assessing the impact on the model accuracy is also essential for understanding the behavior of low-precision training.

The current GEMM accumulator adds up the multiplier outputs one by one, but accumulating values in this way can suffer from the swamping error [46] in the middle of the sequence if the accumulation length is too long. Instead, other summation methods, such as chunking, adder trees and Kahan summation [22] may be useful. There is significant design flexibility on how to structure accumulators that may achieve interesting results.

Data Format

Given that the floating-point multiplier and the fixed-point accumulator combination achieves a good resource-utilization and accuracy tradeoff, the next data format to consider should be block floating-point. A block floating-point implementation requires an extra ancillary module to align the mantissa of the values in the same block before and after the GEMM operations, and it also takes extra time to investigate the proper block size for the task.

Due to time constraints, we have not investigated the data format, but this is a direction to explore in future work. In addition to the use of the block floating-point data format, we suggest investigating the use of other number formats such as posits [5, 16, 43], logarithmic number systems [1, 28, 33, 55], as well as block mini-floats [13], to DNN training.

Targeting Tasks and Models

As for image classification tasks, we will investigate the approach in larger datasets such as ImageNet in future work. In addition, we consider extending the test cases to include tasks other than image classification, with particular attention to include use cases for edge device training such as transfer learning and reinforcement learning.

In recent work, variants of Transformer models [44] have attracted attention because they can generalize to large datasets such as ImageNet-21k (a superset of the conventional ImageNet dataset) better than the conventional convolution-based models [10, 29, 42]. Although it is reported that those models cannot be applied directly to small datasets such as MNIST, and might not be suitable for the deployment to embedded devices, studying the mixed-precision low-precision training of such models can be one option to take in future work.

Bibliography

[1] Mohammad Saeed Ansari, Bruce F. Cockburn, and Jie Han. An improved logarithmic multiplier for energy-efficient neural computing. *IEEE Transactions on Computers*, 70(4):614–625, 2021.

[2] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 5151–5159, Red Hook, NY, USA, 2018. Curran Associates Inc.

[3] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[4] Yanpeng Cao, Chengcheng Wang, and Yongming Tang. Explore efficient lut-based architecture for quantized convolutional neural networks on fpga. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 232–232, 2020.

[5] Rohit Chaurasiya, John Gustafson, Rahul Shrestha, Jonathan Neudorfer, Sangeeth Nambiar, Kaustav Niyogi, Farhad Merchant, and Rainer Leupers. Parameterized posit arithmetic hardware generator. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 334–341, 2018.

[6] Brian Chmiel, Liad Ben-Uri, Moran Shkolnik, Elad Hoffer, Ron Banner, and Daniel Soudry. Neural gradients are near-lognormal: improved quantized and sparse training. In *International Conference on Learning Representations*, 2021.

[7] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj D. Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesús Corbal, Nikita Shustrov, Roman Dubtsov, Evarist Fomenko, and Vadim O. Pirogov. Mixed precision training of convolutional neural networks using integer operations. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[8] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. Flexible communication avoiding matrix multiplication on fpga with high-level synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 244–254, 2020.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[11] Mario Drumond, Tao LIN, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[12] Sean Fox, Julian Faraone, David Boland, Kees Vissers, and Philip H.W. Leong. Training deep neural networks in low-precision with high accuracy using fpgas. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–9, 2019.

[13] Sean Fox, Seyedramin Rasoulinezhad, Julian Faraone, david boland, and Philip Leong. A block minifloat representation for training deep neural networks. In *International Conference on Learning Representations*, 2021.

[14] Yonggan Fu, Haoran You, Yang Zhao, Yue Wang, Chaojian Li, Kailash Gopalakrishnan, Zhangyang Wang, and Yingyan Lin. Fractrain: Fractionally squeezing bit savings both temporally and spatially for efficient dnn training. In *NeurIPS*, 2020.

[15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 1737–1746. JMLR.org, 2015.

[16] John L. Gustafson and Isaac T. Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov.*, 4(2):71–86, 2017.

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

[18] Jeremy Howard. Imagewang.

[19] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[20] Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry P. Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. In Amir Globerson and Ricardo Silva, editors, *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018*, pages 876–885. AUAI Press, 2018.

[21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[22] W Kahan. Further remarks on reducing truncation errors. *Communications of the ACM*, 8:40, 1965.

[23] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of bfloat16 for deep learning training, 2019.

[24] Urs Köster, Tristan J. Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William H. Constable, Oğuz H. Elibol, Scott Gray, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Proceedings of the 31st Interna-*

tional Conference on Neural Information Processing Systems, NIPS'17, page 1740–1750, Red Hook, NY, USA, 2017. Curran Associates Inc.

[25] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[27] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[28] Edward H. Lee, Daisuke Miyashita, Elaina Chai, Boris Murmann, and S. Simon Wong. Lognet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5900–5904, 2017.

[29] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 10012–10022, October 2021.

[30] Cheng Luo, Yuhua Wang, Wei Cao, Philip H.W. Leong, and Lingli Wang. Rna: An accurate residual network accelerator for quantized and reconstructed deep neural networks. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 60–603, 2018.

[31] Naveen Mellempudi, Sudarshan Srinivasan, Dipankar Das, and Bharat Kaul. Mixed precision training with 8-bit floating point, 2019.

[32] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[33] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation, 2016.

[34] Taiga Noumi. *tiny-dnn: header only, dependency-free deep learning framework in C++14*, 2017.

[35] NVIDIA. Train with mixed precision. Technical Report DA-08617-001, NVIDIA Corporation & affiliates, 2022.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach,

H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[37] Christopher De Sa, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R. Aberger, Kunle Olukotun, and Christopher Ré. High-accuracy low-precision training, 2018.

[38] Charbel Sakr, Naigang Wang, Chia-Yu Chen, Jungwook Choi, Ankur Agrawal, Naresh R. Shanbhag, and Kailash Gopalakrishnan. Accumulation bit-width scaling for ultra-low precision training of deep networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

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

[40] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Xiaodong Cui, Wei Zhang, and K. Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. In *NeurIPS*, 2019.

[41] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi (Viji) Srinivasan, and Kailash Gopalakrishnan. Ultra-low precision 4-bit training of deep neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural*

Information Processing Systems, volume 33, pages 1796–1807. Curran Associates, Inc., 2020.

[42] Ilya O Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. Mlp-mixer: An all-mlp architecture for vision. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 24261–24272. Curran Associates, Inc., 2021.

[43] Yohann Uguen, Luc Forget, and Florent de Dinechin. Evaluating the hardware cost of the posit number system. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 106–113, 2019.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[45] Diederik Adriaan Vink, Aditya Rajagopal, Stylianos I. Venieris, and Christos-Savvas Bouganis. Caffe barista: Brewing caffe with fpgas in the training loop. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 317–322, 2020.

[46] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 7686–7695, Red Hook, NY, USA, 2018. Curran Associates Inc.

[47] Chen Wu, Mingyu Wang, Xinyuan Chu, Kun Wang, and Lei He. Low-precision floating-point arithmetic for high-performance fpga-based cnn acceleration. *ACM Trans. Reconfigurable Technol. Syst.*, 15(1), nov 2021.

[48] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[49] Xilinx. Logicore ip floating-point operator v6.1. Technical Report PG060, Xilinx Inc., 2012.

[50] Xilinx. Higher performance neural networks with small floating point. Technical Report WP530, Xilinx Inc., 2021.

[51] Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Christopher De Sa. SWALP : Stochastic weight averaging in low precision training. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long*

Beach, California, USA, volume 97 of *Proceedings of Machine Learning Research*, pages 7015–7024. PMLR, 2019.

[52] Yukuan Yang, Lei Deng, Shuang Wu, Tianyi Yan, Yuan Xie, and Guoqi Li. Training high-performance and large-scale deep neural networks with full 8-bit integers. *Neural Networks*, 125:70–82, 2020.

[53] Pedram Zamirai, Jian Zhang, Christopher R. Aberger, and Christopher De Sa. Revisiting bfloat16 training, 2021.

[54] Tianyi Zhang, Zhiqiu Lin, Guandao Yang, and Christopher De Sa. QPyTorch: A Low-Precision Arithmetic Simulation Framework. *arXiv preprint 1910.04540*, 2019.

[55] Jiawei Zhao, Steve Dai, Rangharajan Venkatesan, Ming-Yu Liu, Brucek Khailany, Bill Dally, and Anima Anandkumar. Low-precision training in logarithmic number system using multiplicative weight update, 2021.

[56] Ruizhe Zhao, Brian Vogel, Tanvir Ahmed, and Wayne Luk. Reducing underflow in mixed precision training by gradient scaling. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 2922–2928. International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track.