2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2027-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPCA53966.2022.00067

FAST: DNN Training Under Variable Precision Block Floating Point with Stochastic Rounding

Sai Qian Zhang[1], Bradley McDanel[2], and H.T. Kung[1]

[1]Harvard University
[2]Franklin and Marshall College

***Abstract*—Block Floating Point (BFP) can efficiently support quantization for Deep Neural Network (DNN) training by providing a wide dynamic range via a shared exponent across a group of values. In this paper, we propose a Fast First, Accurate Second Training (FAST) system for DNNs, where the weights, activations, and gradients are represented in BFP. FAST supports matrix multiplication with variable precision BFP input operands, enabling incremental increases in DNN precision throughout training. By increasing the BFP precision across both training iterations and DNN layers, FAST can greatly shorten the training time while reducing overall hardware resource usage. Our FAST Multipler-Accumulator (fMAC) supports dot product computations under multiple BFP precisions. We validate our FAST system on multiple DNNs with different datasets, demonstrating a 2-6× speedup in training on a single-chip platform over prior work based on mixed-precision or block floating point number systems while achieving similar performance in validation accuracy.**

I. Introduction

Custom floating point (FP) formats, such as Google's bfloat16 [1] and Nvidia's TensorFloat 32 [2], are increasingly replacing IEEE 754 32-bit floating point (FP32) for DNN training. These formats more efficiently fit the empirical distribution of DNN weight, data, and gradient values, leading to a smaller hardware footprint for the multiplier-accumulator (MAC) unit. However, these formats are still significantly more expensive to implement than fixed point (INT) formats of similar bitwidths due to mantissa alignments which are required for each floating point MAC operation.

By comparison, Block Floating Point (BFP) [3] formats offer a middle ground between FP and INT formats, by enforcing that a group of values share a common exponent while maintaining individual mantissas. This constraint enables BFP to achieve higher efficiency than FP for dot product (DP) computations for multiple reasons. First, mantissa alignments are only required at the BFP group level as opposed to after each FP multiplication. Second, there is only one exponent addition between each group as opposed to each FP multiplication. Therefore, performing DP computations in BFP can lead to a significant improvement in training efficiency.

In this work, we propose a **F**ast First, **A**ccurate **S**econd **T**raining (**FAST**) system for variable precision BFP DNN training. Here, variable precision means that (1) the system efficiently supports BFP formats across a range of mantissa widths during training and (2) dot products between BFPs with different mantissa widths is permitted. To support our system, we have designed a FAST multiplier-accumulator (fMAC) which operates on n-bit chunks of mantissas across two groups of BFP numbers being multiplied. Throughout this paper, we use 2-bit chunks. Sub-dividing the computation into 2-bit chunks allows the same fMAC to implement arithmetic operations involving higher precision mantissas by simply running multiple passes of the fMAC. For instance, multiplying two groups with 2-bit and 4-bit BFP mantissas translates to $\frac{2}{2} \times \frac{4}{2} = 2$ passes. The rate at which our FAST system performs dot product computations is based on the BFP precision of the two vectors being multiplied.

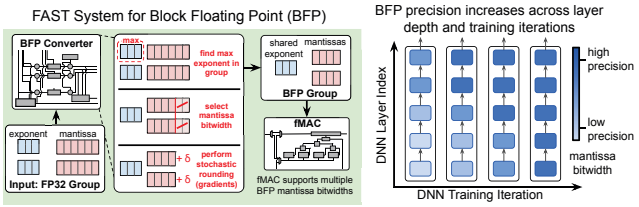

Fig. 1. (left) Overview of the proposed FAST system for DNN training using variable precision BFP with stochastic rounding. A group of FP32 values are converted to a BFP group with a selected mantissa width based on the current error tolerance (see Algorithm 1). (right) FAST incrementally increases the BFP precision across both layer depths and training iterations. See Figure 19 on corresponding empirical data obtained via DNN training with FAST.

With FAST, we propose a DNN training regime that starts with low-precision BFP and increases the precision of weights, data, and gradients over the course of training. Figure 1 presents an overview of how FAST can accelerate training via low-precision operations. The left side of the figure provides a sketch of the FAST system, which supports FP32 to BFP conversion for a range of mantissa bitwidths and stochastic rounding for gradients (to maintain training stability under low-precision BFP). The FAST compute engine consists of a systolic array [4] of fMAC units, which efficiently supports BFP dot products with varying mantissa bitwidths. The right side of the figure shows how the precision of the mantissa field in BFP for a DNN increases across DNN layers and over training iterations. While each DNN layer in the figure is presented with a single precision, in practice the precision of the weight, data, and gradient tensors in each layer are selected independently for a given iteration (see Algorithm 1). Using this approach, FAST is able to reduce DNN training time on (1) CNNs for ImageNet [5], (2) Transformers for the IWSLT14 German-English benchmark [6], and (3) YOLOv2 [7] for the PASCAL VOC2012 [8] dataset.

978-1-6654-2027-3/22/$31.00 ©2022 IEEE
DOI 10.1109/HPCA53966.2022.00067

TABLE I
TERMINOLOGY AND NOTATION USED IN THE PAPER.

Notation	Description
BFP	BFP-quantized values
DP	Dot product
FP	Floating point. Assume FP32, unless otherwise stated
INT	Fixed point integer
UQ	Uniform quantization
SR	Stochastic rounding
g	Group size for BFP. Assume g=16, unless otherwise stated
e	Exponent bitwidth for FP and BFP
m	Mantissa bitwidth for FP, BFP, and INT

In FAST, we use (1) BFP for variable precision training and (2) BFP with stochastic rounding. BFP is an old idea dating back as early as the 1950 (see, e.g., [3]), and there has been recent literature demonstrating the advantages of using BFP in DNN inference [9] and training [10]. We believe that our ideas of (1) and (2) are novel. For (1), we point out the convenience in Section VII of implementing variable precision hardware for BFP. For (2), we note in Section III-C that using stochastic rounding in conjunction with BFP is critical to model accuracy, especially when using BFP for gradients with low-precision mantissa (e.g., 2 or 4 bits). In Section III-D we provide an analysis of the reasons for using stochastic rounding in BFP. The main contributions of the paper are:

- The *FAST variable precision training algorithm* for efficient DNN training. The proposed solution reduces total training time by adaptively selecting the optimal precision for weights, data, and gradients in every DNN layers at each iteration.
- Our proposed use of (1) BFP for variable precision training and (2) BFP with stochastic rounding. We provide a *novel analysis* of the impact of applying stochastic rounding to weight gradient updates for minimizing the loss in gradient decent (Theorem 1 in Section III-D).
- A modular architecture consisting of *FAST multiplier-accumulator* (fMAC) for groups of BFP values. fMAC operates on chunks of BFP mantissas (e.g., 2-bit chunks) to support variable-width mantissas in 2-bit increments.
- The *FAST system* that supports training in both low and high precision BFP. Critically, this system enables dot products between vectors of varying BFP precisions.

The remainder of the paper is organized as follows: Section II presents the background and related work. Section III provides an overview of using BFP for dot products in DNN training. Next, Section IV provides an overview of our FAST strategy for training DNNs using variable precision BFP. In Section V, we present the FAST system. The performance results for training DNNs using variable precision BFP are given in Section VI, followed by the evaluation of the FAST hardware system in Section VII.

Table I lists the terminology and notation used in the paper.

II. BACKGROUND AND RELATED WORK

In Section II-A, we provide an overview of number formats for DNN training and inference. Section II-B provides an overview of the computation dataflow of DNN training. Next, in Section II-C, we review related work on hardware accelerators for DNN training. Finally, in Section II-D, we discuss how FAST could reduce the training time for methods that find efficient DNNs, such as under the Lottery Ticket Hypothesis [11].

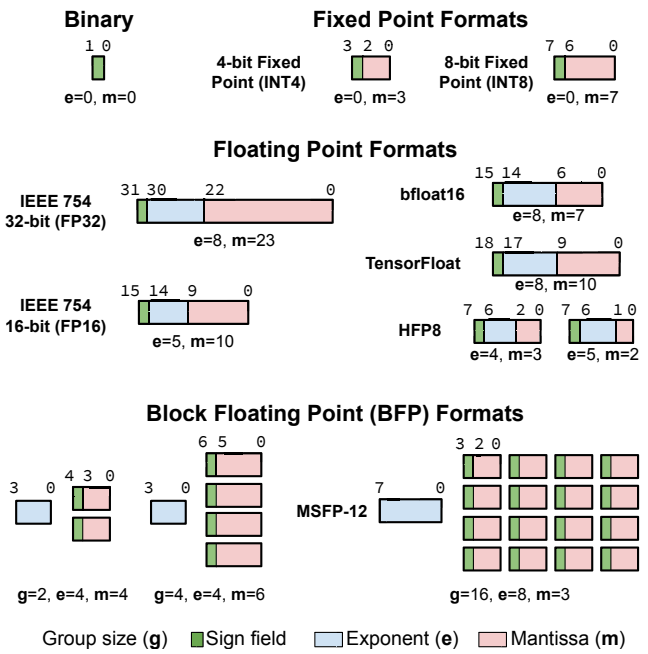

Fig. 2. Number formats commonly used for DNN training and inference. Fixed point formats (top) are often used for inference. Floating point formats (middle) are used for training, with bfloat16 and TensorFloat replacing IEEE 754 in most cases by increasing the number of exponent bits to widen the dynamic range. BFP (bottom) fits between fixed and floating point by sharing a single exponent across the group values.

A. Number Formats for DNNs

As the majority of computation in both DNN training and inference are dot products, the formats for their underlying numbers have been extensively studied to make this computation efficient. Figure 2 divides various formats into three groups: fixed point (top), floating point (middle), and BFP (bottom). The number of exponent bits (e) and mantissa bits (m) are provided for each format.

Fixed point formats do not have an exponent field, which reduces the dynamic range that can be represented but simplifies the hardware. The use of fixed point formats for DNN training and inference has been well explored [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]. The smallest format is a 1-bit binary representation (upper left of Figure 2) used by binarized neural networks [25] which has no exponent or mantissa bits. Floating point formats have a larger dynamic range than fixed point, making them more amenable to wider dynamic range of gradients in DNN training [26]. IEEE 754 32-bit IEEE floating point or FP32 (middle left of Figure 2) is the conventional format for DNN training. Mixed precision training operates on some tensors in a higher precision and other tensors in a lower precision. For instance, Nvidia Mixed

Precision (MP) [27] proposed to perform most computations in the forward pass in FP16, while keeping an FP32 copy of the weights for updating gradients in the backward pass during training. Custom floating point formats like bfloat16 [1], TensorFloat [2] and HFP8 [28] (middle right of Figure 2) operate in a similar mixed precision regime and have been shown to work as well as FP32 for training accurate DNNs. For example, HFP8 performs forward pass computations using 8-bit FP with one bit sign, 4-bit exponent, 3-bit mantissa (1-4-3) and backward pass computations with one bit sign, 5-bit exponent and 2-bit mantissa (1-5-2). These custom formats increase the number of exponent bits and decrease the number of mantissa bits to better approximate the distribution of gradients during training.

While BFP represents a promising direction for improving the efficiency of DNN training as middle ground between fixed and floating point, there has been a small amount of prior work in designing efficient hardware to support it. Drumond et al. [10] proposed to use a large BFP group size of 576 (a 2D tile of size 24×24) which requires a wide mantissa bitwidth of m=12 to achieve good accuracy. Compared to these prior approaches, we show in Section VI-B, that training under INT with similar number of bits (e.g., 12 bits) also has good performance, which suggests that BFP has little advantages over INT12 for such large tiles. Additionally, the paper did not describe a detailed hardware design for the implementation of BFP computation.

More recently, Microsoft proposed a BFP format (MSFP-12 in lower right of Figure 2) for DNN inference via post-training quantization [9] on their Project Brainwave FPGA cloud platform [29]. In our paper, by using BFP-aware DNN training instead of post-training quantization, we are able to use a smaller exponent width of 4 instead of 8 while achieving similar inference accuracy. For training, as we show in Figure 22, via variable precision BFP, FAST reduces the training time and energy consumption compared to the previously mentioned training based on floating point or BFP formats.

B. Matrix Computation of DNN Training

Each iteration of DNN training on a mini-batch consists of a forward pass to compute a loss and a backward pass to update the DNN weights with gradients computed from the loss. Figure 3 illustrates all of the matrix computations required for both forward and backward passes for one convolutional layer in a CNN (fully connected layers operate in a similar manner). Both a convolutional view and a corresponding matrix operation view are presented. During the forward pass, the input activations (A) are convolved with the layer weights (W) to compute the output (O) as depicted in Figure 3a. Then, O is passed through normalization and a non-linear activation function, to become the input activations for the next layer.

During the backward pass, two convolutions are performed at each layer. Figure 3b shows how the output gradients ∇O are convolved with the transposed weights W^T to compute the activation gradients ∇A, which will then be passed to the preceding layer. In Figure 3c, the transposed input activations A^T are convolved with the output gradients ∇O in order to

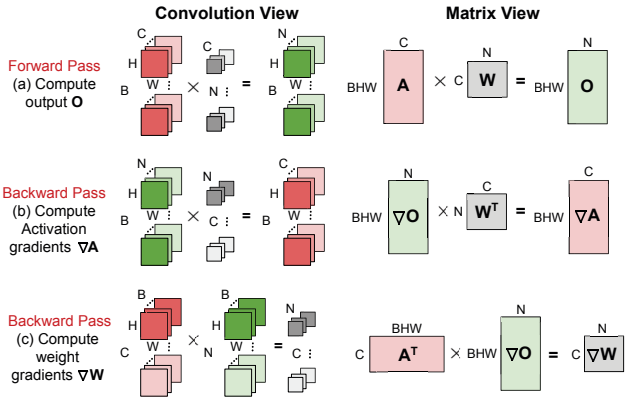

Fig. 3. The forward and backward pass steps for a single layer of DNN training represented in a convolution view (left) and a matrix operation view (right) with matrix dimensionalities shown. The kernel size of the convolution filters is 1×1 for presentation simplicity.

compute the weight gradients ∇W. These weight gradients ∇W are then added in an elementwise fashion with the weights W in order to compute the updated weights W'. For other types of optimizers, such as Adam [30], additional operations are required to compute the first and second moments before updating the weights W.

C. Accelerators for DNN Training

Some of the previous work on accelerating DNN training has a focus on leveraging sparsity present in weights and activations [31], [32], [33], [34]. TensorDash [31] accelerates the DNN training process while achieving higher energy efficiency via eliminating the ineffectual operations resulted from the sparse input data. Eager Pruning [32] and Procrustes [33] improve DNN training efficiency by co-designing the training algorithm with the target hardware platform ("hardware-aware training"). Unimportant DNN weights are pruned in the middle of the DNN training process and ineffectual operation involving zero weights can be eliminated without impacting the final accuracy. In comparison, our approach applies BFP to dynamically adjust the precision of DNN training, which is in orthogonal to these methods which exploit value-level sparsity.

Multi-precision methods of reducing the computation of DNNs have also been explored in the literature. Stripes [35] multiplies two 16-bit integers by only adding those shifted multiplicands corresponding to the nonzero bits during DNN inference. In this work, we use a bit-parallel implementation on 2-bit chunks of mantissas. Lee et al. [36] proposed using fine-grained mixed precision (FGMP), which represents some parts of a tensor in FP8 and other parts in FP16. FAST uses both 2-bit and 4-bit mantissas, and iterates like FGMP on the low-bitwidth hardware for performing the high-bitwidth arithmetics. FAST performs an integer MAC for each partial product in a BFP group, rather than an FP MAC as in FGMP.

D. Finding Winning Tickets under Efficient DNN Training

The *Lottery Ticket Hypothesis* (LTH) [11] conjectures that dense, randomly-initialized DNNs always contain small subnetworks which can match or even outperform the test accuracy

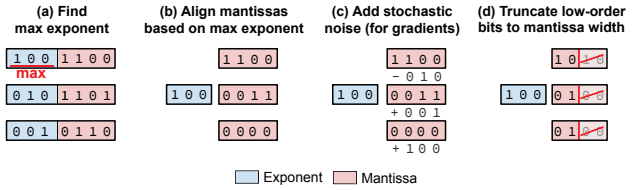

Fig. 4. (a) The max exponent across all values in the group is found. (b) The mantissas fields are aligned based on differences to the max exponent. (c) Stochastic noise is added to each mantissa (for gradients only). (d) The low-order mantissa bits are truncated to a fixed width. (m = 3 in the example). The sign of each value is omitted for presentation clarity.

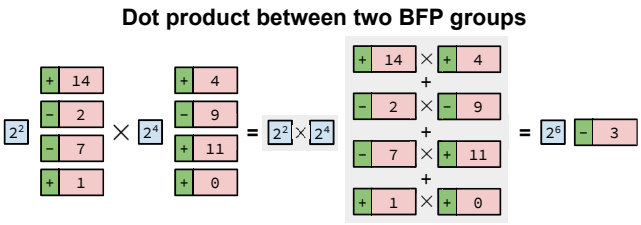

Fig. 5. The dot product between two BFP groups can be decomposed into fixed point multiplication between each pair of values in the groups and exponent addition between the two shared exponents.

of original DNNs when trained alone from scratch using the original weight initiations. The process of finding these sub-networks (i.e., winning tickets) utilizes a procedure called iterative magnitude pruning, which requires repeated iterations of training and pruning until a target sparsity level is reached. In order to achieve high accuracy, it requires multiple rounds of training, making winning tickets computationally expensive to find. FAST can be leveraged to reduce the search time and energy of finding winning tickets as it reduces training time compared to conventional methods.

III. Overview of BFP with Stochastic Rounding

In this section, we provide an overview of how we use BFP under stochastic rounding (SR) in FAST to facilitate efficient and accurate DNN training.

A. Quantization from FP to BFP under Stochastic Rounding

Converting a group of FP values into a BFP group can be viewed as a type of quantization. Figure 4 shows this quantization process for converting a group of three FP values. First, in Figure 4a the largest exponent in the group is found, which becomes the shared exponent for the group. Then, in Figure 4b, the mantissas of each value are aligned based on the difference between the exponent of each value and the max exponent. Next, in Figure 4c, stochastic noise is added for gradients (critically important for low bitwidth mantissas). Finally, in Figure 4d, the low-order mantissa bits are truncated to a specified mantissa bitwidth.

B. Dot Product under FP, INT, and BFP

In this section, we discuss the conversion and computation costs of dot product (DP) under three number formats (BFP, INT, and FP). We argue that BFP DP is less costly than FP DP due to its use of shared exponents, and BFP DP is less costly than INT DP because the former can achieve the same accuracy as the latter with a smaller mantissa bitwidth.

Consider a DP of two length g vectors (e.g., an activation vector and a weight vector). The DP computation can be broken into two parts: (Part M) g integer multiplications for computing g partial products and (Part A) accumulation of g partial products resulting from part M.

1) FP Conversion Cost (BFP DP versus INT DP): Before performing BFP DP, we must first convert values in the two FP input vectors into BFP values. Since the scale factor for BFP quantization is a power of two, the conversion requires only bit shifts on mantissas.

After the DP is computed, we need to convert the result to FP to add to an accumulation across BFP groups. This conversion operation is an FP normalization that involves bit shifts of the mantissa. As we see below, the savings of BFP DP over FP DP justify these conversion costs.

For the INT DP, we need to perform similar conversions between FP and INT. Suppose that we use conventional uniform quantization (UQ) for the conversion from FP to INT. Since the scale factor is in FP, the INT conversion cost is much higher than the BFP conversion. The conversion of the DP result from INT to FP is an FP normalization, like the conversion of the BFP DP result to FP. The savings of BFP DP over INT DP alone are already substantial even without accounting that the BFP conversions are much lower than INT conversions.

2) Computation Cost (BFP DP versus FP DP): Figure 5 illustrates the dot product between two BFP groups of size g = 4. We see that BFP DP costs substantially less than FP DP for three reasons. First, for part M, BFP DP just needs to perform one exponent addition on the shared exponents of the two input vectors. In contrast, FP DP requires g exponent additions. Additionally, unlike FP DP, BFP DP does not perform FP normalization after each of the g multiplications. Finally, for part A, unlike FP DP, BFP DP does not need to align partial products, as they are already aligned.

3) Computation Cost (BFP DP versus INT DP): BFP can have a much smaller mantissa bitwidth m (e.g., m = 4) than INT (e.g., m = 12) while achieving similar classification accuracy (Table II). Since the computational complexity of fixed point multipliers scales in a quadratic fashion with bitwidth, for part M, BFP DP costs much less than INT due to the reduced m. But, BFP does incur a relatively small cost compared to INT for adding the shared exponents between the BFP groups.

C. BFP Exponent and Mantissa Bitwidths

The number of exponent and mantissa bits in BFP play different roles in determining the amount of quantization error after conversion from FP to BFP. If the shared exponent bitwidth is too small, then it may not be able to represent numbers in the dynamic range of the group. If the mantissa bitwidth is too small, then some values with smaller exponents

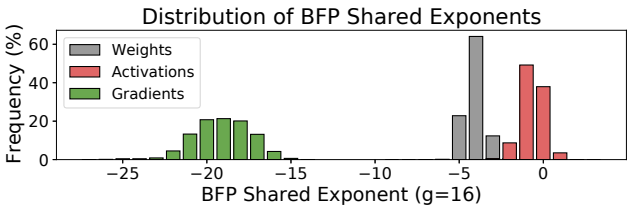

Fig. 6. The distribution of BFP shared exponents for weights, data, and gradients for layer 10 in ResNet-18 on ImageNet halfway through training.

in a BFP group will have all mantissa bits or most of them shifted out of range resulting in data loss (see the third value in Figures 4a and 4b with $m = 2$).

Figure 6 shows the distribution of BFP shared exponents for weights, data, and gradients under a group size $g = 16$ for layer 10 in ResNet-18 at the halfway point of the training (other layers and DNNs generally follow the same trend). For illustration clarity, in this example we do not bound the exponent range to a specific bitwidth (e.g., 3-bit shared exponent). We observe that weights and activations have a tighter exponent distribution (approximately from 2^{-5} to 2^{0}) while the gradient exponents are much smaller and fall under a wider distribution (approximately from 2^{-26} to 2^{-15}). Thus, we expect gradients to require higher precision to achieve the same level of quantization error. In FAST, we employ stochastic rounding for gradients in conversion from FP to BFP (Section III-A), which is critically important during the early stages of training when low-precision gradients are used (see analysis in Section III-D).

Figure 7 presents the distribution of the difference in exponents between the maximum exponent in a group and all other exponents in the group for three different group sizes (g = 8, 16, 32). The weight, data, and gradient tensors are taken from layer 10 in ResNet-18 at the halfway point of the training on ImageNet (other layers and DNNs generally follow the same trend). The difference dictates the amount of shifting required to align mantissas as depicted in Figures 4b. Large differences lead to large worst-case quantization errors. When the difference is larger than the mantissa bitwidth, all bits will be truncated. Compared to the weights and activations, the gradients have a much wider exponent disparity, leading to a larger quantization error. This is why stochastic rounding for gradient computations, as depicted in Figure 4c, is essential to achieve high accuracy when using a low number of mantissa bits (see analysis in Section III-D). We notice that the mass of each distribution moves to the right as the group size g increases, as indicated in the positions of the red vertical line. Thus, increasing the g value will increase truncation errors for the same mantissa bitwidth. In this paper, we set g at 16 unless otherwise stated.

D. Stochastic Rounding of Gradients in Gradient Descent

In this section, we present analysis and illustrations on the workings of stochastic rounding (SR) during gradient descent for DNN training. The analysis shows that for low-precision BFP with small mantissa bitwidth m, applying stochastic rounding to gradients, as illustrated in Figure 4c, can minimize

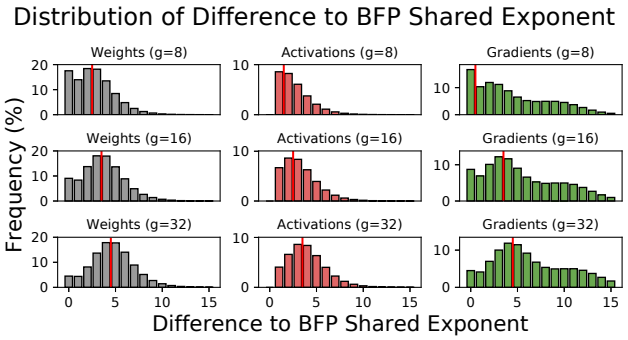

Fig. 7. The distribution of the difference between a BFP shared exponent (for three group sizes g = 8, 16 and 32) and exponents of all other values in the group before mantissa alignment for ResNet-18 at layer 10. A larger difference leads to an increased quantization error (truncation error); see Figure 4b.

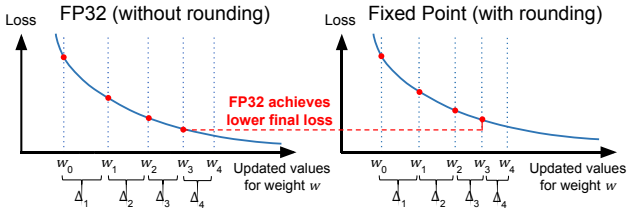

Fig. 8. (left) Weight w is updated over 4 iterations using gradients $\Delta_1, \Delta_2, \Delta_3, \Delta_4$ computed in FP32 leading to a decrease in loss. (right) Suppose that rounding of weight gradients biases rounding each gradient down to smaller values and thus smaller Δ values. Then, these roundings lead to higher loss over multiple iterations.

the impact of rounding on gradient descent performance. As noted earlier, the general idea of stochastic rounding appeared in the literature as early as the 1950s (see, e.g., [37]). The technique gains new perspectives in the context of supporting low-precision gradient descent computation, as described in this section.

Let E be the training loss of a DNN (e.g., E could be cross-entropy loss). We consider the use of the stochastic gradient descent (SGD) optimizer to minimize E over multiple training iterations. We use w to denote any learnable parameter, such as a filter weight, in the neural network. For each iteration i, the backpropagation algorithm computes the partial derivative $\nabla_i = \frac{\partial E}{\partial w_i}$ and updates w with the following rule:

$$w_{i+1} = w_i - \eta \nabla_i \quad \text{or} \quad \Delta_i = -\eta \nabla_i$$

where η is the learning rate and $\Delta_i = w_{i+1} - w_i$. For illustration simplicity, we assume in this analysis that the learning rate η is 1. We refer to a collection of multiple iterations over all training data as an epoch.

In Figure 8 (left), we illustrate four iterations of updating w from its initial value w_0 to w_1, w_2, w_3 and w_4, using infinite precision arithmetics without rounding. We use FP32 as a proxy for such arithmetics, and contrast it to low precision fixed point arithmetics (e.g., INT8) with rounding.

1) Impact of rounding on weight updates: In Figure 8 (right), we consider the scenario when we perform training iterations using fixed point gradients quantized from FP32. The diagram

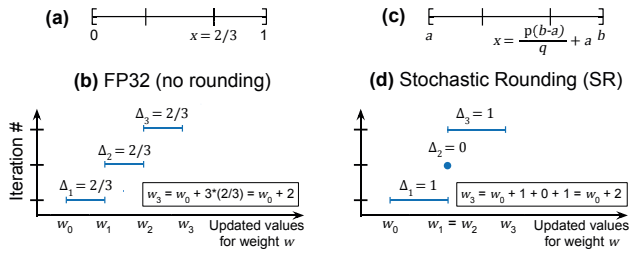

Fig. 9. (a) Gradient $x = 2/3$ in quantization interval $[0,1]$. (b) In FP32, each iteration uses FP32 gradient Δ to update w. (c) General quantization interval $[a,b]$ under SR. (d) SR is expected to increment w the same amount as in (b) by rounding up to 1 twice and down to 0 once.

illustrates that if the gradient at each of the four iterations is rounded down, then the total weight increments $\Delta_1+\Delta_2+\Delta_3+\Delta_4$ will be reduced leading to a higher loss, compared to the FP32 case without rounding. This is because the gradient ∇ at each iteration (red dot), is rounded down to a smaller value, causing a smaller weight increment Δ.

2) Use of stochastic rounding to minimize impact of rounding on weight updates: We use SR to minimize the impact of rounding on weight updates and the corresponding reduction on loss. Suppose that the input value x is in a quantization interval $[a,b]$. Under stochastic rounding, x is rounded down to a or up to b with probability $(b-x)/(b-a)$ or $(x-a)/(b-a)$, respectively. Note that the two probabilities sum to 1, as expected. By scaling and shifting x, we may assume that the quantization interval is $[0,1]$. Therefore, we round x to either 0 or 1 depending on which of the two values is the nearest to $x+r$, where r is a pseudo-random number (noise) in $(-0.5,+0.5)$. Figure 4c illustrates adding such a random noise r to x.

To illustrate how SR is applied to weight gradients, we first consider a simple case where we quantize a gradient x $(=2/3)$ in a quantization interval $[0,1]$, as depicted in Figure 9a. Under SR, x is rounded to 0 and 1 with probability $1/3$ and $2/3$, respectively, reflecting the distance of x to each endpoint. Note that over 3 iterations, x is expected to round down to 0 once and round up to 1 twice. Figure 9d illustrates that x is rounded to 1, 0 and 1 for iteration 1, 2 and 3, respectively. We note that SR increments w_0 by the same amount (i.e., in Figure 9d $\Delta_1+\Delta_2+\Delta_3=2$) towards computing to w_3, as the FP32 case, as depicted in Figure 9b.

We now consider a general case, where we round a gradient x in a quantization interval $[a,b]$, as depicted in Figure 9c. In this case, we express the weight gradient x as $x = p(b-a)/q+a$ for some p and q with $0 \le p < q$. (Note that if $a=0$, $b=1$, and $p=2$, $q=3$, then Figure 9c depicts the scenario of Figure 9a.) As noted earlier, under SR, x is rounded to a and b with probability $(b-x)/(b-a)$ and $(x-a)/(b-a)$, respectively. Thus, each iteration is expected to increment the weight value by $a\times(b-x)/(b-a)+b\times(x-a)/(b-a)=x$, which is the same weight increment under FP32 without rounding. The q value in Figure 9c specifies the precision that SR uses itself. For example, for Figure 4c, $q=8$ since we add in 3 stochastic noise bits before rounding, and $2^3=8$. We find that 8 bits of stochastic noise is sufficient for stochastic rounding to work well. When dealing with small mantissa bitwidths (e.g., 4-8), there is no benefit in using high precision noise (e.g., FP32).

We summarize the above derivation with the following lemma, which captures an objective of stochastic rounding.

Lemma 1: Suppose that x is an input value in a quantization interval $[a,b]$. Let $SR(x)$ be the rounded value of x under stochastic rounding. Then the expected value for $SR(x)$ is x.

In the following, we show that a weight w updated by iterations of gradient descent under stochastic rounding has an expected value equal to the weight updated under no rounding. For presentation simplicity, consider three iterations of gradient descent, starting at an initial weight value w_0, that compute updated weight values w_1, w_2, and w_3. It is straightforward to see that the analysis extends to any number of iterations.

Suppose that the gradients g_1, g_2 and g_3 for the three iterations are close to each other and in the same quantization interval. In other words, these gradients g_i are close to some value g in the interval with high probability, and the expected value for g_i is g. With scaling and shifting, we may assume, without loss of generality, that the quantization interval is $[0,1]$ and that these gradients are centered around a value g smaller than 0.5, say, $g=0.3$. After three iterations under infinite precision arithmetics with no rounding, we note that the updated weight has an expected value of $w_0+3\times0.3$. The following theorem states that under finite precision arithmetics with rounding, stochastic rounding allows these iterations to achieve the same expected values for the updated weights.

Theorem 1: The weight updated by iterations of gradient descent under stochastic rounding has an expected value equal to the weight updated under no rounding.

Proof: Under stochastic rounding, the computed value w_1' for w_1 after the first iteration is:

$$w_1' = w_0 + SR_{[0,1]}(g_1)$$

where $SR_{[0,1]}$ is the stochastic rounding function for the quantization interval $[0,1]$. Similarly, for the computed values of w_2 and w_3 resulting from the next two iterations, we have:

$$\begin{aligned} w_2' &= w_1' + SR_{[0,1]}(g_2) \\ &= w_0 + SR_{[0,1]}(g_1) + SR_{[0,1]}(g_2) \end{aligned}$$

$$\begin{aligned} w_3' &= w_2' + SR_{[0,1]}(g_3) \\ &= w_0 + SR_{[0,1]}(g_1) + SR_{[0,1]}(g_2) + SR_{[0,1]}(g_3) \end{aligned}$$

By Lemma 1, the expected value for $SR_{[0,1]}(g_i)$ is g_i. Thus the expected value for w_3' is $w_0+g_1+g_2+g_3$, which in turn has the expected value of w_0+3g, which is $w_0+3\times0.3$ when g is 0.3.

IV. Fast Strategy for Training

In this section, we describe our FAST strategy for DNN training which varies the BFP precision of weights, activations, and gradients over the course of training. In Section IV-A, we provide motivation for progressively increasing the precision across both training iterations and DNN layers. Then, in Section IV-B, we propose an approach to adaptively increase the precision during training.

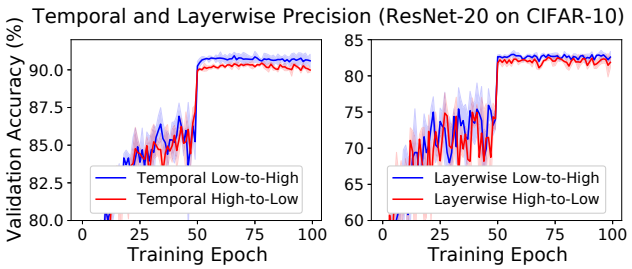

Fig. 10. (left) The Temporal Low-to-High scheme outperforms the Temporal High-to-Low scheme on validation accuracy. (right) The Layerwise Low-to-High scheme outperforms the Layerwise High-to-Low scheme. Each line is the mean of 3 runs, with the standard deviation shown as a shaded region.

A. Progressive Precision Changes over DNN Training

Previous literature has demonstrated that adding zero-mean Gaussian noise to the weight gradient ∇W can reduce overfitting and improve the convergence of DNN training [38]. They show that decreasing the variance of the noise over iterations achieves better performance than using fixed Gaussian noise throughout training. We hypothesize that a similar effect can be achieved by adjusting the BFP precision of weights, activations, and gradients from low to high precisions over training.

To test this hypothesis, we compare two training schemes (using ResNet-20 on CIFAR-10) that use different strategies for switching the DNN training precision over time. In the Temporal High-to-Low scheme, we use FP32 for weights, activations, and gradients for the first half of training, and low-precision BFP with a mantissa bitwidth of 3 and group size of 16 for the second half of training. For the Temporal Low-to-High scheme, we adopt the opposite approach by using low-precision BFP in the first half of training and FP32 in the second half of training. Figure 10 (left) shows the test accuracy of these two schemes over the training process. The Low-to-High scheme achieves a higher performance, which indicates that training is more amenable to low-precision BFP in the early stages.

Additionally, during the backward pass of the training, the BFP quantization error for the data gradient ∇O will be propagated from the later layers to the early layers, so it will have a greater impact on the early layers than later layers. We perform another experiment to show this impact by comparing against two training schemes. In the Layerwise High-to-Low scheme, we use FP32 precision for the first ten layers (half of ResNet-20), and low-precision BFP with a mantissa bitwidth of 3 and group size of 16 for later ten layers. For the Layerwise Low-to-High scheme, we apply the opposite precision setting by switching the training precision between the first and second half of the DNN layers. To eliminate the impact on the architectural difference, we change the structure of ResNet-20 so that the first and the second halves have the same weight filter layout. The results shown in Figure 10 (right) indicate that applying low precision in the early layers works better than later layers.

Algorithm 1: Adaptive FAST DNN Training

Input: I is the total number of training iterations.
L is the total number of DNN layers.
A_l, W_l, G_l are the activation, weight and gradient tensor of layer l, respectively.
$\varepsilon(l,i)$ is a threshold to determine the BFP precision.
$BFP(X,m)$ is a BFP quantization function that returns X under BFP with an m-bit mantissa.
Output: X_q represents the BFP-quantized X.

```
for i ← 1 to I do
    for l ← 1 to L do
        for X ∈ [A_l, W_l, G_l] do
            Compute the relative improvement r(X) for X.
            if r(X) < ε(l,i) then
                Set X_q = BFP(X, 2).
            else
                Set X_q = BFP(X, 4).
```

B. FAST Adaptive Training

Based on the insight of the prior section, we propose an adaptive training strategy that progressively increase the BFP precision across both training iterations and layer depth. Algorithm 1 describes the mechanism of the FAST training algorithm. FAST supports two precision levels by representing the BFP mantissas with either 2 bits (low precision) or 4 bits (high precision). For a given FP tensor X, FAST first evaluates the relative improvement $r(X)$, defined by Equation 2, of using a 4-bit mantissa compared to a 2-bit BFP mantissa. If the relative improvement of using the higher precision setting is smaller than a threshold (i.e., $r(X) < \varepsilon$), then a 4-bit mantissa does not offer significant improvement over a 2-bit mantissa. However, if the relative improvement is larger than the threshold, then using a 4-bit mantissa will significantly reduce the quantization error compared to a 2-bit mantissa.

To allow for an incrementally increasing BFP precision across both layer depth and training iterations, the threshold $\varepsilon(l,i)$ is set to vary with the layer depth l and training iteration i based on the following equation:

$$\varepsilon(l,i) = \alpha - \beta\frac{i}{I} - \beta\frac{l}{L} \tag{1}$$

where I and L are the total number of training iterations and DNN layers, respectively. α and β are the hyperparameters that specify the offset and the slope of the threshold function. Equation 1 sets $\varepsilon(l,i)$ to decrease gradually with both training iteration and layer depth, so that higher precisions will be used as the training iteration and layer depth grow. We define the relative improvement $r(X)$ of using higher-precision BFP ($m=4$) compared to low-precision BFP ($m=2$) as follows:

$$r(X) = \frac{\sum_n |BFP(X_n,4) - BFP(X_n,2)|}{\sum_n |BFP(X_n,2)|} \tag{2}$$

where X_n denotes the nth element of X, and $BFP(X,m)$ represents the quantized X_n with an m-bit mantissa. The numerator of $r(X)$ reflects the total difference between the BFP values with high-precision and low-precision mantissas across each element of X. These difference is normalized by the summation of magnitudes of the BFP-quantized values

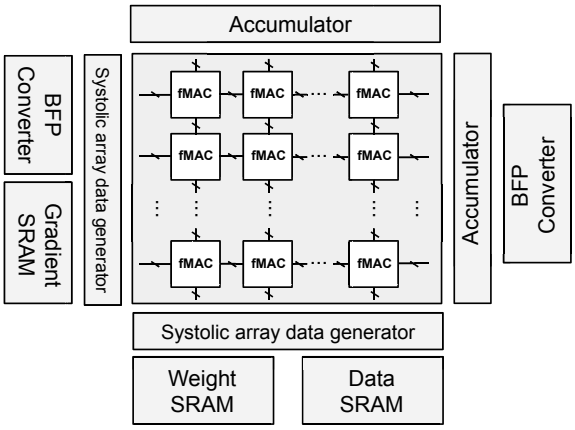

Fig. 11. Major components of the FAST system.

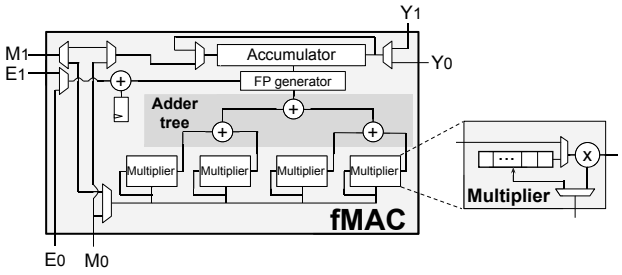

Fig. 12. The design of FAST MAC (fMAC).

so that the scale of $r(X)$ will be consistent across different training iteration and DNN layers. Finally, the numerator and denominator of $r(X)$ are computed by summing the BFP-quantized numbers across each element, which can be implemented with low hardware cost. The expensive division operation is only performed once between the two sums.

V. FAST System

The major components of the proposed FAST system are shown in Figure 11. We use a 2D systolic array (Section V-A) to perform the matrix multiplications for both the forward and backward passes of DNN training. The systolic array contains systolic cells of FAST MAC (fMAC), discussed in Section V-B, which support variable precision BFP. The memory subsystem has three SRAMs used to store weights, activations, and gradients, respectively. When performing matrix multiplication, the systolic array data generator is used to skew input for data synchronization in the systolic array. The accumulator is used to buffer partial accumulations across multiple tiles (for matrices that are larger than the systolic array). The output of the accumulator is passed to the BFP converter (Section V-C), which converts groups of FP values into BFP groups.

A. Systolic Array Operations

To support matrix transposition required during the backward pass of training (see Figure 3), we have designed a systolic array that can perform matrix multiplication involving a transposed matrix operand without explicit transposition. This allows for no extra data copying and thus reduces the implementation overhead of the matrix transposition operation. Our approach differs from the TPUv2 [39], which has a dedicated Transpose Permute Unit for permuting matrices loaded from High-Bandwidth Memory (HBM).

Figure 13 illustrates how this systolic array operates for each of the forward and backward pass matrix operations given in Figure 3. For clarity, we show each systolic cell with single INT values instead of a BFP group. To compute output O for each layer (Figure 13a), the weights W are first pre-stored in systolic cells. Then, the activation A enters the systolic array from bottom and the output O exits the systolic array from the right side (refer to Figure 3). During the backward pass, to compute the activation gradients ∇A (Figure 13b), W is also pre-stored in the systolic array. However, unlike the forward pass, with A entering from below, the output gradients ∇O enter the systolic array from left and the activation gradients ∇A are produced at the top of systolic array. By changing the side that input enters the systolic array while keeping the orientation of W fixed, we can compute $\nabla O \times W^T = \nabla A$ without explicitly transposing W. To summarize, if the input comes from below it is multiplied with the rows of the systolic array and if it comes from the left it is multiplied by the columns of the systolic array. Thus, matrix transposition is supported simply by changing the direction input is passed to the systolic array.

Finally, to compute the weight gradients ∇W (Figure 13c), the systolic array is reconfigured to be accumulation stationary. During the computation, the input activation A and output gradient ∇O will enter the systolic array from left and below, respectively, and the weight gradients ∇W are computed and accumulated within each systolic cell. At the end of this computation, the accumulated gradient in each systolic cell will sum with the weight W to generate the updated weight W', which is then stored back in the weight SRAM. For an optimizer like Adam [30], additional hardware is required to compute the first and second moments for the weight updates.

B. Design of FAST MAC for BFP

Each cell in the FAST systolic array implements a fMAC, which performs the DP between two BFP groups. Figure 12 shows the design of a fMAC for a group size $g = 4$. A DP consists of fixed-point multiplications between each pair of BFP mantissas for values in the groups, which are performed by the multipliers in the fMAC. The output generated by each multiplier is summed using the adder tree. The FP generator takes the fixed point summation from the adder tree to create the FP mantissa. The shared exponents of the two BFP groups are added together to create the FP exponent (refer to Figure 5). The resulting FP value is added to the FP accumulator, which stores the partial result spanning across multiple BFP groups.

Additionally, the fMAC can be reconfigured to support the different operations of DNN training described in Section V-A which may require matrix transposition. To compute the output O during the forward pass (Figure 13a), the BFP shared exponent and mantissas of W are first pre-stored in the fMAC using the E_0 and M_0 ports, respectively (Figure 12). Then, the

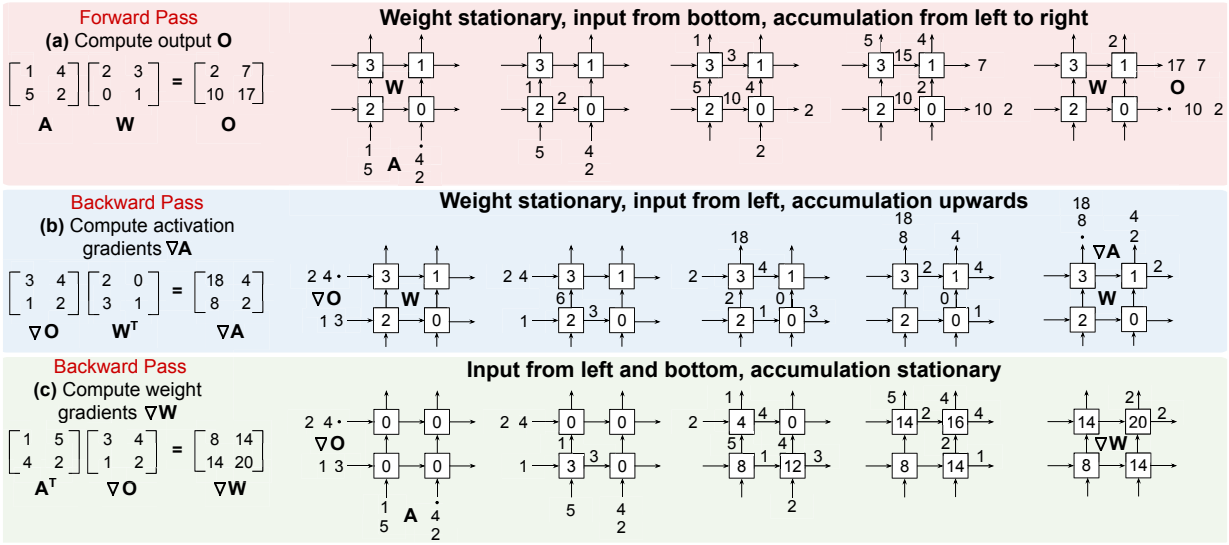

Fig. 13. Our systolic array can supports matrix multiplications required for both the forward and backward pass of DNN training. The matrix transposition required for matrix multiplications in the backward pass shown in (b) and (c) are handled by changing the side which the input enters the systolic array. Refer to Figure 3 for matrix notation.

activation A enters the fMAC via the same E_0 and M_0 ports to perform the DP computation. The output O generated by this computation exits to the right neighbor via output port Y_0. To compute the activation gradient ∇A during the backward pass (Figure 13b), W is pre-stored in the same fashion, and the BFP output gradients ∇O are passed into the multipliers via the E_1 and M_1 input ports, with the output ∇A exiting to the neighbor above via Y_1. Finally, to compute the weight gradients ∇W (Figure 13c), the output gradients ∇O and input activation A enter the fMAC using the ports M_1, E_1 and M_0, E_0, respectively. The accumulator output ∇W then loops back to be summed with the pre-stored FP weights W.

To support BFP DP with variable precision, each DP is processed in 2-bit mantissa chunks as shown in Figure 14. Here, the mantissas bitwidths for two operands (X and Y) are 4 bits and 2 bits, respectively. In the first round, the fMAC computes the dot product between Y and the first 2-bit chunk of the X (X_1). The partial accumulation result is then buffered for subsequent processing. In the second round, the fMAC computes the dot product between Y and the second 2-bit chunk of X (X_2) in order to finish the DP computation. More iterations are required for higher mantissa bitwidth. For example, multiplying a pair of BFP numbers with 4-bit and 4-bit mantissas translates to $\frac{4}{2} \times \frac{4}{2} = 4$ rounds. To account for the difference in exponent magnitude between two chunks, the BFP exponent of the second 2-bit chunk (X_2) is decremented by two. Note that this decrement is performed by the BFP converter when it generates each 2-bit chunk, and therefore fMAC is agnostic to these exponent difference across chunks.

C. BFP Converter

The BFP converter, shown in Figure 15, takes a group of FP values and converts them into BFP following the process outlined earlier in Figure 4. The comparator consists of compare

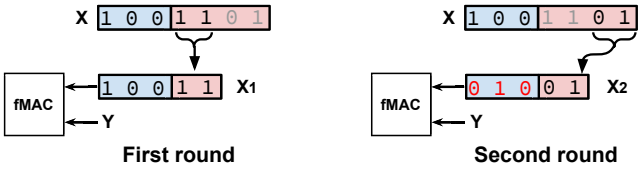

Fig. 14. Variable precision fMAC operations.

and forward (C&F) blocks arranged in a tree structure. Each C&F block takes a pair of FP exponents and forwards the larger exponent to the next tree level. The largest exponent will be output and used as the shared exponent (Figure 4a). Then, a group of subtractors calculate the differences between the shared exponent and each exponent in the group. The shift blocks, which are implemented using Barrel shifters [40], perform right shifts on each FP mantissa based on the exponent difference for each value (Figure 4b). Then, to perform stochastic rounding (Figure 4c), a group of 8-bit random binary streams produced by the linear feedback shift register (LFSR) are summed with the mantissas. Empirically, we find that 8 bits provides sufficiently precise randomness for stochastic rounding. Finally, the low-order bits of the BFP mantissas are truncated (Figure 4d). The BFP exponents and BFP mantissas will also be delivered to the improvement computation block which computes the relative improvement as defined in Equation 2.

D. Memory Layout for BFP Values

We have developed an efficient storage format for variable precision BFP, where the shared exponent and BFP mantissas are stored separately. Figure 16 provides an example for m = 4 and g = 2. The 2-bit chunks across all the mantissas in a group are saved in the same memory entry for efficient access during DP computation. The first 2-bit chunks of each BFP mantissa (Figure 16a) are stored together in the same memory

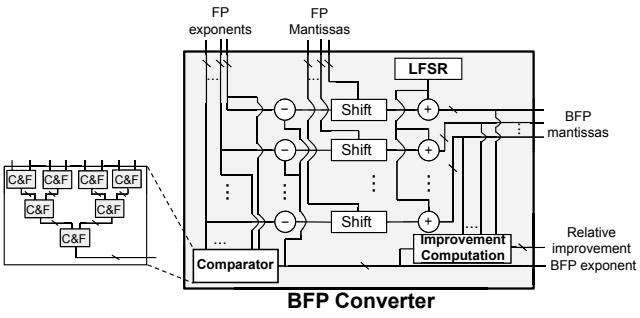

Fig. 15. Design of BFP converter which converts a group of FP values into BFP values.

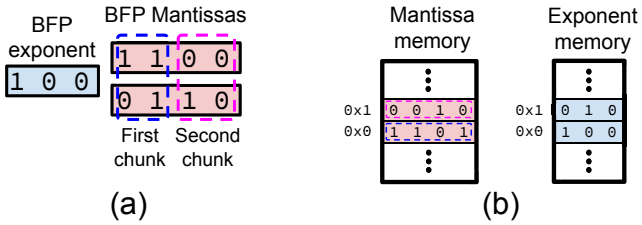

Fig. 16. The BFP memory layout for $g = 2$ and $m = 4$. Each mantissa is divided into 2-bit chunks to support variable precision multiplications. Sign bits are not shown.

entry (Figure 16b), followed by the second 2-bit chunks which are saved in the next memory entry.

Under this storage scheme, each BFP group will be represented by $\lceil \frac{m}{2} \rceil \times (e + g \times 3)$ bits, where e is the bitwidth of the BFP exponent, g is the group size, $\lceil \frac{m}{2} \rceil$ is the number of 2-bit chunks in an m-bit mantissa. An additional bit is required per mantissa to represent the sign, leading to 3 bits per mantissa. In our hardware system, e and g are set to be 3 and 16, respectively, and m is 2 or 4 based on the current precision. This leads to an average of 3.2 ($m = 2$) and 6.4 ($m = 4$) bits to store each value, which significantly reduces the storage overhead compared with other formats we evaluate in Section VI. All outputs from the BFP converter (Figure 15) are stored in BFP with 4-bit mantissas divided into two 2-bit chunks. When Algorithm 1 selects the 2-bit mantissa, the low-order 2-bit chunk is discarded.

E. Training Workflow

Figure 17 summarizes the overall workflow of DNN training using the FAST system. During the forward pass (Figure 17a), the filter weights W in BFP format are first loaded and saved into each fMAC (step 1). Next, the BFP activations A are loaded from data SRAM into the systolic array (step 2). Each systolic cell (fMAC) performs a partial DP for two BFP groups, followed by FP accumulation spanning across many BFP groups. The output O is then delivered to the BFP converter (step 3), which converts these values back into BFP format and stores them in the data SRAM for subsequent processing (step 4). Additionally, the activations A must also be kept in the data SRAM for the backward pass (Figure 17b-c).

To compute the activation gradients ∇A (Figure 17b), the weights W are again pre-stored into systolic array (step 1).

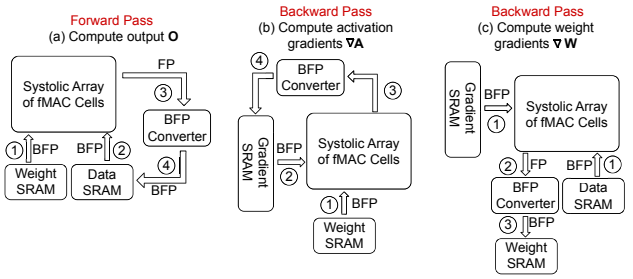

Fig. 17. Workflow during each training iteration. The systolic array operates according to Figure 13.

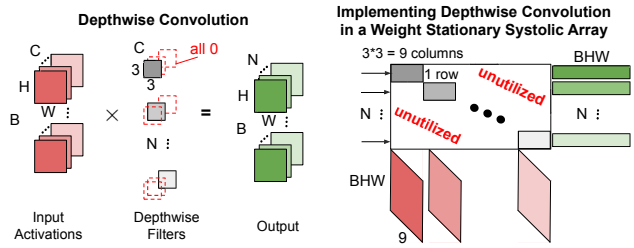

Fig. 18. (left) Each filer in depthwise convolution may have only a single active (nonzero) channel out of C channels. The zero channels are depicted in dotted red. (right) Mapping depthwise convolution into a weight stationary systolic array as used in FAST. The sparse nature of the depthwise filters leads to underutilization of the systolic array. As with other convolution layers, im2col [42] is used to convert depthwise convolution to matrix multiplication.

Then, the output gradients ∇O are delivered to the systolic array from the left (step 2). The results ∇A are produced at the top of systolic array and are converted into BFP format (step 3) before being saved into the gradient SRAM (step 4). To compute ∇W (Figure 17c), the input activation A and the output gradient ∇O are delivered to the systolic array concurrently (step 2). The results ∇W are produced within each systolic cell and then they are used to generate the updated weight W'. Finally, the updated weights W' are converted into BFP and stored in weight SRAM (step 3).

F. Mapping Depthwise Convolution to Systolic Arrays

Some CNNs using depthwise separable convolution, such as MobileNet-v2 [41] used in our evaluation in Section VI, approximate conventional convolution with filters of shape $C \times 3 \times 3$ with two layers: a depthwise convolution layer (filter shape of $1 \times 3 \times 3$) and a pointwise convolution layer (filter shape of $C \times 1 \times 1$). While pointwise convolution maps directly onto the systolic array used in FAST, the depthwise layer has significantly fewer parameters, making it more challenging to implement efficiently. Figure 18 shows the approach taken in this paper, which is to naively convert depthwise convolution into sparse matrix multiplication and handle it the same way as all other convolution layers as discussed in Section V-A.

Since depthwise convolution layers accounts for roughly 10% of the total computation in MobileNet-v2, we did not investigate further how to improve their efficiency in FAST. One possibility would be to add an additional computation mode to FAST for depthwise convolution. For instance, RiSA [43]

TABLE II
THE VALIDATION ACCURACY (CNNS), TEST BLEU (TRANSFORMERS) AND TEST MAP (YOLOV2) FOR NUMBER FORMATS OUTLINED IN FIGURE 2.

Model	FP32	bfloat16	Nvidia MP	INT8	INT12	MSFP-12	LowBFP	MidBFP	HighBFP	HFP8	FAST
ResNet-18	68.60	68.55	68.57	65.53	68.51	68.13	63.10	68.10	68.57	68.53	68.52
ResNet-50	75.17	75.12	75.16	71.01	75.03	74.79	72.10	73.98	75.13	75.07	75.11
MobileNet-v2	68.27	68.22	68.28	65.97	68.16	68.11	64.42	66.93	68.20	68.11	68.17
VGG-16	69.74	69.71	69.70	64.50	69.33	69.32	64.10	69.08	69.79	69.62	69.78
Transformer	35.41	35.39	35.42	29.18	35.27	35.33	34.22	35.40	35.43	35.38	35.40
YOLOv2	73.36	73.32	73.35	61.12	73.07	72.93	65.37	71.04	73.30	72.88	73.28

TABLE III
AREA AND POWER BREAKDOWN OF THE FAST SYSTEM ON ASIC.

Component	Area	Power
Systolic array	47.79%	15.61 W
BFP converter	4.56%	1.77 W
Accumulator	6.63%	2.19 W
Systolic array data generator	0.68%	0.69 W
Memory subsystem	40.34%	3.37 W

adds additional connections to a conventional systolic array to efficiently support both matrix multiplication and depthwise convolution.

VI. TRAINING EVALUATION OF FAST

In this section, we evaluate FAST's training performance for DNNs. In Section VI-A, we visualize the FAST precision adaptation over the course of training to show how FAST is able to achieve faster training time by staying in a low-resolution regime for a significant portion of training. Next, in section VI-B, we compare the accuracy performances of DNNs trained under BFP against other commonly used FP and INT formats. We also compare against three fixed BFP settings that do not change over the course of training: LowBFP uses (e=3, m=2) for all DNN weights, data, and activations, MidBFP uses (e=3, m=3), and HighBFP uses (e=3, m=4). Finally, in Section VI-C, we evaluate the performance of fixed BFP settings to show the relative advantage of using FAST.

We evaluate FAST on four CNNs: ResNet-18 [44], ResNet-50 [44], MobileNet-v2 [41], and VGG-16 [45]. All the CNNs are trained with ImageNet for 60 epochs (120000 iterations). We use the hyperparameter settings from the PyTorch website [46]. For Transformers [47], we use the 12-layer model with 12 heads and a hidden size of 768. The batch size is set to 16. We train on the IWSLT14 German-English dataset [6] for 150 epochs. Finally, YOLOv2 [7] is trained on the PASCAL VOC2012 [8] dataset with 120 epochs using a batch size of 64. We apply the SGD optimizer with an initial learning rate of 10^{-3}, dividing it by 10 at 60 and 90 epochs. The weight decay and momentum are set to 0.0005 and 0.9. The α and β FAST hyperparameters in Equation 1 are set to 0.6 and 0.3 for all the DNNs.

During training, all networks use FP16 input for normalization and activation layers and perform elementwise functions on intermediate FP32 values (as used by Nvidia MP [27]). Additional specialized hardware units are generally used in conjunction with systolic arrays for normalization and activation [39]. However, the FAST system reported in this paper does not support these layers and only implements the convolution and fully connected layers.

A. FAST Precision Adaptation

In this section, we visualize the precision changes during the training of a DNN using the FAST-Adaptive algorithm (Algorithm 1). Since the weights W, activations A, and gradients G independently determine their BFP precision, there are $2^3 = 8$ possible precision settings per layer using two different BFP resolutions ($m = 2$ and $m = 4$). In the figure, we have ordered these settings based on their computational costs when deployed in the FAST system (discussed next in Section VII). For instance, (W, A, G) of (4, 2, 2) has a slightly lower computational cost than (2, 2, 4) due to how the gradients are used multiple times during the backward pass (see Figure 3). We sample the BFP precisions for a batch of training samples during the backward pass. Figure 19 shows the BFP precisions of 5 layers in ResNet-18 on ImageNet and YOLOv2 on PASCAL VOC2012 change over the course of training under FAST. As expected, we observe that the BFP precision grows across both layer depths and training iterations.

B. Comparing Number Formats

Table II shows the validation accuracies for different DNN models trained using a wide range of number formats. The IEEE 754 32-bit FP (FP32) generally achieves the best performance (accuracy or BLEU) across all models. We find that bfloat16, Nvidia Mixed-Precision (MP), MSFP-12 and HFP8 are able to achieve similar performance as the baseline FP32 model for all DNNs. Additionally, the HighBFP setting is also able to achieve comparable accuracy. HighBFP with m = 4 represents a substantial saving compared to FP32 with m = 23. The LowBPF and MidBFP settings with m = 2 and m = 3 loss $4-5\%$ and $1-2\%$ in accuracy across all CNNs compared to the FP32 baseline. The INT-8 setting has an even larger reduction in accuracy, losing 4-6% compared to the baseline, even though it has more mantissa bits over HighBFP. For fixed point to achieve a similar level of performance as the baseline FP model requires an INT-12 with 11 mantissa bits. As we note in Section VII, fixed point multipliers used to perform this computation incur cost quadratically with the mantissa bitwidth, making large mantissa bitwidths costly to implement. By comparison, our FAST-Adaptive approach can achieve a comparable performance to FP32 across all the DNNs.

C. BFP Hyperparameter Sensitivity

In this section, we investigate the impact of the group size g and mantissa bitwidth m on the DNN training accuracy.

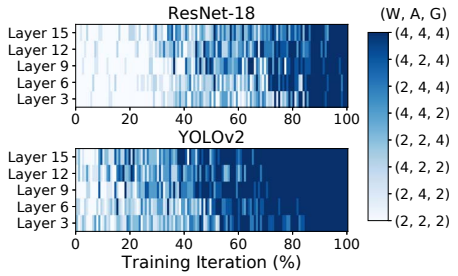

Fig. 19. FAST progressively increases the BFP precision. The different weight W, activation A, and gradient G settings are ordered by their computational complexity when implemented in the FAST system.

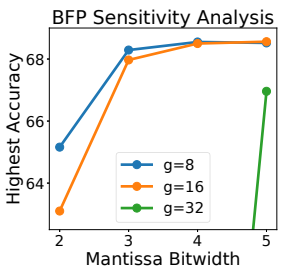

Fig. 20. BFP sensitivity analysis (ResNet-18 on ImageNet).

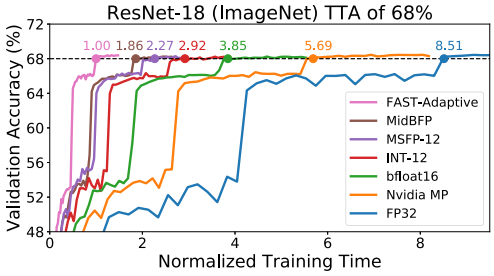

Fig. 21. The validation accuracy curves and TTA values for achieving the 68% accuracy for ImageNet (dotted line).

Figure 20 shows the validation accuracy on ResNet-18 for different BFP configurations settings. The three curves represent different group size configurations (i.e., $g = 8$, $g = 16$, and $g = 32$) and the x-axis corresponds to varying the number of mantissa bits (i.e., $m = 2$, $m = 3$, $m = 4$ and $m = 5$) for each group size.

For a given mantissa bitwidth, a smaller group size (e.g., $g = 8$) is generally able to achieve a higher accuracy than a larger group size (e.g., $g = 32$). This is because a smaller group size has less variations across exponents within a group (Figure 7), resulting in a lower quantization error and consequently higher accuracy, under the same mantissa bitwidth. However, a smaller group size has some additional implementation overhead due to more FP exponent additions as each shared exponent spans fewer elements in a smaller group. Moreover, smaller groups have smaller DP computation savings with BFP (refer to Section III-B). Overall, we observe that a group size of $g = 16$ with $m = 4$ produces the optimal performance, and we use this setting as our baseline for FAST training.

VII. Hardware Evaluation of FAST

In this section, we evaluate the hardware performance of the FAST system described in Section V. We have synthesized our system using the Synopsys Design Compiler [48] with 45nm NanGate Open Cell Library [49] and CACTI [50]. CACTI is used to simulate the performance of the memory subsystem and Synopsys Design Compiler is used for all other subsystems shown in Figure 11. For our FPGA evaluation, we use a Xilinx VC707 FPGA evaluation board. The FAST system contains a 256×64 systolic array of fMAC cells. Gradient SRAM, weight SRAM and data SRAM each consist of 128 16kB memory banks. The FAST system runs at 500 MHz.

Table III summarizes the area and power breakdowns of FAST on ASIC. The majority of the area is used by the systolic array and memory subsystem, which take 47.79% and 40.34%, followed by the accumulator (6.63%), BFP converter (4.56%) and systolic data generator (0.68%). In terms of power, the systolic array has the largest power consumption (15.61W), followed by the memory subsystem (3.37W), accumulator (2.19W), BFP converter (1.77W) and systolic data generator (0.69W).

A. Evaluation of fMAC

We evaluate the efficiency of our fMAC design by comparing it against FP and INT MAC designs. For FP MACs, we implement them with bfloat16, FP16 and HFP8. FP16 is used by Nvidia MP. An FP MAC performs multiply-accumulate operations between two FP numbers followed by a 32-bit FP accumulation. For INT MACs, we implement them with 8-bit (INT-8) and 12-bit (INT-12) variants. Refer to Figure 2 for details on each number format. Two floating point formats are used by HFP8 during training: 4-bit exponent/3-bit mantissa for the forward pass and 5-bit exponent/2-bit mantissa for the backward pass. For a hardware cost comparison to FAST, we implement an MAC that supports a 4-bit exponent/2-bit mantissa, so that the hardware cost is strictly less than either floating point format used by HFP8. Since a single fMAC performs a BFP DP across two groups of $g = 16$ numbers, we use $g = 16$ for all other MAC designs for a fair comparison.

Table IV provides ASIC and FPGA evaluations in terms of area and power consumption and FPGA resource consumption for all MAC designs. Area consumption is normalized by the area of our fMAC design. The fMAC achieves a superior area and power consumption compared to the other MAC designs. The main advantage of the fMAC over the INT MAC designs is the significantly reduced mantissa bitwidth, leading to a substantial reduction in cost of the fixed point multipliers. When comparing fMAC to the FP MACs, the expensive FP accumulator is amortized over the group for fMAC instead of between each pair of elements for FP MACs.

B. Training Speedup of FAST Strategies

In this section, we evaluate the performance of the FAST system by comparing against the other DNN training systems implemented with systolic arrays using different number formats. We configure each system for a given number format to have the same total area as our FAST system. Specifically, with the same area, we are able to fit a DNN training system with a systolic array of 245×245 HFP8 (4-bit exponent and 2-bit mantissa) MACs, 230×230 MSFP-12 MACs, 210×210 INT-12 MACs, 180×180 bfloat16 MACs and 150×150 FP16 MACs, respectively. Note that our FAST system contains a 256×64 fMAC systolic array, and each fMAC can perform multiply-accumulate operations for 16 BFP numbers within one

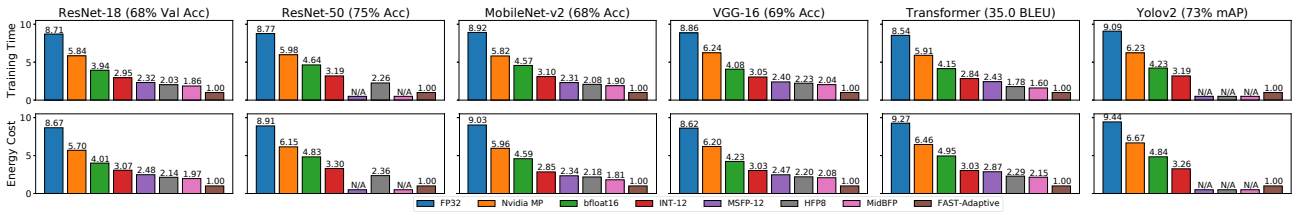

Fig. 22. The normalized training time and energy cost comparisons under different number formats, which are color coded as in Figure 21. N/A means the target accuracy is never reached for a given setting.

TABLE IV
ASIC AREA AND POWER COMPARISON AND FPGA RESOURCE CONSUMPTION FOR DIFFERENT MAC DESIGNS.

MAC Design	ASIC		FPGA	
	Area	Power	LUT	FF
fMAC	1×	0.885mW	269	140
16× INT-8	3.8×	2.241mW	498	195
16× HFP8	4.1×	2.406mW	527	220
16× INT-12	5.6×	2.920mW	730	273
16× bfloat16	9.6×	3.869mW	1305	684
16× FP16	10.6×	4.474mW	1514	753

cycle. The design for other major components (i.e., accumulator, number format converter, systolic array data generator and memory subsystem) of the baseline DNN systems are modified according to a given MAC design. For example, for bfloat16, a bfloat16 converter is used instead of a BFP converter. All designs run at a 500MHz clock frequency.

We use Time-to-Accuracy (TTA) [51] as the evaluation metric to compare different approaches. Figure 21 shows the TTA for ResNet-18 models trained under various number formats to achieve a validation accuracy of 68% on ImageNet. The training time is normalized by the FAST-Adaptive model which has the smallest training time while still achieving the target accuracy. Some settings that were unable to achieve 68% validation accuracy, such as INT8 and LowBFP, were omitted. The results are measured by performing a single round of forward pass and backward pass with an input mini-batch of size 256.

Generally, we see that FP32 is significantly slower than reduced/mixed precision formats such as bfloat16 and Nvidia MP. However, the floating point accumulations required for each MAC using these formats introduces overhead compared to fixed point of BFP formats. The MSFP-12 achieves the best performance of all prior work. Our proposed FAST schemes outperform MSFP-12 by more than 2× by using lower mantissa and exponent bitwidths and switching to a higher precision in the later stage of training.

The evaluation results are generated based on the computation required for all convolutional and fully connected layers. Normalization and Activation layers are not considered in the cost analysis of this paper. Prior work suggests that activations and batch normalization take less than 5% of total running time [52], and a small amount of power relative to the systolic array and other components [53], [54]. However, after significantly reducing the running time of matrix multiplication using FAST, these normalization and activation layers will begin to dominate the running time. For further improvements, we would need to focus on speeding up these operations.

Figure 22 depicts the normalized training time and energy cost for all evaluation DNNs to reach a target accuracy, mAP or BLEU. We note that performance trend and performance gain of FAST-Adaptive are consistent across models, with prior reduced/mixed precision formats outperforming FP32 by a factor of 2-3× and our proposed BFP formats achieving an additional 2-3× improvement.

VIII. CONCLUSION

The FAST system uses block floating point (BFP) to support low-precision arithmetics to reduce DNN training time, power consumption, and hardware requirements. With FAST, we exploit an observation that earlier layers and training iterations can afford larger error margins, making them amenable to efficient low-precision computation. We use high-precision BFP only in the later layers and in later iterations of training.

We empirically demonstrate a 2-6× speedup in training over prior work based on mixed-precision or BFP number systems while achieving similar accuracy. FAST's superior performance is due to our architectural choice of using the BFP number system, use of stochastic rounding in BFP, and modular fMAC design to support multiple precisions. The evaluation results show that FAST achieves more than 2× reduction in training time leading to a corresponding improvement in energy efficiency compared to the other reduced/mixed precision formats. This work shows that variable precison BFP with stochastic rounding offers a promising strategy in speeding up training and improving its efficiency.

As DNN training is now often distributed across multi-chip systems [39], [55], future work is to study how well FAST could scale in such a multi-chip deployment. One possible approach is to use a matrix multiplication partitioning and scheduling algorithm (e.g., CAKE [56]) to divide up the computation in an bandwidth-efficient way via reuse of data in on-chip local memory.

IX. ACKNOWLEDGEMENTS

This research was supported in part by the Air Force Research Laboratory under award number FA8750-18-1-0112, and the Defense Advanced Research Projects Agency under UCLA award number 0160GXA278 and MIT award number S5181. The authors thank the anonymous reviewers of HPCA 2022 for their helpful feedback.

REFERENCES

[1] "Bfloat16: The secret to high performance on cloud tpus," https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus, Google, accessed: 2021-03-29.

[2] "Accelerating ai training with nvidia tf32 tensor cores," https://developer.nvidia.com/blog/accelerating-ai-training-with-tf32-tensor-cores/, Nvidia, accessed: 2021-03-29.

[3] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Dover Publications, 1964.

[4] H. T. Kung, "Why systolic architectures?" *IEEE Computer*, vol. 15, pp. 37–46, 1982.

[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 248–255.

[6] "Iwslt2014 benchmark," https://paperswithcode.com/sota/machine-translation-on-iwslt2014-german.

[7] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[8] M. Everingham and J. Winn, "The pascal visual object classes challenge 2012 (voc2012) development kit," *Pattern Analysis, Statistical Modelling and Computational Learning, Tech. Rep*, vol. 8, p. 5, 2011.

[9] B. Darvish Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner, A. Forin, H. Zhu, T. Na, P. Patel, S. Che, L. Chand Koppaka, X. SONG, S. Som, K. Das, S. T, S. Reinhardt, S. Lanka, E. Chung, and D. Burger, "Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 10 271–10 281. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/747e32ab0fea7fbd2ad9ec03daa3f840-Paper.pdf

[10] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "Training dnns with hybrid block floating point," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 451–461.

[11] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," *arXiv preprint arXiv:1803.03635*, 2018.

[12] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[13] ——, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

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

[15] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

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

[17] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5456–5464.

[18] S. Kapur, A. Mishra, and D. Marr, "Low precision rnns: Quantizing rnns without losing accuracy," *arXiv preprint arXiv:1710.07706*, 2017.

[19] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," in *NeurIPS*, 2018, pp. 5151–5159.

[20] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[21] H. T. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," *24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[22] B. McDanel, S. Q. Zhang, H. T. Kung, and X. Dong, "Full-stack optimization for accelerating cnns using powers-of-two weights with fpga validation," *International Conference on Supercomputing*, 2019.

[23] O. Bilaniuk, S. Wagner, Y. Savaria, and J.-P. David, "Bit-slicing fpga accelerator for quantized neural networks," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2019, pp. 1–5.

[24] S. Q. Zhang, B. McDanel, H. Kung, and X. Dong, "Training for multi-resolution inference using reusable quantization terms," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 845–860.

[25] J.-H. Lin, T. Xing, R. Zhao, Z. Zhang, M. Srivastava, Z. Tu, and R. K. Gupta, "Binarized convolutional neural networks with separable filters for efficient hardware acceleration," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

[26] B. Chmiel, L. Ben-Uri, M. Shkolnik, E. Hoffer, R. Banner, and D. Soudry, "Neural gradients are near-lognormal: improved quantized and sparse training," 2020.

[27] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=r1gs9JgRZ

[28] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks," *Advances in neural information processing systems*, vol. 32, pp. 4900–4909, 2019.

[29] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale dnn processor for real-time ai," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 1–14. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00012

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[31] M. Mahmoud, I. Edo, A. H. Zadeh, O. M. Awad, G. Pekhimenko, J. Albericio, and A. Moshovos, "Tensordash: Exploiting sparsity to accelerate deep neural network training," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 781–795.

[32] J. Zhang, X. Chen, M. Song, and T. Li, "Eager pruning: algorithm and architecture support for fast training of deep neural networks," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 292–303.

[33] D. Yang, A. Ghasemazar, X. Ren, M. Golub, G. Lemieux, and M. Lis, "Procrustes: a dataflow and accelerator for sparse deep neural network training," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 711–724.

[34] S. Choi, J. Sim, M. Kang, Y. Choi, H. Kim, and L.-S. Kim, "An energy-efficient deep convolutional neural network training accelerator for in situ personalization on smart devices," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 10, pp. 2691–2702, 2020.

[35] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[36] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo, "7.7 lnpu: A 25.3 tflops/w sparse deep-neural-network learning processor with fine-grained mixed precision of fp8-fp16," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2019, pp. 142–144.

[37] G. Fobsythe, "Note on rounding-off errors," *SIAM Rev*, vol. 1, pp. 66–67, 1959.

[38] A. Neelakantan, L. Vilnis, Q. V. Le, I. Sutskever, L. Kaiser, K. Kurach, and J. Martens, "Adding gradient noise improves learning for very deep networks," *arXiv preprint arXiv:1511.06807*, 2015.

[39] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.

[40] M. R. Pillmeier, M. J. Schulte, and E. G. Walters III, "Design alternatives for barrel shifters," in *Advanced Signal Processing Algorithms, Architectures, and Implementations XII*, vol. 4791. International Society for Optics and Photonics, 2002, pp. 436–447.

[41] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings*

of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 4510–4520.

[42] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning." *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1410.html#ChetlurWVCTCS14

[43] H. Cho, "Risa: A reinforced systolic array for depthwise convolutions and embedded tensor reshaping," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, sep 2021. [Online]. Available: https://doi.org/10.1145/3476984

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

[45] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013.

[46] "Pytorch official implementation for imagenet," https://github.com/pytorch/examples/blob/master/imagenet/main.py.

[47] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[48] "Design compiler: Rtl synthesis," https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html.

[49] "Nangate freepdk45 open cell library," http://www.nangate.com/?page_id=2325.

[50] "Cacti: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model," https://github.com/HewlettPackard/cacti.

[51] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, "Dawnbench: An end-to-end deep learning benchmark and competition," *Training*, vol. 100, no. 101, p. 102, 2017.

[52] B. Fleischer, S. Shukla, M. Ziegler, J. Silberman, J. Oh, V. Srinivasan, J. Choi, S. Mueller, A. Agrawal, T. Babinsky, N. Cao, C.-Y. Chen, P. Chuang, T. Fox, G. Gristede, M. Guillorn, H. Haynie, M. Klaiber, D. Lee, S.-H. Lo, G. Maier, M. Scheuermann, S. Venkataramani, C. Vezyrtzis, N. Wang, F. Yee, C. Zhou, P.-F. Lu, B. Curran, L. Chang, and K. Gopalakrishnan, "A scalable multi- teraops deep learning processor core for ai trainina and inference," in *2018 IEEE Symposium on VLSI Circuits*, 2018, pp. 35–36.

[53] J. Oh, S. Lee, M. Kang, M. Ziegler, J. Silberman, A. Agrawal, S. Venkataramani, B. Fleischer, M. Guillorn, J. Choi, W. Wang, S. Mueller, S. Ben-Yehuda, J. Bonanno, N. Cao, R. Casatuta, C. Chen, M. Cohen, O. Erez, T. Fox, G. Gristede, H. Haynie, V. Ivanov, S. Koswatta, S. Lo, M. Lutz, G. Maier, A. Mesh, Y. Nustov, S. Rider, M. Schaal, M. Scheuermann, X. Sun, N. Wang, F. Yee, C. Zhou, V. Shah, B. Curran, V. Srinivasan, P. Lu, S. Shukla, K. Gopalakrishnan, and L. Chang, "A 3.0 tflops 0.62v scalable processor core for high compute utilization ai training and inference," in *2020 IEEE Symposium on VLSI Circuits, VLSI Circuits 2020 - Proceedings*, ser. IEEE Symposium on VLSI Circuits, Digest of Technical Papers. Institute of Electrical and Electronics Engineers Inc., Jun. 2020, publisher Copyright: © 2020 IEEE.; null ; Conference date: 16-06-2020 Through 19-06-2020.

[54] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[55] "Tesla dojo technology," https://tesla-cdn.thron.com/static/SBY4B9_tesla-dojo-technology_OPNZ0M.pdf?xseo=&response-content-disposition=inline%3Bfilename%3D%22tesla-dojo-technology.pdf%22, 2021.

[56] H. Kung, V. Natesh, and A. Sabot, "Cake: matrix multiplication using constant-bandwidth blocks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.