FlexBlock: A Flexible DNN Training Accelerator with Multi-Mode Block Floating Point Support

Seock-Hwan Noh*, Jahyun Koo*, Seunghyun Lee*, Jongse Park†, Jaeha Kung*‡

*DGIST, †KAIST, Republic of Korea

‡ Corresponding author

Abstract—Training deep neural networks (DNNs) is a computationally expensive job, which can take weeks or months even with high performance GPUs. As a remedy for this challenge, community has started exploring the use of more efficient data representations in the training process, e.g., block floating point (BFP). However, prior work on BFP-based DNN accelerators rely on a specific BFP representation making them less versatile. This paper builds upon an algorithmic observation that we can accelerate the training by leveraging multiple BFP precisions without compromising the finally achieved accuracy. Backed up by this algorithmic opportunity, we develop a flexible DNN training accelerator, dubbed FlexBlock, which supports three different BFP precision modes, possibly different among activation, weight, and gradient tensors. While several prior works proposed such multi-precision support for DNN accelerators, not only do they focus only on the inference, but also their core utilization is suboptimal at a fixed precision and specific layer types when the training is considered. Instead, FlexBlock is designed in such a way that high core utilization is achievable for i) various layer types, and ii) three BFP precisions by mapping data in a hierarchical manner to its compute units. We evaluate the effectiveness of FlexBlock architecture using well-known DNNs on CIFAR, ImageNet and WMT14 datasets. As a result, training in FlexBlock significantly improves the training speed by 1.5∼5.3× and the energy efficiency by 2.4∼7.0× on average compared to other training accelerators and incurs marginal accuracy loss compared to full-precision training.

I. INTRODUCTION

With the development of high-performance computing systems and ever-growing open source datasets, deep learning has advanced at a very rapid pace. Due to its accuracy improvement, many applications have started to utilize deep learning including computer vision, language modeling, autonomous driving, robotics, and even chip design [3], [6], [14], [41], [44], [51]. Moreover, many researchers have focused on reducing the model complexity of deep neural networks (DNNs) to move intelligence into mobile devices [16], [22], [36], [40], [59], [62], [70]. As different deep learning models are actively developed for a wide range of applications and domains, various layer types and precision levels are being used. Unfortunately, there is still a lack of training accelerators optimized for these various conditions with high efficiency.

Generally, deep neural networks are trained in IEEE single-precision format, i.e., $FP32$, to minimize the accuracy loss during the training on CPUs/GPUs. To increase the effective arithmetic and memory bandwidth during the training, one may reduce the precision in representing activations, weights, and/or gradients [12], [18], [43]. Micikevicius et al. pro-

posed mixed precision training [43], multiplying two inputs in IEEE half-precision ($FP16$), while accumulating the results in $FP32$, using Tensor Cores in NVIDIA GPUs. This approach doubles the effective memory bandwidth and achieves up to 2∼4× speed-up in DNN training. Instead, one may preserve the exponent bits of $FP32$ (8-bit) but truncate the mantissa bits to make 16-bit, i.e., $bfloat16$ [13]. There are several commercial DNN training accelerators that utilize $bfloat16$ to support wider numeric representations [12], [18].

Considering the overwhelming size of the recently developed DNNs, e.g., 469M parameters for AmoebaNet-A [50] and 175B parameters in GPT-3 [6], keeping all tensors in floating point representations would require huge memory footprint and significant training time. Recently, a block floating point (BFP) representation has been revived and applied in training DNNs to improve performance and energy efficiency [10], [31]. However, prior work on BFP-based DNN accelerators rely on a specific BFP representation, making them less versatile and offering limited opportunities for performance and efficiency gains. Moreover, DNNs for mobile environment are trained at a low precision which are suited for the hardware running at that specific precision, e.g., $INT8$ on Google Edge TPU [27]. As training such edge-optimized DNNs entails both low- and high-precision arithmetics, it further motivates the accelerator architecture with the multi-precision support on both fixed- and floating-point representations.

Unlocking the missing opportunities, we first propose a BFP-based training hardware, dubbed FlexBlock, which supports multiple BFP precision modes and layer types. FlexBlock is designed to support 4-bit, 8-bit and 16-bit (sign+mantissas) with 8-bit shared exponents[1]. With the hardware support, we empirically demonstrate the possibility of training DNNs even with 4-bit arithmetic ($FB12$) for computing feature maps and local gradients, while allowing 8-bit/16-bit arithmetic ($FB16/FB24$) for computing weight gradients. This aggressive precision scaling during the DNN training results in 5.3× speedup compared to the training in $bfloat16$ with negligible accuracy loss.

The main contributions can be summarized as follows:

1) **Multi-mode BFP support:** We develop a BFP-based DNN training accelerator supporting the multiple precision modes. A DNN model trained in $FB12$, $FB16$

---

[1]The basic FlexBlock formats are defined as $FB12$ (=FB4+8) for 4-bit, $FB16$ for 8-bit, and $FB24$ for 16-bit mantissas with 8-bit shared exponents.

① FW: forward pass (compute the loss) ② BW: backward pass (get local gradients) ③ WU: weight update (get weight gradients)

Fig. 1: Three important computational steps involved during the training of a convolutional layer.

or `FB24` can be executed on an accelerator supporting `bfloat16` [12], [18] or CPUs/GPUs with single-precision as they use the same exponent bits. In addition, we can train DNNs in `INT4`, `INT8` or `INT16` with a quantization scaling factor since the compute units of FlexBlock are mainly fixed-point arithmetic units.

2) **High core utilization:** We maximize the core utilization at all training steps and various layer types by proposing two design techniques: i) mapping tensor dimensions in a hierarchical manner to compute units, and ii) placing a separate reduction unit for depthwise operations.

3) **Low precision training:** We demonstrate the use case of FlexBlock that maximizes the energy efficiency of the training by using 4-bit arithmetics (`FB12`). We accomplish this by statically/dynamically selecting higher bit precisions when computing weight gradients.

II. BACKGROUND

A. Training Deep Neural Networks

To train DNN models, three important computational steps are required: i) computing the training loss (*forward pass*; FW), ii) computing local gradients (*backward pass*; BW), and iii) computing weight gradients (*weight update*; WU). As an example, Fig. 1 illustrates these steps for a convolutional (Conv) layer. We can easily extend the similar analysis to fully-connected (FC) layer as well. During the forward pass, 'C_i' input feature maps (fmaps) are convoluted with a set of weight kernels to generate 'C_o' output fmaps. After completing the forward pass, the total loss ($\mathcal{L}$) for a given mini-batch is computed. Then, the backward pass begins to backpropagate $\mathcal{L}$ to every layer in the network. In backpropagation, the same convolution operation is performed where the input becomes the local gradient $G_Y = \partial\mathcal{L}/\partial Y$ at layer 'l' and weight kernels are transposed and flipped. The output of this operation is the local gradient $G_X = \partial\mathcal{L}/\partial X$ for layer '$l-1$'. At each layer, the weight gradient $\partial\mathcal{L}/\partial W_{ck}$ is computed by performing a pairwise (and depthwise) full convolution between the local gradient $G_Y[k]$ and input fmap $X[c]$. The weight gradient is then used to update the weights after multiplying it with the learning rate.

B. Block Floating Point

As mentioned in Section I, DNNs are generally trained with `FP32`. The real value x_i in the floating point representation is expressed as

$$x_i = (-1)^{s_i} \cdot m_i \cdot 2^{e_i}, \quad (1)$$

where s_i is the sign, m_i is the mantissa, and e_i is the exponent of the number x_i. Block floating point (BFP) is a special form of the floating point representation, where a block of N numbers share an exponent corresponding to the number with the largest magnitude [10], [29]. Then, numbers within the block are represented as

$$\vec{x} = [x_1, x_2, \ldots, x_N] = [\hat{x_1}, \hat{x_2}, \ldots, \hat{x_N}] \cdot 2^{e_s} = \hat{\vec{x}} \cdot 2^{e_s}, \quad (2)$$

where $e_s = \lfloor log_2(max(|x_1|, \cdots, |x_N|)) \rfloor$ is the shared exponent of the block, and $\hat{x}_i = x_i \cdot 2^{-e_s}$ is the aligned number represented by only 'sign+mantissa'. With the BFP representation, therefore, we can perform a dot product in fixed-point arithmetic without the in-place alignment of intermediate results (cheaper in hardware). A dot product between two vectors $\vec{w}$ and $\vec{x}$ can be computed by

$$\vec{w} \cdot \vec{x} = (\hat{\vec{w}} \cdot \hat{\vec{x}}) \cdot 2^{e_w + e_x}, \quad (3)$$

where e_w and e_x are the shared exponents of $\vec{w}$ and $\vec{x}$, respectively. One of the goals of this work is to design hardware accelerator that supports various precision levels in computing $\hat{\vec{w}} \cdot \hat{\vec{x}}$ for the efficient DNN training.

C. Precision-Scalable MAC Array

Many research efforts have been made over the recent years to design precision-scalable MAC arrays that enable on-device DNN inference [7], [34], [35], [45], [46], [52], [54]–[56]. Earlier work use a simple data gating scheme to zero out operand(s) to minimize the dynamic power consumed by the MAC array [34], [35], [45], [56]. A bit-serial data fetching on weight tensors has been presented to allow fully-variable weight precision (*temporal scalability*) [35]. This temporal scalability has been extended to both operands, i.e., input and weight tensors, to simplify the computing logic [54]. However, the bit-serial approach consumes varying clock cycles depending on the precision level and requires more complex control logic. On the other hand, Shin et al. proposed to utilize sub-word parallelism on weight tensors [56]. A full-precision multiplier is built out of multiple sub-multipliers, which are always active (*spatial scalability*). To provide the sub-word parallelism on both operands (2D parallelism), a systolic array in which each processing engine consists of sixteen multipliers has been presented [55]. As pointed out by Camus et al. [7], the 2D sub-word parallelism shows the best energy efficiency when designing the precision-scalable MAC array.

III. MOTIVATION

In this paper, we aim to devise a multi-precision support accelerator architecture for DNN training. While the existing multi-precision architectures are exclusively designed for inference, one may think that the naïve adaptation of such architecture is sufficient for training. Thus, we first delve into the prior work and identify the limitations of existing architectures that we target to optimize in this work.

Limitation: One of the representative works on a precision-scalable MAC array is Bit Fusion [55]. Fig. 2(a) shows a fusion unit (FU) presented in [55] using the 2D sub-word parallelism. Each partial product in a 16b×16b multiplication is computed at a dedicated 4b×4b multiplier (some are color-coded). Since all accumulations within an FU need to be completed prior to passing the result to the next FU, the number of accumulated partial sums (psums) quadruples when there is 2× precision reduction on both operands (X and W). A simple motivational example on this issue is provided in Fig. 3(a-c). This may result in the underutilization of MACs limiting the speed-up expected by the precision scaling. In addition, Bit Fusion requires a significant number of shifters, e.g., ∼98k 4-bit shifters for the 64×64 array. To improve the power-efficiency, BitBlade [52] clusters multipliers with the identical shift length and reduces the number of shifters by 93.8% compared to Bit Fusion. Still, the number of accumulations increases at the same rate as Bit Fusion with precision scaling.

Solution: To mitigate this problem, a subset of multipliers are grouped together, as a processing unit (PU), to realize 1D sub-word parallelism on X (Fig. 2(b)). Across multiple PUs, i.e., four in FlexBlock, 1D sub-word parallelism on W is realized where psums from PUs are accumulated at the end depending on the precision mode. The advantage of splitting the 2D parallelism is more clear by looking at the example shown in Fig. 3(d-f). Instead of forcing all PUs to perform the same vector multiplication with a lengthy vector dimension, each accumulation path can be assigned to compute different output channels. With this hierarchical sub-word parallelism, the number of accumulated psums doubles even with the 2× precision reduction on both X and W (Fig. 3(d-f)).

Analysis: To quantitatively examine the implication of such architectural difference, we analyze the MAC utilization of FW, BW and WU steps on Bit Fusion, BitBlade, and our FlexBlock architectures, as we change the input and weight tensor precisions. For the analysis, we assume the training variants of Bit Fusion and BitBlade architectures attached with necessary `FP32` accumulators and BFP modules at the end of the MAC array. Fig. 4 reports that the MAC utilization is 76.5% for both Bit Fusion and BitBlade on training MobileNetV1 [19] when both input (X) and weight (W) tensors are 16-bit (denoted as X16W16). When we reduce the precision to 8-bit (X8W8), the utilization reduces by 13.8% on average. If we further reduce the precision from 8-bit to 4-bit (X4W4), additional 22.0% utilization drop is observed on average. This is because a much larger number of accumulations are required at a reduced precision to avoid

Fig. 2: An illustration of performing a 16b×16b multiplication using two precision-scalable multipliers: (a) a fusion unit (FU) in Bit Fusion [55] and (b) a processing unit (PU) in the proposed FlexBlock. Four PUs in FlexBlock split the 2D sub-word parallelism into two 1D sub-word parallelisms.

TABLE I: Examples of two and three-dimensional operations supported by FlexBlock

Modes	Operations
2D	Computing $\partial\mathcal{L}/\partial W$, Depthwise Conv, Dilated Conv, Up Conv
3D	General Conv, Pointwise Conv, FC (for both forward and backward pass)

wasting computing resources.

This MAC underutilization problem gets exacerbated when considering the weight gradient calculation, i.e., WU step, since the WU consists of a number of depthwise operations that require a small number of accumulations and thus utilize a small subset of MAC units. As the depthwise operations do not entail computations across multiple channels, we classify them as 2D operations in this paper. Table I summarizes 2D and 3D DNN operations supported by the FlexBlock core.

IV. DESIGN OF A FLEXBLOCK CORE

A. Hierarchical Design in FlexBlock

For the fine-grained control of hardware modules depending on the precision mode and layer type, we designed a processing core of FlexBlock to have a hierarchical structure, i.e., multiplier→processing element (PE)→processing unit (PU)→subcore (Fig. 5). A multiplier in FlexBlock accepts both signed and unsigned operands similar to [55]. One global sign bit is used to indicate whether the input/weight tensor is in signed or unsigned representation. Then, nine multipliers

Fig. 3: A motivational example explaining the rate of increase in the number of accumulations at a reduced precision: (a-c) Bit Fusion and (d-f) FlexBlock. All accumulations within an FU in Bit Fusion need to be completed prior to passing the partial sum. However, only a subset of accumulations are completed at each PU in FlexBlock and the remaining accumulations, if required, happen at the end. In addition, FlexBlock provides a better input data reuse requiring less number of new operands.

Fig. 4: Comparison of the MAC utilization during the training of MobileNetV1 with mini-batch size of 16 using various precision-scalable MAC arrays (Bit Fusion [55], BitBlade [52] and the proposed FlexBlock).

Fig. 5: Hierarchical structure of hardware modules in Flex-Block. Four PUs form a subcore in FlexBlock.

are clustered together to make one PE to efficiently map and compute 2D convolutions[2]. In FlexBlock, the sub-word parallelism on the input tensor X is achieved across four PEs in a PU. For the 16-bit input tensor X, each 4-bit sub-word (x3, x2, x1 or x0) is mapped to the corresponding PE. Four PEs are then clustered to form a PU. Note that a PU in Fig 2(b) has a PE with one multiplier for the simple illustration. Another sub-word parallelism on the weight tensor W is realized across four PUs in a subcore. For the 16-bit weight tensor, only a 4-bit sub-word (w3) is mapped to PU3 and transferred to all PEs within the PU assuming that the precision of the input tensor is 16-bit. The remaining three sub-words, i.e., w2, w1 and w0, are distributed to the rest of PUs, respectively.

B. Bit Reconfigurability

In FlexBlock, we provide three levels of bit precision, i.e., 4-bit, 8-bit and 16-bit, for 'sign+mantissa' of each tensor for Conv/GEMM operations. For all three precision levels, we use the same 8-bit shared exponents. Thus, we define three basic FlexBlock formats as `FB12` (X4W4), `FB16` (X8W8) and `FB24` (X16W16). We can also mix-and-match different mantissa bits on the associated tensors, e.g., 8-bit inputs, 4-bit weights, and 16-bit gradients, allowing $3^3 = 27$ combinations.

Sub-word parallelism on X: Fig. 6 shows how the input activations (forward pass) or local gradients (backward pass) are delivered to PEs and PUs for the Conv3 layer[3]. For the

---

[2]Having nine multipliers removes the burden of `im2col` operations on the host CPU. However, the number of multipliers per PE can vary depending on the design strategy (e.g., 8 multipliers per PE).

[3]ConvK represents a K×K convolutional layer.

Fig. 6: Examples on distributing input data to PEs in each PU depending on the input tensor precision (feature maps or local gradients). Inputs are broadcast to PUs in a single cycle.

16-bit mode (X16), each fmap element consists of four 4-bit sub-words. Thus, each sub-word is mapped to a $4b \times 4b$ multiplier in the corresponding PE with the matching gray color. Note that input operands are *broadcast* to PUs rather than mapping different input channels to each PU. For the 8-bit mode (X8), each fmap element consists of two 4-bit sub-words. In this case, we can broadcast two input channels to PUs with the 144-bit interconnect. Similarly, we broadcast four input channels to PUs in the 4-bit mode (X4). In this case, each input channel is mapped to a PE in the PU.

Sub-word parallelism on W: Fig. 7 shows how the weight parameters are delivered to PUs for the Conv3 layer as well. For the 16-bit mode (W16), a 4-bit sub-word of each weight parameter is delivered to the corresponding PU. The other three sub-words are *distributed* to the remaining PUs in a subcore. In this mode, the outputs from all PUs are accumulated by the selective adder tree. For the 8-bit mode (W8), we partition PUs into two clusters and assign the dimension C_{out} across clusters. Thus, PU2-3 and PU0-1 provide partial sums for $C_{out} = k$ and $C_{out} = k + 1$, respectively. The selective 4-way adder tree at the end of the reduction unit produces the two partial sums. For the 4-bit mode (W4), four output channels are distributed to four PUs. In this case, each PU produces a partial sum for the assigned output channel bypassing the selective adder tree.

C. Mapping Various DNN Layers

Mapping 3D operations: In this subsection, we present how the input/weight tensors are being mapped to a FlexBlock core for various DNN layers. The compute modules in FlexBlock are designed with hierarchy so that different tensor

Fig. 7: Examples on distributing weight data to processing units in a subcore depending on the precision of the weight. In this example, 16-bit inputs are assumed (thus, a single input fmap is shared across multiple weight kernels).

TABLE II: Tensor dimensions mapped to each FlexBlock module depending on layer types

FlexBlock Module	Mapped Tensor Dimension
Subcore	C_{in} (Conv1/FC, Conv3), C_{out} (2D Mode), W/H (Conv5, Conv7)
Processing Unit	C_{out} (determined by the precision of weights)
Processing Element	C_{in} (determined by the precision of inputs)
Multiplier	C_{in} (Conv1/FC), W/H (Conv3, Conv5, Conv7, 2D Mode)

dimensions can be easily mapped to these modules depending on the layer type as summarized in Table II. Some examples on how FlexBlock clusters subcores or PUs depending on the layer type are shown in Fig. 8. We assume FB16 (X8W8) as a precision level for the illustration here. For the Conv1 or FC layer, the only partial sums to be accumulated are in dimension C_{in}. Thus, input elements and the corresponding weight parameters across the dimension C_{in} are mapped to subcores, PEs and multipliers (Fig. 8(a)). Subcore0 is responsible for the first 18 input channels ($C_{in} = 0 \sim 17$), Subcore1 is in charge of computing the next 18 input channels ($C_{in} = 18 \sim 35$), and so on. For the Conv3 layer, the only difference over the Conv1 case is in mapping operands to multipliers (Fig. 8(b)). The dimension mapped to the multipliers becomes the fmap width/height (W/H). With larger weight kernels, e.g., a Conv5 or Conv7 layer, we cluster multiple subcores to assign all input elements in the W/H dimension with the size of a weight kernel. To maximize the core utilization on various layer types, we placed six subcores in FlexBlock. For the Conv5 layer, we make two clusters with three subcores each. Then, the core utilization becomes $(5 \times 5)/(3 \times 9) = 0.93$. For the Conv7 layer, we make a cluster with all six subcores providing the core utilization of $(7 \times 7)/(6 \times 9) = 0.91$.

Fig. 8: Examples on how FlexBlock groups subcores depending on the layer type and distributes input/weight tensors. FlexBlock can efficiently process (a) a Conv1 or fully-connected layer, (b) a Conv3 layer and (c) a Conv5 layer. Note that three subcores are grouped together to compute the Conv5 layer. Since we have six subcores in FlexBlock, we can map another set of input feature maps (e.g., two subsequent input channels for 8-bit activations) to the remaining subcores. To compute a Conv7 layer, which is omitted here for brevity, FlexBlock groups all six subcores.

Mapping 2D operations: Thus far, we explained the mapping strategy for 3D operations where the outputs from PUs are vertically accumulated by the reduction unit for 3D mode. FlexBlock has a separate reduction unit for 2D operations to maximize the core utilization. The depthwise convolution (DW Conv) layer is a good example of the 2D operation. In Fig. 9, we illustrate the mapping of a DW Conv3 layer to FlexBlock subcores. For 2D operations, we recommend to keep 8-bit or 16-bit for each tensor since some 2D operations are sensitive to the precision reduction (see Section VII-A). Then, all outputs from PUs per subcore are accumulated by the 4-way adder tree in the 2D reduction unit. For the DW Conv3, each output that comes from the subcore is for each output channel C_{out}. We cluster subcores for larger weight kernels, e.g., DW Conv5 (three subcores) or DW Conv7 (six subcores), which is similar to scaling up the Conv size in the 3D mode (Fig. 8(c)).

Fig. 9: An example of mapping 2D operations to FlexBlock subcores. Here, we select a widely used depthwise Conv3 as an example. Note that outputs from each subcore move horizontally to the reduction unit for 2D mode.

V. OVERALL ARCHITECTURE

The detailed microarchitecture of FlexBlock is provided in Fig. 10. There are three major blocks in the FlexBlock core design: i) a processing core, ii) a reduction unit for 2D operations, and iii) a reduction unit for 3D operations.

A. Major Building Blocks

1) Processing Core: As mentioned earlier, each processing core consists of six subcores to maximize the core utilization on various DNN layers. The 'sign+mantissas' of input and weight tensors are mapped to these subcores and the multiplication results are accumulated together by integer adders, i.e., $(\vec{w} \cdot \vec{x})$ in Eq. (3), if their shared exponents are extracted a priori. Depending on the bit precision (FB12, FB16 or FB24), we block each sub-tensor differently that shares the same exponent. Table III summarizes the minimum block size for each FlexBlock format on various DNN layers. With the reduced precision, the number of input channels mapped to

the processing core doubles (FB16) or quadruples (FB12) compared to FB24 (refer to Fig. 6). In addition, the number of input channels grouped by the block proportionally decreases as the size of weight kernels increases. This fine-grained control of the block size is proposed to make use of all the multipliers available in the processing core, i.e., high core utilization, for various precision levels and DNN layers.

2) Reduction Units: The prior work [7], [52], [54], [55] on the design of precision-scalable MAC arrays have two major limitations: i) no support for DNN training, and ii) low core utilization for 2D operations. The former is handled by supporting the block floating point arithmetic in FlexBlock with a shared exponent handler, arithmetic converters placed prior to FP32 accumulation units, and an FP2BFP converter (Fig. 10). The latter is resolved by placing the dedicated reduction unit for 2D operations along with the default reduction unit for 3D operations (*dual-path reduction units*). The core utilization for the 2D operation becomes important for the DNN training since the computation of a weight gradient ($\Delta \mathbf{W}_{ck}^l$) involves

Fig. 10: The overall microarchitecture of FlexBlock core. There are two reduction units where each unit is dedicated to either 2D or 3D operation mode. Our FlexBlock core is capable of processing various block floating point numbers with 8-bit shared exponents (e.g., `FB12`, `FB16` and `FB24`). The 'out_CORE' goes to a weight update or a batch normalization unit. Before storing the data to DRAM, we extract shared exponents of sub-tensors at a FP2BFP converter for computing the next layer.

TABLE III: The minimum block size of an input tensor sharing the exponent at each FlexBlock format on various layer types

Layer Type	Format	Block Size	Layer Type	Format	Block Size
CONV1/FC	FB12	1×1×216	**CONV5**	FB12	5×5×8
	FB16	1×1×108		FB16	5×5×4
	FB24	1×1×54		FB24	5×5×2
COVN3	FB12	3×3×24	**CONV7**	FB12	7×7×4
	FB16	3×3×12		FB16	7×7×2
	FB24	3×3×6		FB24	7×7×1

Fig. 11: Design of a reconfigurable ReLU-Pool unit placed after batch normalization unit.

a depthwise full convolution between every pair of the local gradient ($\mathbf{G_Y}_k^{l+1}$) and input fmap ($\mathbf{X}_c^l$). Due to the nature of channel-wise computations, a small number of multiplication results are required to be accumulated, which significantly reduces the core utilization in the prior work (Section III).

B. Other Functional Blocks for BFP-based DNN Training

1) Batch Normalization Unit: When training DNNs, batch normalization (BN) is an essential step to find better weight parameters with faster convergence. The BN reduces the internal covariate shift making the training process more stable [24]. To update the BN parameters, i.e., running mean μ and variance σ^2, all input tensors need to be read from DRAM three times [28]. In [28], authors present a fusion technique to reduce the read accesses to twice. In FlexBlock, we use range batch normalization [4] that further reduces the number of DRAM accesses to one with simpler hardware modules.

2) ReLU-Pool Unit: In general, the BN layer is followed by a nonlinear activation function and an optional pooling layer in CNNs. Since the pooling layer may not exist between the BN layer and Conv layer, we design a reconfigurable ReLU-

Pool unit as shown in Fig. 11. For the activation function, FlexBlock provides ReLU and ReLU-α. The ReLU-α unit accepts a clipping value α as a parameter, which is set to 6 for MobileNets [19], [53]. The α can also be trained for improving the accuracy of quantized neural networks [8]. For the pooling layer, FlexBlock allows no pooling, max pooling, or avg pooling by controlling the 'out_sel' signal.

3) Weight Update Unit: A weight update unit is directly connected to the core output buffer. After the weight gradients are computed by the processing core, they are passed to the weight update unit at which a vector unit is placed to multiply a learning rate η to the weight gradients. Then, the weight parameters (W_{ck}^l) are subtracted by the scaled weight gradients ($\eta \cdot \Delta W_{ck}^l$) using element-wise subtract units.

4) Block Floating Point Converter: The FP2BFP converter is placed after the weight update unit to prepare input/gradient and weight tensors for computations at the next layer. This unit blocks each sub-tensor by the pre-defined block size as summarized in Table III depending on the precision level and

Fig. 12: A simple illustration of the process performed by the FP2BFP converter. The number of zero setting errors (ZSEs) can be used to determine the precision mode of each tensor at runtime (Section VII-D).

layer type of the following layer. It has a shared exponent extractor and mantissa aligners that let the processing core of FlexBlock handle only fixed-point computations (Fig. 12). The quantization unit is followed by the FP2BFP unit to minimize communication overheads between the core and DRAM.

VI. METHODOLOGY

A. Software Framework for BFP-based DNN Training

For evaluating the accuracy of a fine-grained blocking of sub-tensors during the DNN training, we implemented a configurable BFP trainer using PyTorch. The BFPsim first defines the network, then it replaces all 'torch.nn.Conv2d' and 'torch.nn.Linear' modules with 'BFP.Conv2d' and 'BFP.Linear' modules in a configuration file provided by the user ('bfp_config.json'). The bit precision of each tensor can be configured in this file as well as the block size that shares the exponent. To monitor the impact of the reduced precision during the computation of weight gradients, we allow users to individually control the mantissa bits for local and weight gradients. The values for blocked sub-tensors are converted to BFP format by extracting the shared exponent and aligning mantissas within the block. Then, we perform pseudo BFP computations in the BFPsim, which means that we store the converted BFP values in FP32 to fully utilize internal functions of PyTorch.

B. Hardware Implementation

To evaluate the proposed FlexBlock hardware in detail, we implemented RTL of all building blocks shown in Fig. 10

except SRAMs. Then, they are synthesized in 65nm CMOS technology using Synopsys Design Compiler (ver. N-2017.09-SP5 [61]) running at 333MHz (T_{clk} = 3ns). To extract more accurate area estimation, post-PnR area is obtained by using Synopsys IC Compiler [60]. For the power analysis, we extracted saif files after setting different BFP modes and layer types then feeding testbenches to FlexBlock. The extracted saif files are then used in Design Compiler to estimate the power consumption of FlexBlock with more realistic switching probabilities at each BFP mode. The energy consumption and cycle time of accessing SRAMs in 65nm are estimated by using CACTI [38], [68]. We assume a FlexBlock accelerator with a 512KB input buffer, a 512KB weight buffer, and a 256KB output buffer that are distributed to 64 FlexBlock cores. Double buffering is utilized to hide the DRAM access latency when possible. The 64 FlexBlock cores are capable of computing $54 \times 64 = 3,456$ 16b×16b MAC operations in FB24. The number of operations increases by $4\times$ (13,824) or $16\times$ (55,296) when the mode is set to FB16 or FB12.

VII. EXPERIMENTAL RESULTS

A. Accuracy of DNN Training with Multi-Mode BFP Support

1) Benchmarks: To evaluate the algorithmic stability of training DNNs in various BFP formats, we selected four datasets, i.e., CIFAR-10, CIFAR-100 [32], ImageNet [9] and WMT14 [5]. Note that FlexBlock is able to individually set mantissa bits for activations, weights, and gradients to achieve the minimum training cost in terms of energy consumption. First, we extensively studied the training of five representative CNNs, i.e., AlexNet [33], VGG16 [57], ResNet-18 [17], MobileNetV1 [19] and DenseNet-121 [20] on simple CIFAR datasets (Fig.13). Then, we trained ResNet-18 on ImageNet and Transformer [63] on WMT14 in various BFP formats to check how well they scale to more complex tasks (Fig. 14).

2) Basic BFP Formats: As a baseline, we trained all benchmarks in FP32. As another baseline, we trained the benchmarks using mixed precision supported by Tensor Cores in NVIDIA RTX3090. In the mixed precision training, multiplications are performed in FP16 and accumulations are done in FP32, i.e., 'FP16+FP32' in Table IV. First, we trained all benchmarks with basic BFP formats, i.e., FB24, FB16 and FB12. In the basic BFP format, 'sign+mantissa' bits of all

Fig. 13: Accuracy of five CNN models trained in FB12 on CIFAR datasets improves by setting 8-bit 'sign+mantissas' for weight gradients (WG), i.e., denoted as FB12+WG16. With the precise weight gradient computation, the accuracy of FlexBlock closely matches with the baseline (FP32) even with FB12 for most computations.

Fig. 14: Comparison of training curves between different precision formats on more complex datasets, i.e., ImageNet and WMT14 En-De (best viewed in color).

TABLE IV: Achieved final accuracy when using a diverse set of precisions for training DNNs on four well-known datasets

Precision	FP32	FP16 +FP32	FB24	FB16	FB12	FB12 +WG16
Dataset	**CIFAR-10 (Top-1 Accuracy)**					
AlexNet	87.12	86.96	87.06	86.68	85.11	86.41
VGG16	92.83	92.90	92.93	92.97	87.43	92.71
ResNet-18	93.37	93.68	93.79	93.75	90.36	93.50
MobileNetV1	87.07	86.85	85.98	86.67	79.80	87.22
DenseNet-121	93.37	93.61	93.06	93.32	89.42	93.01
Dataset	**CIFAR-100 (Top-1 Accuracy)**					
AlexNet	59.52	59.21	59.39	59.77	57.03	59.66
VGG16	73.35	73.16	73.06	73.20	62.51	72.35
ResNet-18	77.26	76.88	77.45	77.23	66.69	76.45
MobileNetV1	67.40	66.80	66.95	67.60	57.11	67.06
DenseNet-121	77.24	77.57	77.68	77.05	70.53	74.93
Dataset	**ImageNet (Top-1 Accuracy)**					
ResNet-18	69.95	69.23	69.92	68.60	58.49	68.20†
Dataset	**WMT14 En-De (Perplexity)**					
Transformer-base	3.87	3.92	4.33	4.29	4.26	4.27

†For ImageNet dataset, training with FB12+WG24 achieves the similar accuracy to the baseline.

tensors are set to the same bit-width, e.g., 8-bit for activation, weight, and gradient tensors in FB16. Throughout the experiments, we stick to the block size provided in Table III to evaluate the training/test accuracy of FlexBlock. If we look at the accuracy comparisons in Table IV, the test accuracy with FB24 or FB16 is similar to the baselines. However, the test accuracy is significantly lower than the baselines when we train the model with FB12 (-6.21% on average for CIFAR datasets and -11.46% for ImageNet, respectively). For Transformer trained on WMT14 dataset, FB12 still provides similar perplexity to other high-precision data formats (Fig. 14).

3) BFP Variants: The accuracy degradation in FB12 is due to the limited precision by having 4-bit 'sign+mantissas'. Note that all FlexBlock formats use 8-bit shared exponents, i.e., same as FP32 and bfloat16, making the dynamic range of FB12 wide enough to train DNNs. As emphasized by the prior work, the precision and/or dynamic range during the weight gradient computation is extremely important for the reliable DNN training [10], [31], [43], [58]. Thus, we may elevate the bit precision to FB16 during the weight update when training with FB12. All computations use 4-bit except when computing the weight gradients (marked as FB12+WG16). As shown in Fig. 13, the test accuracy on CIFAR datasets mostly

Fig. 15: Area and power breakdowns of a FlexBlock core and hardware blocks for the BFP-based training.

matches with the FP32 baseline by using FB12+WG16 in FlexBlock. For DenseNet-121 on CIFAR-100, the accuracy with FB12+WG16 is 2.31% short from the FP32 baseline (but, still 4.4% better than the model trained with FB12). As shown in Fig. 14, training ResNet-18 on ImageNet fails when we use FB12+WG16. By elevating the precision to FB12+WG24, we can achieve similar accuracy to the baseline. This set of experiments shows that supporting multi-mode BFP arithmetic maximizes the efficiency of DNN training.

B. Area and Energy Analysis

To analyze the area and energy consumption, we synthesized the RTL of a single FlexBlock core and all the required functional blocks for the BFP-based training. The reported area and power consumption are shown in Fig. 15. The numbers for the FlexBlock core include the processing core, dual-path reduction units, and control blocks in Fig. 10. About 36% of area and 41% of power consumption are used by the core. In total, 1.48mm^2 of area (~2.81mm^2 after PnR) and 295.59mW of power consumption are used by the single core.

For more realistic analysis, we scaled up the FlexBlock accelerator with 64 cores, which places 54×64 full-precision (i.e., 16b×16b) multipliers for the FB24 mode. RTLs of two baselines are designed and compared to FlexBlock in Table V: i) a systolic array using bfloat16 (in short, SA) and ii) a BFP-based training accelerator using Bit Fusion architecture (in short, BF). For BF, the array size is set to 64×64 for the FB24 mode. The both FlexBlock and BF support multi-precision modes, e.g., FB12, FB16 and FB24. The array size of SA is set to 128×128 to match the number of multipliers to the FB16 mode in BF or FlexBlock (i.e., 8-bit mantissas + 8-bit shared exponents; a BFP version of bfloat16). All three training accelerators are running at 333MHz in 65nm CMOS technology. The area of 64 FlexBlock cores (33.82mm^2) is 2.2× and 1.2× smaller than SA and BF, respectively. The power consumption of 64 FlexBlock cores, i.e., 7.48mW on average, is 1.3× and 2.5× lower than SA and BF, respectively.

To compare the throughput of three training accelerators, RTL simulations were performed to extract the MAC utilization depending on the layer type, precision mode, and tensor dimensions. The extracted MAC utilization is being used in our cycle-approximate simulator to estimate the required clock

TABLE V: Architectural comparisons between TPU-like systolic array, BitFusion-like BFP accelerator, and FlexBlock Cores in terms of area, power consumption, and energy efficiency

Training Hardware	TPU-like Systolic Array (SA)	BitFusion-based BFP Accelerator (BF)	FlexBlock Cores (Proposed; FB)
Technology	65nm	65nm	65nm
Supported Precision	`bfloat16`	FB12, FB16, FB24	FB12, FB16, FB24
Array Size	128×128	64×64 (for FB24)	54×64 (for FB24)
# of Multipliers	**128×128**	256×256 (FB12) / **128×128 (FB16)** / 64×64 (FB24)	216×256 (FB12) / **108×128 (FB16)** / 54×64 (FB24)
Area [mm^2]	74.38	40.22	33.82
Power Consumption [W]	9.84	17.83 (FB12) / 17.13 (FB16) / 15.96 (FB24)	8.27 (FB12) / 7.80 (FB16) / 7.36 (FB24)
Clock Frequency	333MHz	333MHz	333MHz
Throughput [TFLOPS]	**1.77**	2.82 (FB12) / **1.77 (FB16)** / 0.66 (FB24)	8.78 (FB12) / **3.35 (FB16)** / 0.95 (FB24)
Efficiency [GFLOPS/W]	179.6	157.96 (FB12) / **103.16 (FB16)** / 41.11 (FB24)	1,061.7 (FB12) / **428.9 (FB16)** / 128.7 (FB24)

Fig. 16: Comparisons of the performance and energy consumption between the systolic array (SA), the BitFusion-based BFP accelerator (BF), and the proposed FlexBlock (FB in red). For the analysis, we evaluated five CNN benchmarks on ImageNet and Transformer-base model on WMT14. Here, `FB12` represents `FB12+WG24` format.

cycles considering the memory access latency and the on-chip buffer size. Instead of using small CIFAR datasets for CNN benchmarks, we used ImageNet for all CNN models (with mini-batch size of 128) to compare three architectures in terms of the performance and energy consumption. For Transformer model, we also used the mini-batch size of 128. With the estimated clock cycles and the extracted power consumption of each accelerator, we report and compare the performance and energy consumption in Fig. 16. The training accelerators at an equivalent precision level are compared, e.g., SA with `bfloat16` is compared to BF and FlexBlock in `FB16`. As a result, FlexBlock reduces the energy consumption (training time) by 65.3%, 68.2%, and 79.3% (32.0%, 47.5%, and 68.4%) on average compared to BF at `FB24`, `FB16`, and `FB12+WG24`, respectively. Compared to SA, FlexBlock reduces the energy consumption and training time by 44.3% and 52.7% on average. When we train DNN models with `FB12+WG24` in FlexBlock, we can reduce the energy consumption and training time by 76.4% and 81.0% on average compared to SA.

C. Performance Comparison with GPU

In this subsection, we compare the training speed and energy efficiency with a high-end GPU card, i.e., NVIDIA RTX3090. When training in GPU, we utilized the mixed precision training (`FP16+FP32`) presented in [43]. The runtime for a single training iteration on 128 batches in GPU is measured

TABLE VI: Comparisons of the performance and energy efficiency between GPU (NVIDIA RTX3090) and FlexBlock when training CNN benchmarks on ImageNet

	DNN Benchmark	AlexNet	VGG16	ResNet	MobileNet	DenseNet
GPU (`FP16` `+FP32`)	**Runtime [ms]**	46.0	296.4	71.4	65.9	214.0
	Power [W]	207.7	326.7	321.4	322.7	336.2
	GFLOPS/W	41.1	61.0	36.3	9.8	15.5
FlexBlock (`FB16`)	**Runtime [ms]**	178.8	1372.3	243.6	68.0	296.1
	Power [W]	19.0	19.0	19.0	19.0	19.0
	GFLOPS/W	115.5	226.4	179.9	160.9	197.9
FlexBlock (`FB12` `+WG24`)	**Runtime [ms]**	61.8	391.4	114.1	36.7	116.9
	Power [W]	19.5	19.5	19.5	19.5	19.5
	GFLOPS/W	325.9	774.4	374.8	290.8	489.0

* All functional units listed in Fig. 15 are included in the power report.

by a built-in function in Python. The power consumption of running each CNN benchmark is measured by `nvidia-smi`. The reported numbers are summarized in Table VI. As one RTX3090 card has 384 Tensor Cores, it is equivalent to 20,992 `FP16` multipliers. Thus, we compare the performance with FlexBlock using `FB16` and `FB12+WG24`. The performance of FlexBlock with `FB16` is 2.9× lower than GPU. However, this may come from the ~1/4 of GPU clock frequency used by the current FlexBlock hardware. The training speed with `FB12+WG24` on AlexNet, VGG16 and ResNet-18 is 1.4× slower than GPU. However, training in FlexBlock is 1.8× faster on MobileNetV1 and DenseNet-121 than GPU. Considering the 15.5× lower power consumption, FlexBlock in `FB12+WG24` achieves similar training speed with much higher energy efficiency (18.4×) compared to the recent GPU.

D. Case Study: Dynamic Precision Control

So far, we studied the benefit of statically assigning a different bit precision to each tensor for efficient DNN training. However, it will be extremely useful if we can automatically tune the precision of each tensor at runtime while training a DNN model. To accomplish this, we count the number of zero setting errors (ZSEs) due to the shift operations in the FP2BFP converter explained in Fig. 12. We keep track of ZSEs of each tensor for the current epoch and determine the precision for the next training epoch by comparing the ratio of ZSEs to pre-defined thresholds. We utilize a hysteresis controller to slowly change the bit precision (Fig. 17(a)). If the ratio of ZSEs is too large, it means that the current mantissa bit is not sufficient to train the model. To demonstrate the feasibility of this approach, we fixed activation and weight precisions to `FB12` and dynamically adjusted the precision of weight gradients between `FB12` and `FB16` at runtime. We tested this approach on ResNet-18 with CIFAR-10 dataset. Fig. 12(b) shows how layer-wise precision adaptation is done by the proposed control mechanism. Thanks to this dynamic precision control, we observed 16% speed-up compared to the static `FB12+WG16` case with no accuracy degradation ($\sim$45% of weight gradients, `WG`, were set to `FB12` instead of `FB16`).

VIII. RELATED WORK

A. Accelerators for Training Deep Neural Networks

As training DNNs requires higher memory bandwidth and more computational resources than the inference, many prior work proposed accelerators [25], [30], [64] or systems [11], [21], [26], [66] optimized for the training. In ScaleDeep [64], heterogeneous processing tiles are utilized to map different types of DNN layers for the efficient training. In Deep-Train [30], authors present temporally heterogeneous tensor mapping with near-memory computing using a 3D-stacked memory. Gist [25] encodes the feature maps computed during the forward pass to efficiently store them for later use in the backward pass. In addition, many research focus on the distributed (or pipelined) training of DNN models [11], [21], [26], [66] to achieve the best training performance.

Recently, sparse DNN accelerators are proposed to increase the throughput of training DNNs by exploiting the possible sparsity at each tensor [48], [69]. SIGMA [48] proposes a training accelerator that handles both sparsity and irregular structure in GEMM operations by using a Benes network for efficient workload distribution. Authors in [69] present a sparse DNN training accelerator, named Procrustes, that exploits one source of sparsity (either activations or weights) during the forward pass, backward pass, or weight update. Procrustes leverages the mini-batch dimension, i.e., a dense tensor dimension, for the balanced workload distribution when performing arithmetic operations involving sparse tensors.

B. Reduced Precision During DNN Training

To maximize the arithmetic density of training accelerators, fixed-point logic can be used during the DNN training [10], [15], [31]. In [15], stochastic rounding is used in training

Fig. 17: Dynamic precision control: (a) a hysteresis controller is used to determine the precision of a tensor for the next training epoch, (b) Layer-wise precision adaptation by checking the ZSE ratio (only three layers are shown for brevity).

DNNs with `INT16` to minimize the expected numerical error. This work, however, evaluated the proposed method on relatively simple tasks, i.e., classifying 10 different image classes using MNIST and CIFAR-10. Other previous work aggressively reduce the precision during the training at the cost of noticeable accuracy degradation [2], [22], [23], [37], [39], [42], [49], [59]. To overcome the limited range of the fixed-point representation, Flexpoint [31] extracts a 5-bit shared exponent for each tensor (coarse-grained) and adjusts the exponent twice per mini-batch to prevent the overflows. To perform in-place exponent extraction, rather than periodically checking the overflow, an accelerator with hybrid block floating point [10] is proposed that performs multiply-and-accumulate operations on the fixed-point logic while other remaining operations are done in `FP32`. Compared to [10], [31], FlexBlock allows more fine-grained blocking of sub-tensors to support variable precisions for accelerating the training process as discussed in Section VII-B. In the very recent work on low-precision training [1], [47], [58], [65], [67], 8-bit floating point (`FP8` or `HFP8`) has been used to train DNNs with a little accuracy loss on a wide spectrum of benchmarks. However, the hardware associated with FP8 training uses specific mantissa and exponent bits for its maximum energy efficiency, which lacks flexibility.

IX. CONCLUSION

In this work, we proposed a DNN training accelerator, i.e., FlexBlock, designed to support multi-precision block floating point arithmetics. This multi-mode BFP support has two main advantages: i) enabling users to train DNNs at desired precision levels, and ii) reducing the training time for faster DNN exploration. We identified the inherent limitation of the prior precision-scalable MAC arrays and hierarchically allocated the tensor dimensions to compute units in FlexBlock for better performance. As the computations involved in the DNN training are rapidly increasing, this work will encourage developing training hardware with better flexibility and higher energy efficiency using various BFP formats.

REFERENCES

[1] A. Agrawal, S. K. Lee, J. Silberman, M. Ziegler, M. Kang, S. Venkataramani, N. Cao, B. Fleischer, M. Guillorn, M. Cohen, S. Mueller, J. Oh, M. Lutz, J. Jung, S. Koswatta, C. Zhou, V. Zalani, J. Bonanno, R. Casatuta, C.-Y. Chen, J. Choi, H. Haynie, A. Herbert, R. Jain, M. Kar, K.-H. Kim, Y. Li, Z. Ren, S. Rider, M. Schaal, K. Schelm, M. Scheuermann, X. Sun, H. Tran, N. Wang, W. Wang, X. Zhang, V. Shah, B. Curran, V. Srinivasan, P.-F. Lu, S. Shukla, L. Chang, and K. Gopalakrishnan, "9.1 a 7nm 4-core AI chip with 25.6TFLOPS hybrid FP8 training, 102.4TOPS INT4 inference and workload-aware throttling," in *IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 144–146.

[2] H. Alemdar, N. Caldwell, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient AI applications," *arXiv:1609.00222*, 2016. [Online]. Available: http://arxiv.org/abs/1609.00222

[3] M. Andrychowicz, B. Baker, M. Chociej, R. Józefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning dexterous in-hand manipulation," *The International Journal of Robotics Research (IJRR)*, vol. 39, no. 1, pp. 3–20, 2020.

[4] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," in *Proceedings of International Conference on Neural Information Processing Systems (NeurIPS)*, 2018, pp. 5151–5159.

[5] O. Bojar, C. Buck, C. Federmann, B. Haddow, P. Koehn, J. Leveling, C. Monz, P. Pecina, M. Post, H. Saint-Amand, R. Soricut, L. Specia, and A. Tamchyna, "Findings of the 2014 workshop on statistical machine translation," in *Proceedings of the Ninth Workshop on Statistical Machine Translation (WMT)*, June 2014, pp. 12–58.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[7] V. Camus, L. Mei, C. Enz, and M. Verhelst, "Review and benchmarking of precision-scalable multiply-accumulate unit architectures for embedded neural-network processing," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS)*, vol. 9, no. 4, pp. 697–711, 2019.

[8] J. Choi, Z. Wang, S. Venkataramani, P. I. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: parameterized clipping activation for quantized neural networks," *arXiv:1805.06085*, 2018. [Online]. Available: http://arxiv.org/abs/1805.06085

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

[10] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "Training DNNs with hybrid block floating point," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[11] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xu, R. Patel, and M. Herbordt, "FPDeep: Acceleration and load balancing of CNN training on FPGA clusters," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 81–84.

[12] Google, "Cloud TPU," https://cloud.google.com/tpu, 2017, [Online; accessed 07-June-2021].

[13] Google Cloud, "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, 2019, [Online; accessed 07-June-2021].

[14] S. M. Grigorescu, B. Trasnea, T. T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *arXiv:1910.07738*, 2019. [Online]. Available: http://arxiv.org/abs/1910.07738

[15] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of International Conference on Machine Learning (ICML)*, vol. 37, July 2015, pp. 1737–1746.

[16] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *International Conference on Learning Representations (ICLR)*, 2016. [Online]. Available: http://arxiv.org/abs/1510.00149

[17] 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 (CVPR)*, 2016, pp. 770–778.

[18] B. Hickmann, J. Chen, M. Rotzin, A. Yang, M. Urbanski, and S. Avancha, "Intel Nervana neural network processor-t (NNP-T) fused floating point many-term dot product," in *IEEE Symposium on Computer Arithmetic (ARITH)*, 2020, pp. 133–136.

[19] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[20] G. Huang, Z. Liu, G. Pleiss, L. Van Der Maaten, and K. Weinberger, "Convolutional networks with dense connectivity," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, pp. 1–1, 2019.

[21] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, "GPipe: Efficient training of giant neural networks using pipeline parallelism," *arXiv:1811.06965*, 2018. [Online]. Available: http://arxiv.org/abs/1811.06965

[22] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proceedings of International Conference on Neural Information Processing Systems (NIPS)*, 2016, pp. 4114–4122.

[23] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *Journal of Machine Learning Research (JMLR)*, vol. 18, no. 187, pp. 1–30, 2018.

[24] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of International Conference on International Conference on Machine Learning (ICML)*, 2015, pp. 448–456.

[25] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2018, pp. 776–789.

[26] Y. Jang, S. Kim, D. Kim, S. Lee, and J. Kung, "Deep partitioned training from near-storage computing to DNN accelerators," *IEEE Computer Architecture Letters (CAL)*, pp. 1–4, 2021.

[27] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 1–12, 2017.

[28] W. Jung, D. Jung, B. Kim, S. Lee, W. Rhee, and J. H. Ahn, "Restructuring batch normalization to accelerate CNN training," in *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2019, pp. 1–13.

[29] K. Kalliojarvi and J. Astola, "Roundoff errors in block-floating-point systems," *IEEE Transactions on Signal Processing (TSP)*, vol. 44, no. 4, pp. 783–790, 1996.

[30] D. Kim, T. Na, S. Yalamanchili, and S. Mukhopadhyay, "DeepTrain: A programmable embedded platform for training deep neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 37, no. 11, pp. 2360–2370, 2018.

[31] U. Köster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloss, R. J. Pai, and N. Rao, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Proceedings of International Conference on Neural Information Processing Systems (NIPS)*, 2017, pp. 1740–1750.

[32] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10 and CIFAR-100 dataset," https://www.cs.toronto.edu/~kriz/cifar.html, 2010, [Online; accessed 20-July-2021].

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification

with deep convolutional neural networks," *Advances in Neural Information Processing Systems (NIPS)*, vol. 25, pp. 1097–1105, 2012.

[34] J. Kung, D. Kim, and S. Mukhopadhyay, "Dynamic approximation with feedback control for energy-efficient recurrent neural network hardware," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2016, pp. 168–173.

[35] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: A 50.6TOPS/W unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2018, pp. 218–220.

[36] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo, "LNPU: A 25.3TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2019, pp. 142–144.

[37] F. Li and B. Liu, "Ternary weight networks," *arXiv:1605.04711*, 2016. [Online]. Available: http://arxiv.org/abs/1605.04711

[38] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 694–701.

[39] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proceedings of International Conference on Machine Learning (ICML)*, 2016, pp. 2849–2858.

[40] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny deep learning on IoT devices," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33. Curran Associates, Inc., 2020, pp. 11 711–11 722.

[41] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 751–766.

[42] N. Mellempudi, A. Kundu, D. Das, D. Mudigere, and B. Kaul, "Mixed low-precision deep learning inference using dynamic fixed point," *arXiv:1701.08978*, 2017. [Online]. Available: http://arxiv.org/abs/1701.08978

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

[44] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, Q. V. Le, J. Laudon, R. Ho, R. Carpenter, and J. Dean, "A graph placement methodology for fast chip design," *Nature*, vol. 594, pp. 207–212, June 2021. [Online]. Available: https://doi.org/10.1038/s41586-021-03544-w

[45] B. Moons, B. D. Brabandere, L. V. Gool, and M. Verhelst, "Energy-efficient convnets through approximate computing," in *IEEE Winter Conference on Applications of Computer Vision (WACV)*, Mar. 2016.

[46] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 246–247.

[47] J. Park, S. Lee, and D. Jeon, "A 40nm 4.81TFLOPS/W 8b floating-point training processor for non-sparse neural networks using shared exponent bias and 24-way fused multiply-add tree," in *IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 1–3.

[48] 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 *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2020, pp. 58–70.

[49] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *European Conference on Computer Vision (ECCV)*. Springer International Publishing, 2016, pp. 525–542.

[50] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *arXiv:1802.01548*, 2018. [Online]. Available: http://arxiv.org/abs/1802.01548

[51] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *arXiv:1506.02640*, 2016.

[52] S. Ryu, H. Kim, W. Yi, and J.-J. Kim, "BitBlade: Area and energy-efficient precision-scalable neural network accelerator with bitwise sum-mation," in *ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[53] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4510–4520.

[54] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[55] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit Fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018, pp. 764–775.

[56] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks," in *IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 240–241.

[57] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[58] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, 2019.

[59] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan, "Ultra-low precision 4-bit training of deep neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, pp. 1796–1807.

[60] Synopsys, "IC Compiler II Implementation User Guide: Version L-2016.03," 2016.

[61] Synopsys, "Design Compiler User Guide: Version N-2017.09," 2017.

[62] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the International Conference on Machine Learning (ICML)*, vol. 97, June 2019, pp. 6105–6114.

[63] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[64] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017, pp. 13–26.

[65] S. Venkataramani, V. Srinivasan, W. Wang, S. Sen, J. Zhang, A. Agrawal, M. Kar, S. Jain, A. Mannari, H. Tran, Y. Li, E. Ogawa, K. Ishizaki, H. Inoue, M. Schaal, M. Serrano, J. Choi, X. Sun, N. Wang, C.-Y. Chen, A. Allain, J. Bonano, N. Cao, R. Casatuta, M. Cohen, B. Fleischer, M. Guillorn, H. Haynie, J. Jung, M. Kang, K.-h. Kim, S. Koswatta, S. Lee, M. Lutz, S. Mueller, J. Oh, A. Ranjan, Z. Ren, S. Rider, K. Schelm, M. Scheuermann, J. Silberman, J. Yang, V. Zalani, X. Zhang, C. Zhou, M. Ziegler, V. Shah, M. Ohara, P.-F. Lu, B. Curran, S. Shukla, L. Chang, and K. Gopalakrishnan, "RaPiD: AI accelerator for ultra-low precision training and inference," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2021, pp. 153–166.

[66] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, no. 2, Mar. 2020. [Online]. Available: https://doi.org/10.1145/3377454

[67] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018, p. 7686–7695.

[68] S. Wilton and N. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 31, no. 5, pp. 677–688, 1996.

[69] 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 *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2020.

[70] J.-H. Yoon, M. Chang, W.-S. Khwa, Y.-D. Chih, M.-F. Chang, and A. Raychowdhury, "A 40nm 64Kb 56.67TOPS/W read-disturb-tolerant compute-in-memory/digital RRAM macro with active-feedback-based

read and in-situ write verification," in *IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 404–406.