# ETA: An Efficient Training Accelerator for DNNs Based on Hardware-Algorithm Co-Optimization

Jinming Lu, Chao Ni, and Zhongfeng Wang, *Fellow, IEEE*

***Abstract*— Recently, the efficient training of deep neural networks (DNNs) on resource-constrained platforms has attracted increasing attention for protecting user privacy. However, it is still a severe challenge since the DNN training involves intensive computations and a large amount of data access. To deal with these issues, in this work, we implement an efficient training accelerator (ETA) on field-programmable gate array (FPGA) by adopting a hardware-algorithm co-optimization approach. A novel training scheme is proposed to effectively train DNNs using 8-bit precision with arbitrary batch sizes, in which a compact but powerful data format and a hardware-oriented normalization layer are introduced. Thus the computational complexity and memory accesses are significantly reduced. In the ETA, a reconfigurable processing element (PE) is designed to support various computational patterns during training while avoiding redundant calculations from nonunit-stride convolutional layers. With a flexible network-on-chip (NoC) and a hierarchical PE array, computational parallelism and data reuse can be fully exploited, and memory accesses are further reduced. In addition, a unified computing core is developed to execute auxiliary layers such as normalization and weight update (WU), which works in a time-multiplexed manner and consumes only a small amount of hardware resources. The experiments show that our training scheme achieves the state-of-the-art accuracy across multiple models, including CIFAR-VGG16, CIFAR-ResNet20, CIFAR-InceptionV3, ResNet18, and ResNet50. Evaluated on three networks (CIFAR-VGG16, CIFAR-ResNet20, and ResNet18), our ETA on Xilinx VC709 FPGA achieves 610.98, 658.64, and 811.24 GOPS in terms of throughput, respectively. Compared with the prior art, our design demonstrates a speedup of 3.65× and an energy efficiency improvement of 8.54× on CIFAR-ResNet20.**

***Index Terms*— Deep neural networks (DNNs), field-programmable gate array (FPGA), hardware accelerator, normalization, training.**

## I. Introduction

OVER the past decade, there has been a substantial advancement of deep neural networks (DNNs) across a variety of domains, such as computer vision [1], automatic speech recognition [2], natural language processing [3], and autonomous vehicles [4]. The success of DNNs benefits from enormous accessible data, growing computing capacity, and evolving algorithms. Besides, the improvement of model performance is usually accompanied by increasing model size or data volume. Traditionally, due to the intensive computation and data access, DNNs can only be deployed on high-end graph processing units (GPUs) with powerful computing capacity and high energy consumption, which incurs considerable economic costs and is unrealistic to be applied in edge computing scenarios. As a result, deploying DNNs on a power-efficient and resource-constrained platform is a critical obstacle, which hinders more widespread application of DNNs.

Previous researchers have tried to improve the processing efficiency of DNNs from various aspects [5]. From the perspective of algorithmic optimization, some studies aimed to reduce DNNs complexity and produce compact DNN models. Typical technologies include model pruning [6], [7], quantization [8]–[10], low-rank decomposition [11], [12], and knowledge distillation [13], [14]. From the perspective of hardware, many domain-specific accelerators were developed to process DNNs on different platforms efficiently. Usually, they explore the dataflow optimization, parallel computing, fast algorithm transformation, or sparsity exploitation to improve the throughput while reducing the energy and resource consumption [15]–[19].

Nowadays, people usually collect massive data, train DNNs on GPU clusters, deploy well-trained models on embedded devices, and then execute model inference for specific applications. Nevertheless, there is an increasing demand for training models on personal devices. On the one hand, users take personal privacy more seriously and prefer to keep their data locally [20], [21]. On the other hand, data distribution in real scenarios may differ from the distribution of original training data, and thus, models are expected to adjust themselves on new tasks [22], [23].

Most works mentioned above mainly focus on the hardware optimization for the inference of DNNs, which can not be applied for model training directly. Compared with inference, training has very different computational patterns and memory requirements. For supervised training, the labeled data are fed into a DNN until convergence in an iterative procedure, in which each iteration involves forward propagation (FP), backward propagation (BP), weight-gradient generation (WG), and weight update (WU). Therefore, roughly three times more operations are executed for one sample. The intermediate

Manuscript received April 12, 2021; revised August 18, 2021 and November 22, 2021; accepted January 20, 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 61774082, in part by the Fundamental Research Funds for the Central Universities under Grant 021014380065, and in part by the Key Research Plan of Jiangsu Province of China under Grant BE2019003-4. *(Jinming Lu and Chao Ni contributed equally to this work.) (Corresponding author: Zhongfeng Wang.)*

The authors are with the School of Electronic Science and Engineering, Nanjing University, Nanjing 210008, China (e-mail: jmlu@smail.nju.edu.cn; nichao@smail.nju.edu.cn; zfwang@nju.edu.cn).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNNLS.2022.3145850.

Digital Object Identifier 10.1109/TNNLS.2022.3145850

activations have to be kept in memory for a longer time due to the following gradient computation during WG, which causes a dramatic increase in storage requirements. As a result, there are more severe challenges in designing DNN training accelerators.

More recently, some efforts tried to optimize the training process from the algorithm or hardware. However, most existing studies take limited discussions where some issues are not thoroughly investigated yet.

1) Low-bit precision numbers, such as 16-bit floating-point (FP16) [24], [25], 8-bit integer (INT8) [26]–[28], and Posit [29], are adopted to represent data throughout the training procedure. However, existing methods either incur nonnegligible accuracy loss for large-scale models because of the aggressive quantization or leverage complex auxiliary tricks and floating-point arithmetic. Therefore, a data format with both high representation ability and efficient computing is urgently required.
2) Batch normalization (BN) layers play an indispensable role in modern DNN models [30], and they can not be folded into the adjacent convolutional (CONV) layers during training. A BN layer can not be executed until the preceding CONV layer completes the whole process of a mini-batch. Meanwhile, the batch size should be large enough to ensure the convergence of training. As a result, the straightforward implementation of BN layers on hardware leads to a severe increase in memory consumption and execution time. Besides, the original L2 normalization approach involves some complicated operations, such as square, sqrt, and division, which are sensitive to quantization and challenging to implement on hardware. Most of the previous works have chosen to neglect them or keep them in the floating-point [27], [31]. Rounding number computation with HUB format [32] will be a promising choice to handle complex numerical issues in BN layers.
3) Although all CONV layers can be formulated in convolutional operations during different phases, their characteristics vary dramatically. The current prevailing architectures [1], [33], [34] prefer a CONV layer with a stride of 2 rather than a pooling layer to downsample feature maps, which brings more computational diversity in the training process.

In this work, we mainly concentrate on addressing the above issues by using hardware and algorithm co-optimization. Finally, we design a reconfigurable hardware accelerator for efficiently training DNNs on field-programmable gate array (FPGA) devices. Our contributions are summarized as follows.

1) We propose an efficient and robust DNN training algorithm using 8-bit precision. A compact but powerful format, named piecewise integer (PINT), is proposed to represent the data effectively, and the dedicated arithmetic unit is designed. We also introduce a novel normalization layer, L1-norm filter response normalization (L1-FRN) layer, to eliminate the batch dependence and remove the complicated L2-norm operations. Experimental results show that our training framework could achieve the same accuracy with baseline across various datasets and model architectures.
2) A reconfigurable processing element (PE) is designed to efficiently support various computational patterns in training. Therefore, dummy operations can be skipped in nonunit-stride CONV layers. To fully exploit the data reuse, we develop a hierarchical PE array, which forms a convolutional core (C-Core) together with an optimized memory management strategy and efficient network-on-chip (NoC). Hence, the PE utilization is kept at a high level at all times.
3) We present an auxiliary core (A-Core) to process auxiliary layers, including L1-FRN, threshold linear unit (TLU), and WU. We extract the basic operations and optimize the computation flow of these layers. Hence, they can be performed in a unified architecture. Besides, because of the algorithm optimization, CONV layers and their auxiliary layers are executed in a tilewise pipelined manner. Thus, the overall latency can be remarkably reduced.
4) An efficient training accelerator (ETA) is designed for DNN training. Implemented on the Xilinx VC709 platform, our design achieves up to 811.25 GOPS and 93.86 GOPS/W in terms of throughput and energy efficiency, respectively. The comparison results illustrate that our accelerator outperforms prior works significantly.

TABLE I
NOTATIONS INVOLVED IN DNN TRAINING

| Notation | Definition |
|---|---|
| $l$ | layer index. |
| $s$ | stride in the convolution layer. |
| $x/y$ | row/column index of feature map. |
| $k_x/k_y$ | row/column index of weight kernel. |
| $c/m$ | input/output channel index. |
| $X/Y$ | feature map height/width. |
| $K_x/K_y$ | weight kernel height/width. |
| $C/M$ | number of input/output channel. |

The rest of this article is organized as follows. Section II gives an overview of the DNN training and introduces the related works. Section III describes our efficient training algorithm. The design of our reconfigurable training accelerator is presented in Section IV, and the implementation results are reported in Section V. Finally, Section VI concludes this article.

## II. BACKGROUND

### A. DNN Training

In this section, we briefly introduce the overall computation flow in DNN training. The training dataflow is shown in Fig. 1 for a DNN. All involved data are categorized into weights (W), activations (A), errors (E), and gradients (G), whose superscripts indicate layer indices. The definitions of related notations are listed in Table I. We divide the training process into four phases, including FP, BP, WG, and WU. Their calculation processes are elaborated as follows.

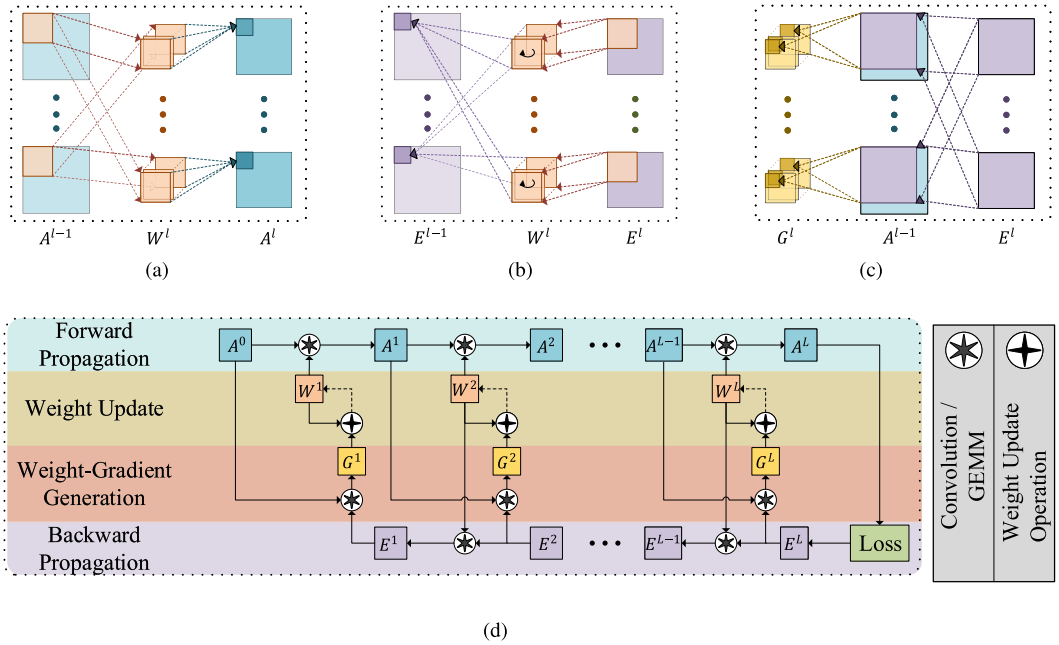



Fig. 1. Diagram of overall computation flow in DNN training and computations of a CONV layer during different phases. (a) FP. (b) BP. (c) WG. (d) Overall computation flow in DNN training.

1) *FP:* The input images are fed into the model, and then, activations ($A^l$) of each layer are successively calculated based on the layer type. For a CONV layer shown in Fig. 1(a), according to (1), the output value $A^l[m, x, y]$ is calculated by convolving weights ($W^l$) with the activations of the previous layer ($A^{l-1}$), where an accumulation is conducted across the input channels. For a fully connected (FC) layer, the computation can be formulated as a simple matrix–vector multiplication in (2). At the end of FP, a predicted label is compared with the target label, and the loss is computed

$$A^l[m,x,y] = \sum_{c=0}^{C-1}\sum_{k_x=0}^{K_x-1}\sum_{k_y=0}^{K_y-1} W^l\big[m,c,k_x,k_y\big] \times A^{l-1}\big[c,x+k_x,y+k_y\big] \quad (1)$$

$$A^l = W \times A^{l-1}. \quad (2)$$

2) *BP:* The computation procedure of BP is similar to that of FP. Nevertheless, kernels need a rotation of 180° as well as transposition between input channel and output channel dimensions for CONV layers, as shown in Fig. 1(b), while the weight matrix needs to be transposed for FC layers. The computation in a CONV layer during the BP phase is shown in (3). The errors of an FC layer are obtained by (4)

$$E^{l-1}[c,x,y] = \sum_{m=0}^{M-1}\sum_{k_x=0}^{K_x-1}\sum_{k_y=0}^{K_y-1} W^l\big[m,c,k_x,k_y\big] \times E^l\big[c,x+K_x-1-k_x, y+K_y-1-k_y\big] \quad (3)$$

$$E^{l-1} = E^l \times \left(W^l\right)^T. \quad (4)$$

3) *WG:* Weight gradients are the derivatives of the loss with respect to weights. For a CONV layer, weight gradients are obtained by convolving activations of the previous layer with errors of the present layer as shown in (5), which is a large window convolution operation presented in Fig. 1(c). For an FC layer, they are completed by an outer production operation as shown in (6). An accumulation operation across the batch dimension will be conducted if we take the batch size into account

$$G^l\big[m,c,k_x,k_y\big] = \sum_{m=0}^{M-1}\sum_{x=0}^{X-1}\sum_{y=0}^{Y-1} E^l[m,x,y] \times A^{l-1}\big[c,x+k_x,y+k_y\big] \quad (5)$$

$$G^l = A^{l-1} \circ E^l. \quad (6)$$

4) *WU:* It can be performed once the weight gradients are obtained. Here, we adopt the stochastic gradient descent (SGD) with momentum as the optimizer, which is one of the most popular optimizers. Then, new velocities and new weights are obtained as follows:

$$V_{t+1}^l \leftarrow \mu V_t^l + G_{t+1}^l$$
$$W_{t+1}^l \leftarrow W_t^l - \eta V_{t+1}^l \quad (7)$$

where $V$, $\mu$, and $\eta$ denote velocities, momentum, and learning rate, respectively.

### B. Related Works

With the significant advance of DNNs, many researchers' efforts have been devoted to training DNNs efficiently on resource-constrained devices.

The quantization technique plays an increasingly important role in this area. Gupta *et al.* [35] explored training DNNs

with 16-bit fixed-point, in which the stochastic rounding method was utilized. In [36], the binary logarithmic was exploited to represent data in both training and inference procedures. Then, the multiplication can be replaced by a shift operation. Dorefa-Net [37] proposed a framework for training DNNs with different bit widths for weights, activations, and gradients. However, these methods caused severe accuracy degradation for large-scale networks. Micikevicius *et al.* [24] declared that training DNNs with FP16 could reach baseline accuracy on multiple applications. Besides, the brain floating-point (BFLOAT16) format [25] was also studied for efficient training. Nevertheless, both FP16 and BFLOAT16 need complicated floating-point arithmetic operations, which brings unacceptable costs for edge devices. More recently, Zhao *et al.* [27] and Zhu *et al.* [31] successively presented 8-bit integer training frameworks, which have achieved good performance on various tasks. However, they only quantized CONV layers and kept BN layers in the floating point.

In addition, several DNN training accelerators are developed based on application-specific integrated circuit (ASIC) or FPGA platforms. Zhao *et al.* [38] proposed a training framework by reconfiguring a stream datapath at runtime, where various customizable kernels are designed for basic operations. Liu *et al.* [39] presented a uniform computation engine for various operations. A multi-FPGA cluster was utilized to deploy a whole DNN training program in FP-Deep [40]. By exploiting the intralayer and interlayer parallelism, the lifetime of immediate activations can be minimized. Dey *et al.* [41] implemented a reconfigurable architecture for training a sparse network whose connectivity is predefined. In [42], a compressed CNN training framework was proposed using a filterwise pruning and fixed-point quantization method. Both operator- and result-sparse patterns were utilized in their dedicated PEs. An automatic compiler for CNN training accelerator was developed in [43], and then, high-bandwidth memory (HBM) was used in [44] to support modern CNNs more efficiently.

## III. Hardware-Friendly Training Algorithm

In this section, we introduce an efficient and hardware-oriented algorithm for DNN training. With a novel and compact data format, named PINT, DNNs can be trained successfully using 8-bit representation without noticeable accuracy loss. Therefore, memory accesses are reduced by four times, and the basic computation unit is significantly simplified. By developing a hardware-friendly and quantization-insensitive normalization layer, the overall training process is further simplified.

### A. Piecewise Integer

Integer arithmetic is one of the most popular choices when deploying DNNs in reduced precision due to its efficient computation and flexible representation. Typically, a dynamic quantization scheme is utilized to convert the continuous data to the corresponding integer value. Given the tensor data $\boldsymbol{x}$ and bit width $k$, the quantization procedure can be formulated as

$$\begin{aligned} s &= \frac{\max(|\boldsymbol{x}|)}{2^{k-1}} \\ \boldsymbol{q} &= \text{Clamp}\left(\text{Round}\left(\frac{\boldsymbol{x}}{s}\right), -2^{k-1}, 2^{k-1}-1\right) \\ \hat{\boldsymbol{x}} &= \boldsymbol{q} \times s \end{aligned} \tag{8}$$

where $s$ denotes the scaling factor to normalize the data, $\boldsymbol{q}$ is the integer format after quantization that will be used in the subsequent calculations, and $\hat{\boldsymbol{x}}$ represents the dequantized value.

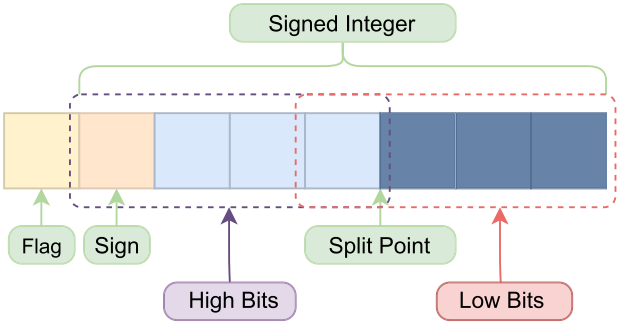



Fig. 2. Basic structure of an (8, 3) PINT number.

Unlike the integer, a PINT number is defined by two parameters, including the bit width $k$ and the split point $d$. $d$ indicates how to split a $k$-bit PINT into two overlapped sets, i.e., high bits (HBs) and low bits (LBs). Besides, a $(k, d)$ PINT number comprises one *flag* bit and one $(k-1)$-bit signed integer. flag and $d$ jointly split the encoding space into three parts, each of which corresponds to a different scaling factor and has a different resolution. Taking an (8, 3) PINT as an example, we depict its basic format in Fig. 2. Regardless of the tensor-shared scaling factor, the integer value $x_p$ of the PINT is acquired by

$$x_p = \begin{cases} \text{Low bits signed integer}, & \text{flag} = 0 \text{ and } \odot(\text{HB}) = 1 \\ \text{Signed integer} \cdot 2^{d}, & \text{flag} = 1 \\ \text{Signed integer} \cdot 2^{k-2}, & \text{otherwise} \end{cases} \tag{9}$$

where $\odot(\text{HB}) = 1$ means that HBs are all "0" or all "1." In this case, only LBs are effective. From (9), we can deduce that $k$-bit PINT could cover a dynamic range of $[-2^{2(k-2)}, 2^{2(k-2)} - 1]$, which is the same as that of the $(2k-3)$-bit INT.

To quantitatively compare INT and PINT, we use decimal accuracy to measure their representation abilities. The decimal accuracy is given by [45]

$$-\log_{10}\left(\left|\log_{10}\left(\frac{q}{x}\right)\right|\right) \tag{10}$$

where $x$ is a real number and $q$ is the quantized value of $x$. For each data type, we create a number set, including all positive points. Then, these sets are interpolated uniformly and used to compute decimal accuracies. We plot the decimal accuracies in Fig. 3, in which the $x$-axis denotes the magnitude of data. As shown in Fig. 3, the accuracy of INT drops linearly as the value decreases. By contrast, the accuracy curve of PINT

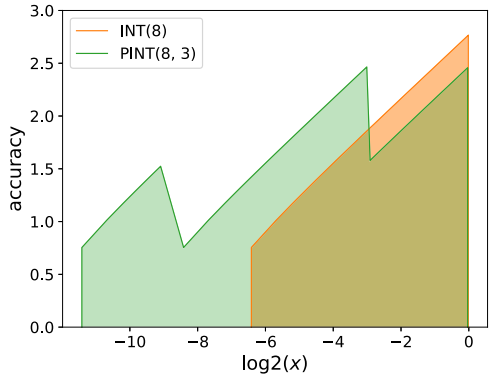



Fig. 3. Decimal accuracies of INT and PINT.

is approximately equal to a piecewise linear function of $x$, hence its name. Compared with INT, PINT not only could cover a broader range but also has a more reasonable precision distribution, where more data with larger magnitudes have relatively higher precision.

When used in DNN training, the elaborate PINT quantization process is illustrated in Algorithm 1. Given the bit width $k$ and split point $d$, the PINT is defined, and the layerwise scaling factors need to be calculated at first (Lines 1–6). To make the computations more friendly in the hardware implementation, we force all scaling factors to be powers of 2 (Line 1). Hence, a simple shift operation could complete the scaling of data. Next, according to the magnitude of $x$, we could decide which segment of PINT encoding space the data belong to and select the proper quantization resolutions (Lines 7–18). The stochastic rounding method [35], i.e., Sround($x$), is adopted to ensure unbiased quantization and improve the training stability, which can be implemented by a simple linear-feedback shift register (LFSR)

$$\text{Sround}(x) = \begin{cases} \lfloor x \rfloor, & \text{w.p. } \lceil x \rceil - x \\ \lceil x \rceil, & \text{w.p. } x - \lfloor x \rfloor. \end{cases}$$

### B. L1-Norm Filter Response Normalization

Compared with the original BN, the filter response normalization (FRN) [46] eliminates the batch dependence in the training procedure. DNNs can be successfully trained with arbitrary batch size while achieving the same performance as BN. From a global perspective, it becomes practical to train DNNs with a small batch size on edge devices. From a local perspective, in one layer, operated on a per-channel basis, the normalization does not need to wait that the preceding CONV layer finishes the calculation of a whole mini-batch, and the intermediate data needed to be buffered are reduced. Given an input feature vector $\boldsymbol{x} = \boldsymbol{X}_{b,:,:,c} \in \mathbb{R}^N$, where $\boldsymbol{X}$ is the input tensor with shape $[B, H, W, C]$ and $N = H \times W$, the FRN is defined as

$$\nu^2 = \frac{\Sigma_i x_i^2}{N}$$
$$y_i = \gamma \frac{x_i}{\sqrt{\nu^2 + \epsilon}} + \beta \tag{11}$$

where $\epsilon$ is a small positive constant around $10^{-5}$ to prevent division by zero and $\gamma$ and $\beta$ are learnable parameters.

**Algorithm 1** PINT Quantization Algorithm

**Require:** $\boldsymbol{x} \in \mathbb{R}$, word size $k \in \mathbb{N}^+$, and split point $d \in \mathbb{N}^+$

1: $r_1 = 2^{\lceil \log_2(\max |\boldsymbol{x}|) \rceil}$
2: $s_1 = r_1 \div 2^{k-2}$
3: $r_2 = r_1 \div 2^{k-2-d}$
4: $s_2 = r_2 \div 2^{k-2}$
5: $r_3 = r_2 \div 2^{k-2}$
6: $s_3 = r_3 \div 2^{d}$
7: **if** Abs($x$) $> r_2$ **then**
8: $\quad q = \text{Clamp}(\text{Sround}(x \div s_1), -2^{k-2}, 2^{k-2} - 1)$
9: $\quad \hat{x} = q \times s_1$
10: **else**
11: $\quad$ **if** Abs($x$) $> r_3$ **then**
12: $\quad\quad q = \text{Clamp}(\text{Sround}(x \div s_2), -2^{k-2}, 2^{k-2} - 1)$
13: $\quad\quad \hat{x} = q \times s_2$
14: $\quad$ **else**
15: $\quad\quad q = \text{Clamp}(\text{Sround}(x \div s_3), -2^{d}, 2^{d} - 1)$
16: $\quad\quad \hat{x} = q \times s_3$
17: $\quad$ **end if**
18: **end if**

Nevertheless, the complicated operations, such as square and sqrt, still hinder the efficient implementation of the normalization layer. Therefore, based on FRN, we propose a more hardware-friendly normalization method, L1-FRN, which uses the L1-norm to replace the mean squared norm $\nu$ in (11). Correspondingly, the L1-FRN is defined as

$$\nu = \frac{\Sigma_i |x_i|}{N}$$
$$y_i = \gamma \frac{x_i}{\nu + \epsilon} + \beta. \tag{12}$$

In addition, we introduce the TLU with a learnable threshold $\tau$ to mitigate the biased activations because of the absence of mean centering [46]. The definition of TLU is shown in the following equation:

$$z = \max(\boldsymbol{y}, \tau) \tag{13}$$

where $\tau$ is a learnable parameter like other parameters that can be updated by gradient descent algorithms.

The training process of the L1-FRN is presented in Algorithm 2. To unify both the FP and BP phases into one hardware architecture, we extract basic operations and optimize the computational flow. Meanwhile, data reuse is fully exploited to reduce computational complexity. The details of the hardware implementation will be introduced in Section IV-C.

### C. Experimental Results

To illustrate the effectiveness and efficiency of our proposed training algorithm, we have performed extensive experiments over a variety of models on image classification tasks. The L1-norm FRN is adopted to replace the conventional L2-norm BN. All involved data (i.e., W, A, G, and E) are represented in PINT(8, 3). We use the extended Pytorch framework to simulate the training procedure on NVIDIA Tesla V100 GPUs. The experimental configurations are as follows.

TABLE II
EXPERIMENTAL RESULTS AND COMPARISONS WITH PRIOR 8-BIT TRAINING SCHEMES

| Dataset | Model | Method | Baseline | Accuracy | Accuracy Loss | Normalization |
|---|---|---|---|---|---|---|
| CIFAR-10 | VGG16 | NITI [47] | 91.41 | 87.94 | 3.47 | None |
| | | PINT8 | 93.86 | 93.55 | 0.31 | Quantized L1-FRN |
| | ResNet20 | UINT8 [31] | 92.32 | 91.95 | 0.37 | L2-BN |
| | | PINT8 | 92.10 | 91.93 | 0.17 | Quantized L1-FRN |
| | InceptionV3 | UINT8 [31] | 94.89 | 95.00 | -0.11 | L2-BN |
| | | PINT8 | 94.83 | 94.59 | 0.24 | Quantized L1-FRN |
| ImageNet | ResNet18 | UINT8 [31] | 70.30 | 69.67 | 0.63 | L2-BN |
| | | WAGEUBN [26] | 68.70 | 63.62 | 5.08 | Quantized L2-BN |
| | | PINT8 | 70.90 | 70.55 | 0.35 | Quantized L1-FRN |
| | ResNet50 | UINT8 [31] | 76.60 | 76.34 | 0.26 | L2-BN |
| | | WAGEUBN [26] | 74.66 | 67.95 | 6.71 | Quantized L2-BN |
| | | PINT8 | 77.11 | 77.02 | 0.09 | Quantized L1-FRN |

**Algorithm 2** L1-FRN Training Process

**1. Forward Propagation:**

**Require:** An input feature vector $\boldsymbol{x}$; Learnable parameters $\gamma, \beta, \tau$.

$\nu \leftarrow \frac{1}{N}\Sigma_{i=1}^{N}|x_i|$
$\lambda = 1/(\nu + \epsilon)$
$\hat{x}_i \leftarrow x_i \times \lambda$
$y_i \leftarrow \hat{x}_i \times \gamma + \beta$
$b_i = \mathbb{I}(y_i >= \tau)$ // TLU bitmap.
$z_i = y_i \times b_i$
Pass $\boldsymbol{z}$ to the next layer.
Store $\lambda, \hat{\boldsymbol{x}}, \boldsymbol{b}$ for backward.

**2. Backward Propagation:**

**Require:** An error feature vector $\mathbf{d}\boldsymbol{z} = \boldsymbol{E}_{b,:,:,c} \in \mathbb{R}^N$; Stored data $\lambda, \hat{\boldsymbol{x}}, \boldsymbol{b}$ during FP.

$\mathrm{d}\tau \leftarrow \Sigma_{i=1}^{N}\mathrm{d}z_i \times (1 - b_i)$
$\mathrm{d}y_i \leftarrow \mathrm{d}z_i \times b_i$
$\mathrm{d}\beta \leftarrow \Sigma_{i=1}^{N}\mathrm{d}y_i$
$\mathrm{d}\gamma \leftarrow \Sigma_{i=1}^{N}\mathrm{d}y_i \times \hat{x}_i$
mean $\leftarrow \frac{\mathrm{d}\gamma}{N}$
$\gamma\,\lambda \leftarrow \gamma \times \lambda$
$\mathrm{d}x_i \leftarrow \gamma\,\lambda \times (\mathrm{d}y_i - \mathrm{sign}(\hat{x}_i) \times \text{mean})$
Pass $\mathbf{d}\boldsymbol{x}$ to the previous layer.

1) CIFAR-10 dataset contains 32 × 32 pixels images with ten classes, in which 50 000 pictures for training and 10 000 pictures for testing. We conduct experiments on CIFAR-10 with the same hyperparameters for different networks. All models (VGG16, ResNet20, and InceptionV3) use SGD with momentum 0.9 as the optimizer and are trained for 200 epochs. The learning rate is linearly increased to 0.1 in five epochs as a warm-up and then decreased with the cosine annealing strategy [48].
2) ImageNet dataset includes images of 1000 classes totally, where 1 million pictures for training and 50 000 pictures for validation. We evaluated ResNet18 and ResNet50 for ImageNet. Both models use SGD with momentum 0.9 as the optimizer and are trained for 120 epochs. The learning rate is linearly increased to 0.1 × batch_size/256 in five epochs as a warm-up and then decreased with the cosine annealing strategy.

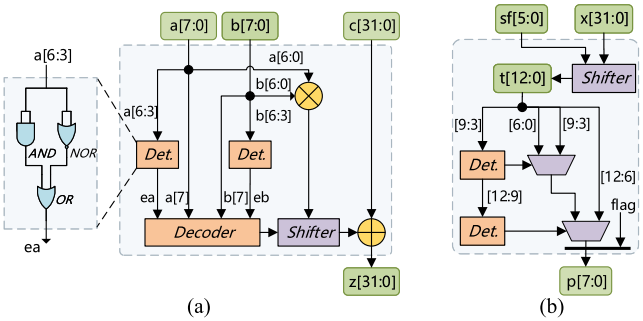



Fig. 4. Microarchitecture of PINT Arithmetic. Det. refers to the detector. (a) PINT MAC. (b) PINT conversion.

The experimental results and comparisons with prior 8-bit training schemes are summarized in Table II. Compared with the FP32 baseline, our training approach could achieve less than 0.5% accuracy degradation for various tasks. Due to differences in training scripts, the baseline accuracies among the methods are different. Thus, we add the accuracy loss as the metrics. It is shown that our approach significantly outperforms WAGEUBN [26] and NITI [47] with about more than 4% accuracy preservations. Although UNIT8 keeps BN layers in the FP32 format, our strategy performs better than it in most cases.

### D. Design of Dedicated Arithmetic Unit

The above experimental results fully confirm the representation ability of 8-bit PINT in DNN training. In general, 8-bit signed integer multipliers and 32-bit adders are used to build DNN accelerators when data are represented in INT8. Being different from this, PINT8 needs a dedicated multiplier to support its computation. The microarchitecture of a PINT(8, 3) multiplier is shown in Fig. 4, which comprises a 7-bit signed integer multiplier, a left shifter, two detectors, and a decoder. A detector determines whether HBs are all "1" or all "0," which includes an AND gate, a NOR gate, and an OR gate. Taking the outputs of two detectors (ea and eb) and *flag* bits as input, the decoder is used to determine how to shift the product. Then, the product is accumulated by a 32-bit integer adder.

In addition, a conversion unit shown in Fig. 4(b) is proposed to convert the final results from INT32 to PINT(8, 3) before

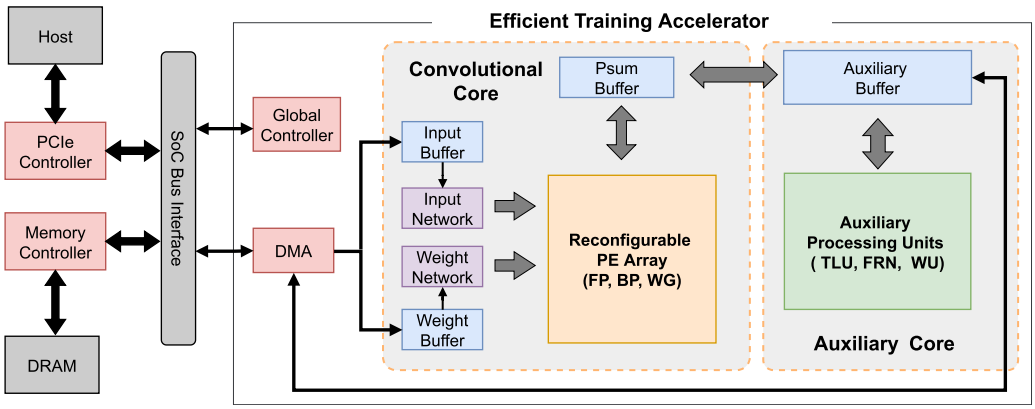



Fig. 5. Overall architecture of ETA.

sending them to the external memory. The 32-bit input $x$ is first shifted and rounded into a 13-bit temp value $t$ based on the scaling factor $sf$. Then, two detectors are used to determine the numerical range of input and select the corresponding output $p$ from three overlapped fields.

## IV. Efficient DNN Training Architecture

### A. Overall Architecture

In this section, we propose a reconfigurable training accelerator, named ETA, which could efficiently perform the training of modern DNNs (e.g., ResNet) on FPGA. The top-level diagram of our design is shown in Fig. 5. The global controller coordinates all modules to complete different phases of the training process of DNNs. All weight parameters, intermediate features, and gradients are stored in DRAM. DMA is used to transfer data tiles between on-chip buffers and DRAM, which is independent of the host. Buffers store input activations, weights, partial sums, and data of auxiliary layers on chip.

A C-Core is responsible for operating FP, BP, and WG phases of key layers, including CONV and FC layers. A C-Core consists of an array of PEs, accumulation units, NoC, and data buffers. Each PE, which contains three multiplication and accumulation (MAC) units, is designed to perform various 1-D convolutions using a reconfigurable datapath. To exploit computational parallelism and data reuse as much as possible, PE units are arranged in a hierarchical structure with a shape of $8 \times 3 \times 3 \times 8$. To deal with the complex data access patterns, multibanked data buffers equipped with flexible NoC are designed to transfer data to the PE array. Details of the C-Core are described in Section IV-B.

An A-Core is developed to perform the WU process and all operations in auxiliary layers. According to the experimental settings in Section III, the SGD with momentum algorithm is utilized for the WU process. Besides, L1-FRN and TLU are adopted for normalization and activation, respectively. All these layers are extracted into several basic operations, and then, they can be executed in a time-multiplexed manner using a small number of hardware resources. The detailed design of the A-Core is introduced in Section IV-C.

The accelerator conducts the training process layer by layer. When the accelerator works, data tiles are loaded from DRAM to on-chip buffers of the C-Core at first. Based on the layer information, the NoC sends data to the PE array, and then, the PE array works with the configured datapath and conducts the corresponding computations. The results obtained in the C-Core are streamed to A-Core to be processed, and intermediate data are stored in the auxiliary data buffer. During the FP phase of training, the A-Core receives the outputs from the C-Core and performs the subsequent L1-FRN and TLU layers. As for the BP phase, the backpropagation of the front TLU and L1-FRN layers is executed successively. When the WG phase is completed in the C-Core, the A-Core uses the generated weight gradients to update new velocities and new weights. If necessary, the final results of the A-Core will be written to DRAM for later usage.

### B. Design of C-Core

*1) Reconfigurable Computational Patterns:* The C-Core is a homogeneous architecture that processes FP, BP, and WG phases for all kinds of key layers, where there exist obvious diversities in terms of computational characteristics. To clearly demonstrate the computational process of different kinds of CONV layers, we perform them in a 1-D convolution manner. Based on the layer configurations, including kernel size, stride, and phase during training, we categorize CONV layers into different computational patterns. Here, we take a 1-D convolution with a $1 \times 3$ kernel as an example and then illustrate these computational patterns and the respective feasible hardware implementations in the diagram in Fig. 6. The detailed descriptions of each pattern are discussed as follows.

1) *FP, Stride = 1:* When the sliding stride is equal to 1, the computational pattern is intuitive and straightforward during FP, where a traditional convolutional operation is executed at spatial dimensions in Fig. 6(a). Correspondingly, a typical convolver architecture working in weight-stationary dataflow shown in Fig. 6(b) is adequate to process this situation efficiently.
2) *BP, stride = 1:* The computation flow in BP shown in Fig. 6(c) is similar to the case in FP when stride = 1. Nevertheless, we need to note that the kernel must be reversed (rotated by 180° for 2-D convolution) to obtain

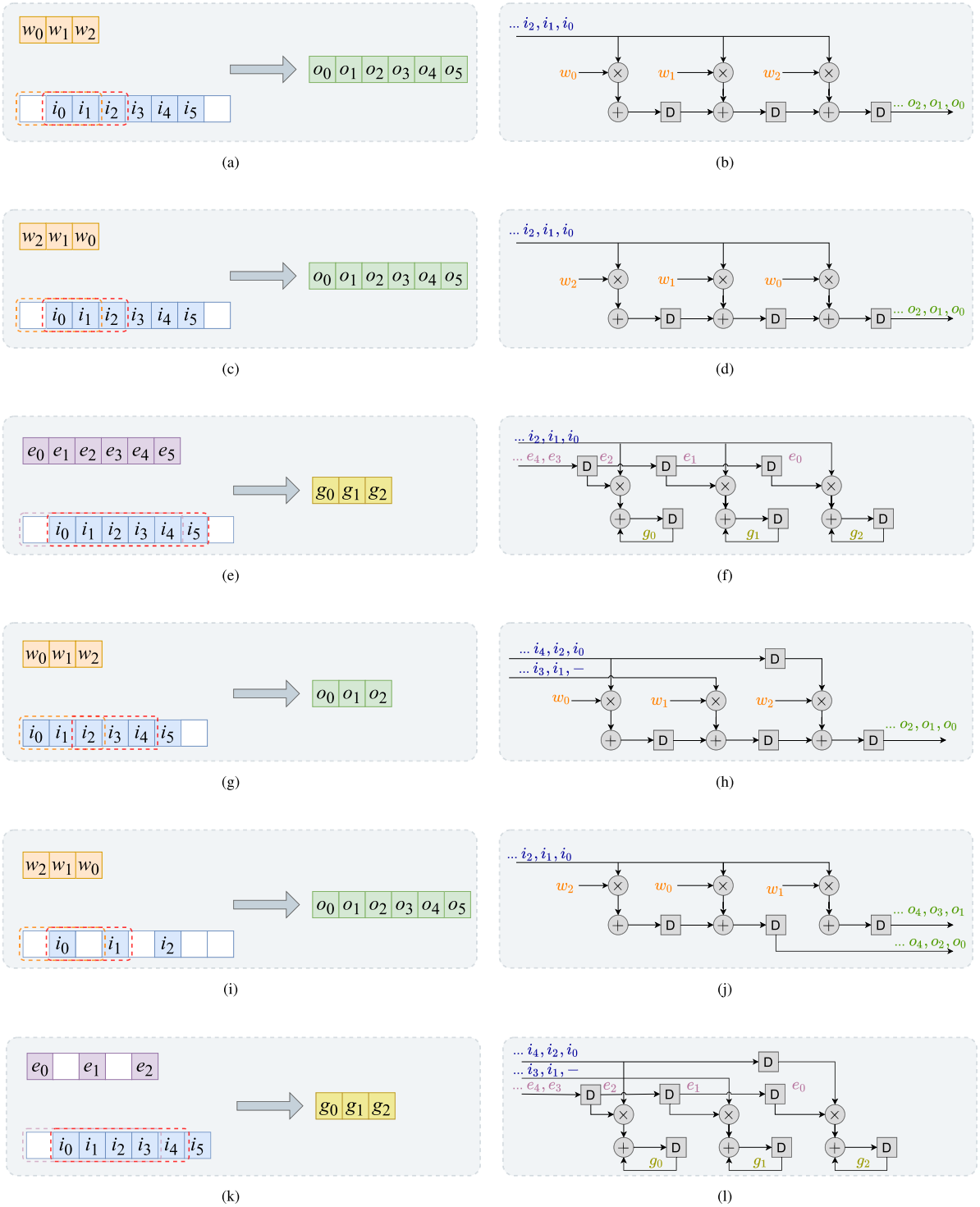



Fig. 6. Computational patterns and feasible hardware implementations in training process, which takes the 1-D convolution with $1 \times 3$ weight as an example. (a) and (b) FP, stride $= 1$. (c) and (d) BP, stride $= 1$. (e) and (f) WG, stride $= 1$. (g) and (h) FP, stride $= 2$. (i) and (j) BP, stride $= 2$. (k) and (l) WG, stride $= 2$.

correct results in the BP phase, which can be dealt with by the weight network.

3) *WG, Stride = 1:* The computation during WG can also be formulated into convolution operation, as shown in Fig. 6(e). Being different from FP and BP, WG performs the convolution between error maps and feature maps. There is a large window convolution, and the window sizes vary in different layers. Considering that, we process this pattern in an output-stationary way as in Fig. 6(f) to fully utilize data reuse and avoid the shape problem.

4) *FP, Stride = 2:* As shown in Fig. 6(g), an input-skip operation is conducted to downsample the input feature map, so the size of outputs is smaller than that of inputs. To meet the characteristic of the downsampling operation, we split one-row inputs into two groups based

on coordinates, i.e., even input group and odd input group, which are sent to different MACs in parallel, as presented in Fig. 6(h).

5) *BP, Stride = 2:* As shown in Fig. 6(i), a zero-inserting operation is conducted to enlarge the size of the input error map before calculation, and then, weights are convolved with the enlarged inputs. As a result, the size of outputs is large than that of inputs. As we all know, calculations with zeros are unnecessary. To avoid the zero-inserting operations and ensure the right results, we split one-row outputs into two groups based on their coordinates, i.e., even output group and odd output group, which can be obtained from different MACs in parallel, as shown in Fig. 6(j).
6) *WG, Stride = 2:* As for the WG phase with a stride of 2, the zero-inserting operations are also needed for error maps. In the WG phase, the error maps play the role of weight kernels in the convolution, as shown in Fig. 6(k). Fig. 6(l) shows that we adopt the input division method to avoid zero-inserting operations, just like what happens in the FP phase with stride = 2.
7) *Kernel Size = 1:* Here, we categorize all cases involving $1 \times 1$ CONV and FC layers into this computational pattern. Since there is no partial sum reduction across spatial dimensions and the input-skip and zero-inserting operations can be easily realized by flexible memory access, the working phases and stride sizes do not influence the computational efficiency. We utilize output-stationary dataflow for all phases. The input data are broadcasted to MACs, and different weights are transmitted to three MACs in parallel.

In conclusion, we summarize seven computational patterns in the DNN training. Under the guidance of the above analysis, a reconfigurable PE is developed.

*2) Microarchitecture of PE:* The basic PE unit shown in Fig. 7 is composed of three MAC units, several multiplexers, and an input first input first output (FIFO). Multiplexers are used to determine the operating mode of the PE based on layer configurations. The input FIFO stores the split data groups and prepares them for calculations.

A PE performs CONV layers in the 1-D convolution manner with the flexible dataflow for various computational patterns. Inside each PE, the computation is performed in the weight-stationary dataflow during FP and BP phases for CONV layers owning $3 \times 3$ kernels. For the WG phase and $1 \times 1$ CONV layers, the computation is performed in the output-stationary dataflow.

Benefitted from the data split scheme, the redundant operations coming from input data skipping or zero data inserting can be eliminated in CONV layers whose strides are equal to 2. As a result, almost $4\times$ speedup can be achieved in these layers. Therefore, our reconfigurable PE can achieve high utilization of computation resources during the whole process of training.

*3) Structure of PE Array:* To fully utilize the data reuse both in the spatial and temporal dimensions, we construct a hierarchical structure of PE units, as shown in Fig. 8.

There exist three window processing units (WPUs) in a PE cluster, and each WPU comprises a $3 \times 8$ PE array. Inside

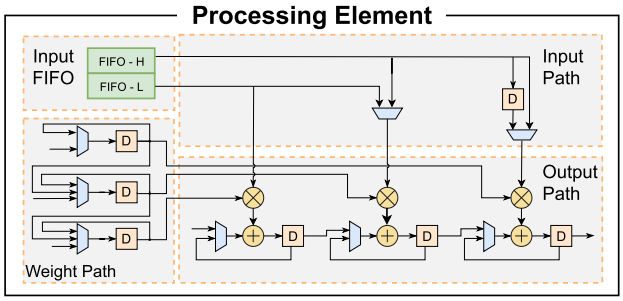



Fig. 7. Microarchitecture of PE.

one WPU, different input channels are unrolled along the row direction of PEs. An accumulator merges the partial sums from PEs in one row and then writes the results into the partial sum buffer. Different PE rows are used to process different rows of the input feature map. At the same time, the weight data are streamed along the column direction in a systolic way. Different WPUs share the same weights and take different convolutional windows of the input feature map to exploit the convolutional reuse. A C-Core contains eight PE clusters in total, each of which shares the same input tile and holds the individual weights from different output channels.

Consequently, various data reuse opportunities are exploited extensively. The convolutional reuse is achieved inside each PE and among WPUs in one PE cluster. The partial sum reuse exists in each PE and each WPU. Meanwhile, the input data are reused among different PE clusters.

*4) Memory Management and NoC:* To support various computational patterns and ensure high PE utilization in the computing module, the memory module has to provide enough bandwidth and flexible data delivery patterns. Therefore, C-Core adopts multibank buffers to store all kinds of on-chip data, including input feature maps, weights, and partial sums. The corresponding interconnected networks are also designed to deliver data between buffers and PE array.

*a) Data buffers:* As shown in Fig. 8, each entry of the input buffer stores two input blocks from adjacent coordinates in one row to meet the requirements of the data split strategy, and each input block consists of eight input channels. Hence, 16 input elements can be fetched simultaneously in one cycle from a bank. The tile index increases with the increment of the memory address. Correspondingly, each entry of the weight buffer is also composed of eight input channels. Different columns and output channels are distributed in banks. Each physical bank is uniquely identified by an index, which is determined by the HBs of the address of an SRAM bank.

*b) Input network:* The input network is designed to reuse data that read from the input buffer as many times as possible, thereby reducing the number of memory accesses and decreasing the total power consumption. The neighboring convolution windows along the column direction share common rows; therefore, the calculations for these windows are performed simultaneously, and a higher level of data reuse will be achieved. Moreover, the input network makes it possible to access different physical banks to complete the window

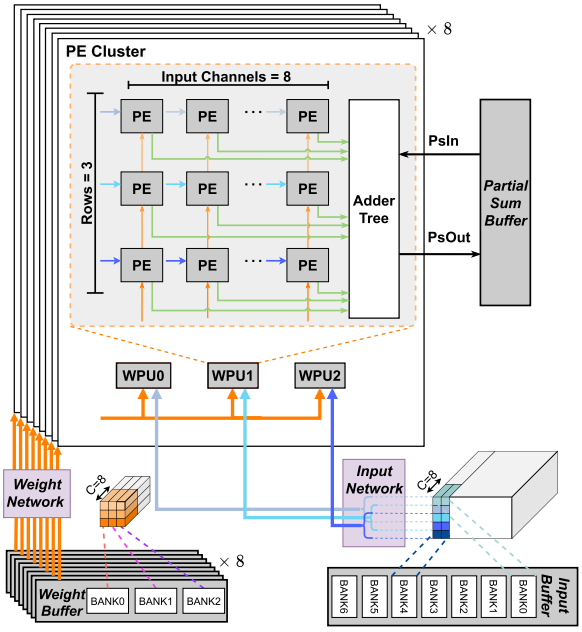



Fig. 8. Diagram of the PE array and memory management.

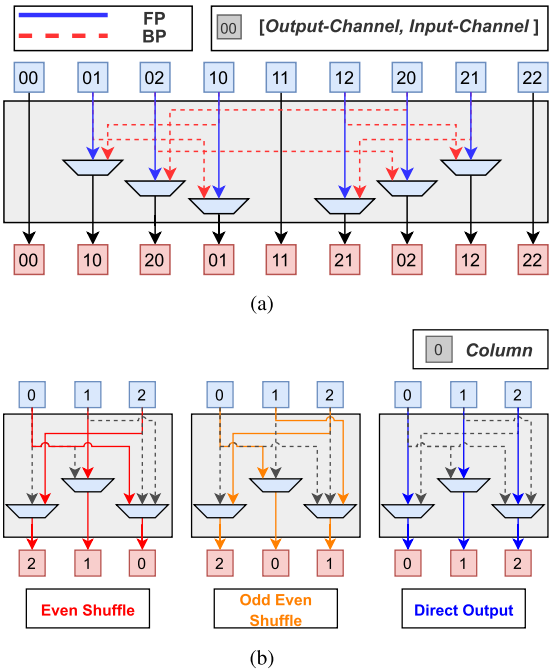



Fig. 9. Two-level weight network. (a) First level of weight network. (b) Second level of weight network.

sliding process in the column direction without moving the data between banks.

*c) Weight network:* The FP is performed by the convolution of weight kernels and input activations, but the BP is computed by the convolution of rotated weight kernels and the local gradients. In addition, it is also necessary to exchange the order of input channels and output channels during the BP phase. Therefore, the weight needs to be read in different ways for FP and BP processes. As shown in Fig. 9, a two-level network is proposed to avoid the conflict of data access or the duplication of weight storage.

In the first level, the transposition between input channels and output channels is achieved. There are two modes for weight delivery: direct mode and transposed mode. During the FP phase, weights are directly sent to PE clusters in the original data layout. When the network works in transposed mode during the BP phase, weights read from SRAMs are reorganized into the layout of the output channel first and then transfer to the next level. The second level of the weight network arranges the order of weights in one row belonging to one PE, which is consistent with the required computational pattern. This component has three operating modes, including direct mode, even shuffle mode, and odd–even shuffle mode. During the BP phase, the module works in the even shuffle mode when the stride is equal to 1 and works in the odd–even shuffle mode when the stride is equal to 2; otherwise, weights are directly delivered to the PE array.

### C. Design of A-Core

Being consistent with the structure of PE array in the C-Core, the A-Core includes eight auxiliary processing units (APUs) and an auxiliary buffer. Each APU individually performs the computations associated with one output channel of auxiliary layers, including L1-FRN, TLU, and WU. After thoroughly analyzing the computational flow in these layers, we extract six basic operations and build the respective hardware circuits. As shown in Fig. 10(a), these operations are "Abs," "Division," "Accumulation," "Multiplication-Addition," "Multiplication," and "Comparison."

An APU is composed of a local controller, a register file, an organization network, and an operation block containing those basic operations. The controller is used to configure the registers and the organization network. Accordingly, the computing resources (registers and basic operations) can be efficiently organized into arbitrary forms required by the workloads. The APU works in a time-multiplexed manner for different workloads, and thereby, significant resource savings are achieved. If multiple basic operations are selected at the same time, they will work in a fully pipelined manner to improve the processing speed.

Taking the WU process in 7 as an example, the basic operations are organized in the order of "OP4 → OP5 → OP3," where OP3 only performs the addition. New velocities can be exported from "OP4" after a multiplication–addition calculation, and new weights can be obtained after two consecutive multiplication–addition calculations.

## V. Evaluation

### A. Experimental Setup

We describe our accelerator in Verilog HDL and then synthesize, place, and route it using Vivado Design Suite 2018.3. We choose the Xilinx VC709 (Virtex7 XC7VX690T) as the target hardware platform, which is equipped with 443.2k lookup tables (LUTs), 886.4k filp-flops (FFs), 1470 BRAM blocks, 3600 DSP blocks, and two 4-GB DDR3 memory with 29.9-GB/s bandwidth in total. The accelerator runs at 200-MHz frequency.

TABLE III
COMPARISON WITH PREVIOUS WORKS

| | [38] | [39] | [40] | [51] | [42] | [50] | [44] | | ETA | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Device | Stratix V | ZU19EG | VC709 | PYNQ-Z1 | KCU1500 | ZCU111 | Stratix 10 MX | | VC709 | | |
| Precision | FP32 | FP32 | INT16 | INT16 | INT8 | INT8 | FP16 | | PINT8 | | |
| Frequency (MHz) | 150 | 200 | - | 50 | 250 | 180 | 185 | | 200 | | |
| Dataset | MNIST | CIFAR-10 | ImageNet | CIFAR-10 | CIFAR-10 | CIFAR-10 | CIFAR-10 | | CIFAR-10 | | ImageNet |
| Model | LeNet-5 | LeNet-10 | AlexNet | MobileNet-V1 | VGG-like | VGG-16 | VGG-like | ResNet20 | VGG-16 | ResNet20 | ResNet18 |
| Normalization | No | No | No | Yes | No | No | No | No | Yes | | |
| DSP | - | 1500(76.22%) | - | 96(43.6%) | 1030(19%) | 1037(24.69%) | 1046(26%) | 1040(26%) | 1728(48%) | | |
| Logic Element† | - | 33K(63.00%) | - | 36K(67%) | 199K(30%) | 73.1K(17.18%) | 221K(31%) | 239K(34%) | 132K(30.39%) | | |
| BRAM‡ | - | 174(17.68%) | - | - | 1060(49%) | 1045(96.76%) | 2998(44%) | 2558(37%) | **240**(13.60%) | | |
| Throughput (GOPS) | 7.01 | 86.12 | 1022 (Per FPGA) | 10.0 | 641.1 | - | 158.54 | 180.33* | 610.98 | 658.64* | 811.25* |
| Performance (images/s) | 7.0 | - | - | - | - | 3.3 | - | - | 326.5 | 2352.9 | 59.2 |
| Power (W) | 27.30 | 14.20 | 32.00 | 0.346 | 26.8 | - | 20 | 20 | 8.44 | 8.55 | 8.64 |
| Energy Eff. (GOPS/W) | 0.27 | 6.05 | 31.97 | 28.90 | 24.01 | - | 7.93 | 9.02 | **72.37** | **77.04** | **93.86** |

* zero-inserting operations in CONV layers with stride=2 are taken into account.
† Xilinx FPGA in LUTs and Intel FPGAs in ALMs.
‡ Xilinx FPGA in BRAMs (36Kb) and Intel FPGAs in M20K RAMs (20Kb).

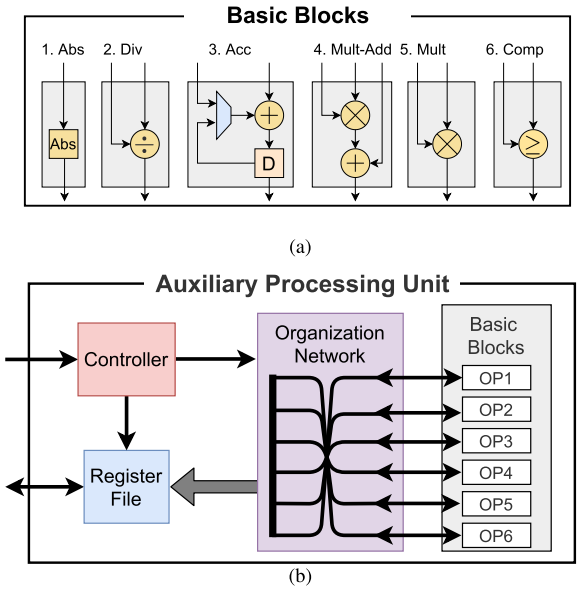



Fig. 10. Architecture of APU and its basic operations. (a) Basic blocks. (b) APU.

We evaluate ETA on three CNN benchmarks: VGG16 and ResNet20 on the CIFAR-10 dataset as well as ResNet18 on the ImageNet dataset. The 8-bit PINT data format is applied in the DNN training to reduce the memory footprint and simplify the arithmetic complexity. The detailed configurations of data representation are consistent with descriptions in Section III-C.

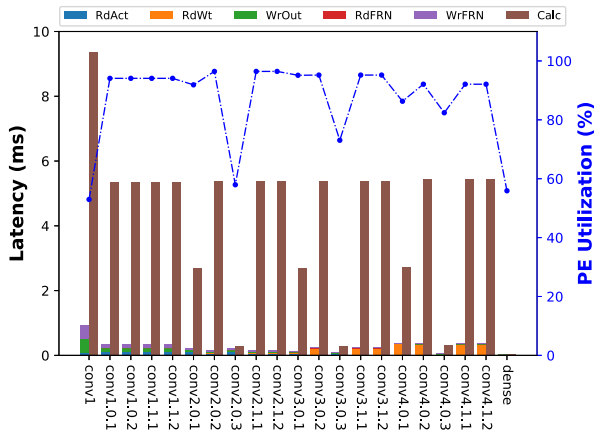
We conduct the functional simulation with the extracted actual data from benchmarks and measure the processing latency of the training accelerator. Meanwhile, we generate the annotated toggle rate from the simulator and dump it into switching activity interchange format (SAIF). Then, FPGA power consumption is estimated by incorporating the SAIF file into Vivado Power Analysis Tool.

### *B. Results and Analysis*

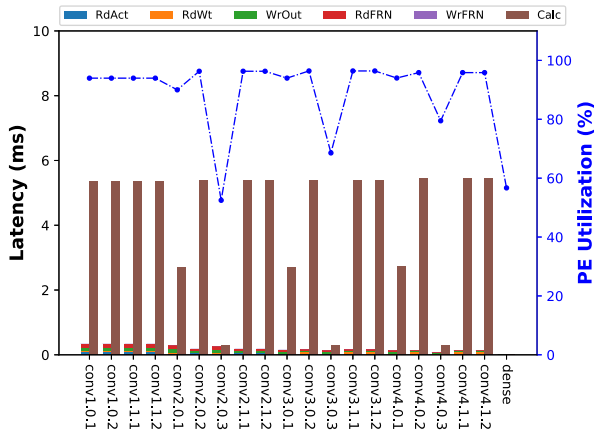
In general, the hardware performance is bounded by the computational resource or memory bandwidth [51]. Benefitted from the joint optimization of hardware and algorithm, the bandwidth requirement is significantly reduced, and the PE utilization maintains at a high level. Figs. 11–13 show the latency breakdowns of different layers during different training phases of ResNet18, CIFAR-ResNet20, and CIFAR-VGG16, respectively. In these three figures, *Calc* denotes the time consumption of calculations, and the others represent the transaction time between DRAM and on-chip buffers. *RdAct*, *RdWt*, and *WrOut* come from reading inputs, reading weights, and writing activated outputs, respectively. *RdFRN* and *WrFRN* include all parameters and intermediate data that need to be read and written in FRN layers. *RdWU* and *WrWU* cover the time for reading old weights as well as old velocities and writing new velocities, respectively.

As shown in Fig. 11, PE utilizations of most layers exceed 95% except the first layer, some $1 \times 1$ CONV layers, and the last dense layer. Because the first layer has a small number of input channels (3) and a large kernel size ($7 \times 7$), the workload cannot be fully mapped to the PE array. Therefore, the hardware performance is degraded. As for $1 \times 1$ CONV layers and the dense layer, it is obvious that the DRAM access occupies a considerable percentage of time (18%–42%), which is because the data reuse opportunity is much lower than that of other layers. This problem can be alleviated by using devices equipped with HBM. Since these layers account for only a small part of the overall calculation, the average PE utilization still achieves up to 91%, and the average throughput achieves 811.25 GOPS.

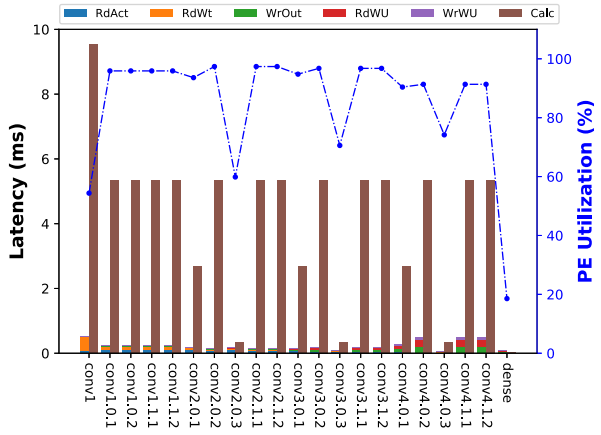
When our accelerator trains networks on CIFAR, we can see that the impact from DRAM access has increased in Figs. 12 and 13. Because the size of input images is much smaller than that of ImageNet, the opportunity for data reuse is reduced. For CIFAR-ResNet20, the performance is mainly




(a)

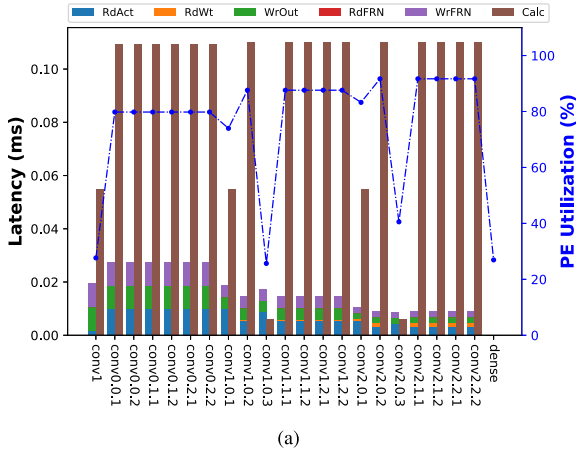



(b)

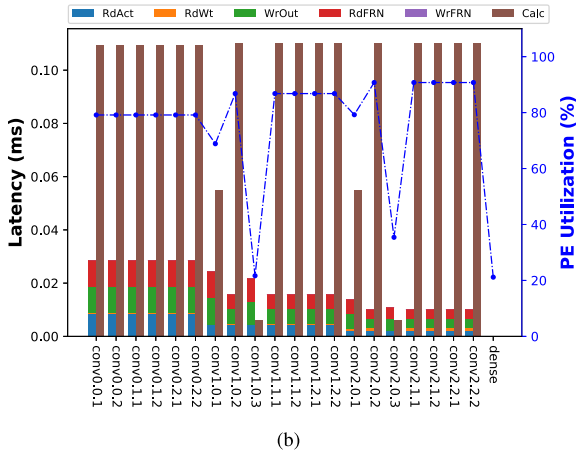



(c)

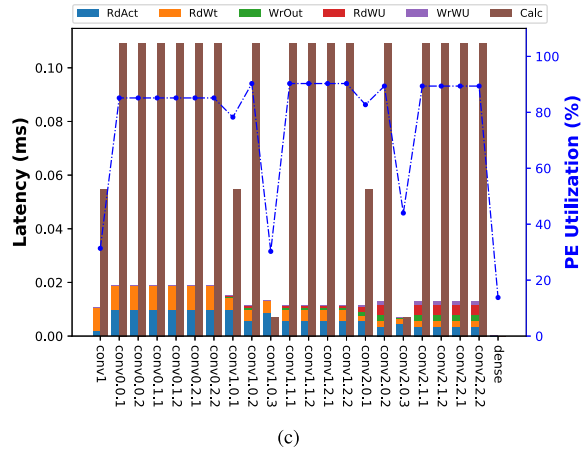
Fig. 11. Latency breakdowns of different training phases of ResNet18. (a) FP. (b) BP. (c) WG.




(a)




(b)




(c)

Fig. 12. Latency breakdowns of different training phases of CIFAR-ResNet20. (a) FP. (b) BP. (c) WG.

limited by the access to feature maps, which is more severe during the FP and BP phases. By contrast, the performance bound of CIFAR-VGG16 exists in the access to weights since CIFAR-VGG16 is an overparameterized network, where the WG phase is the most affected. The situation can be mitigated by increasing the batch size to improve the weight reuse. As a result, ETA achieves 658.64 GOPS with 83% PE utilization on CIFAR-ResNet20 and 610.98 GOPS with 88% PE utilization on CIFAR-VGG16.

### C. Comparisons With Previous Works

Comparisons between our ETA and previous training accelerators on FPGAs are summarized in Table III. Due to different benchmarks and platforms, it is not easy to make

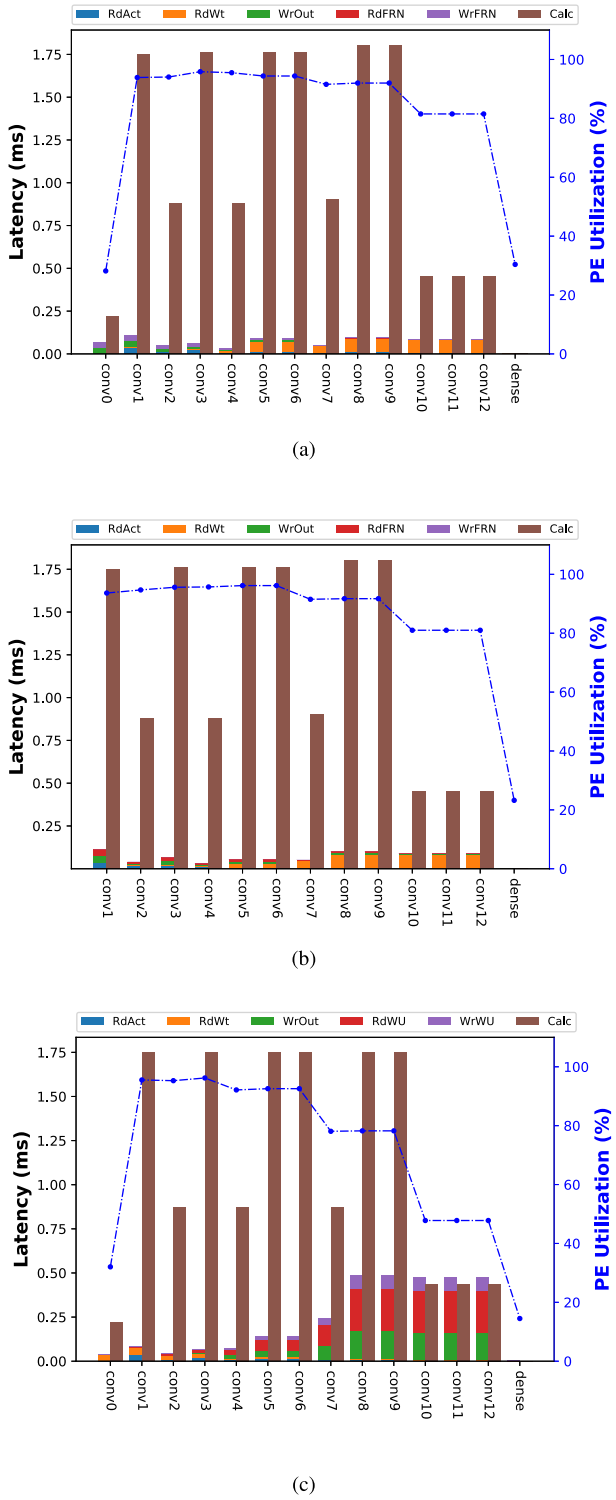



Fig. 13. Latency breakdowns of different training phases of CIFAR-VGG16. (a) FP. (b) BP. (c) WG.

fair comparisons. Nevertheless, our design shows evident superiority in both function and performance.

Accelerators in [38] and [39] adopt FP32 to represent data and are only evaluated on small models. Thus, ETA significantly outperforms them in terms of performance and efficiency by utilizing the proposed 8-bit training strategy. By using a multi-FPGA cluster, FPDeep [40] could store all weights in on-chip buffers and implement dedicated computing units for each layer independently. Therefore, their design shows higher throughput than ours. However, their design consumes too many resources (ten FPGAs) and lacks flexibility, which is not suitable for edge computing. The 8-bit training accelerator [50] spends too much time on software processing, which severely limits its overall performance. By deploying the entire training process on the FPGA platform, ETA shows 99× faster execution speed than it on the same network. Guo *et al.* [42] proposed a sparse training accelerator by exploiting pruning and 8-bit quantization techniques, which can achieve comparable performance with ETA on a VGG-like model. Nevertheless, their design can only be used in the fine-tuning stage, and their energy efficiency is much lower than ours due to the higher memory overhead.

Compared with [44], a training accelerator designed for modern CNNs by utilizing HBM, working on CIFAR-ResNet20, our ETA still achieves 3.65× and 8.54× improvements in terms of throughput and energy efficiency, respectively. In Table III, PRBN [49] is the only one supporting the training of BN layers. However, they only implemented a small-scale array to show the effectiveness. Hence, the reported throughput is very low (10 GOPS), and the scalability of the design is unclear. Besides, we apply ETA on the large-scale network training and ResNet18 on ImageNet and obtain higher throughput than that on CIFAR, which has not been evaluated on the existing FPGA-based training accelerators.

In summary, the comparison results have proved that ETA is capable of efficiently training various prevailing DNNs on different datasets. Therefore, it would be promising to leverage ETA in real applications, which can handle dynamic data and maintain desirable accuracies. Furthermore, ETA can be extended to explore the feasibility on other vision tasks, such as object detection and image generation, to find more applications.

## VI. Conclusion

In this article, we propose an efficient hardware accelerator for DNN training based on hardware-algorithm co-optimization. An efficient training algorithm is developed by utilizing a novel PINT format and a hardware-friendly normalization layer. We demonstrate that training DNNs using 8-bit PINT achieves the state-of-the-art performance across different models without batch dependence. We further design a reconfigurable training accelerator for DNNs. The C-Core can flexibly complete various computational patterns and avoid dummy computations in nontraditional CONV layers. The A-Core can execute auxiliary layers in a unified architecture and reduce the latency of the normalization operation. Due to our hardware-algorithm co-optimization approach, data accesses are remarkably reduced, and PE utilization maintains a high level. As a result, our design can efficiently support the training of prevailing DNNs, such as VGG and ResNet. The implementation results demonstrate that our design achieves up to 811.25 GOPS in throughput and 93.86 GOPS/W in energy efficiency. On the same workloads, our accelerator has achieved 3.65–99× speedup than prior designs, which sufficiently proves the superiority of our approach.

## REFERENCES


[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[2] D. Amodei *et al.*, "Deep speech 2: End-to-end speech recognition in English and mandarin," in *Proc. 33nd Int. Conf. Mach. Learn. (ICML)*, 2016, pp. 173–182.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, vol. 1, Jun. 2019, pp. 4171–4186.

[4] M. Bojarski *et al.*, "End to end learning for self-driving cars," 2016, *arXiv:1604.07316*.

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

[6] B. Li, B. Wu, J. Su, and G. Wang, "EagleEye: Fast sub-net evaluation for efficient neural network pruning," in *Eur. Conf. Comput. Vis. (ECCV)*, 2020, pp. 639–654.

[7] T. Zhang *et al.*, "StructADMM: Achieving ultrahigh efficiency in structured pruning for DNNs," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, Feb. 15, 2021, doi: 10.1109/TNNLS.2020.3045153.

[8] Y. Li *et al.*, "BRECQ: Pushing the limit of post-training quantization by block reconstruction," 2021, *arXiv:2102.05426*.

[9] R. Ni, H.-m. Chu, O. Castañeda, P.-y. Chiang, C. Studer, and T. Goldstein, "WrapNet: Neural net inference with ultra-low-resolution arithmetic," 2020, *arXiv:2007.13242*.

[10] Y. Cai, Z. Yao, Z. Dong, A. Gholami, M. W. Mahoney, and K. Keutzer, "ZeroQ: A novel zero shot quantization framework," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 13166–13175.

[11] Y. Idelbayev and M. A. Carreira-Perpinan, "Low-rank compression of neural nets: Learning the rank of each layer," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 8046–8056.

[12] N. Li, Y. Pan, Y. Chen, Z. Ding, D. Zhao, and Z. Xu, "Heuristic rank selection with progressively searching tensor ring network," *Complex Intell. Syst.*, vol. 17, pp. 1–15, Mar. 2021.

[13] X. Jiao *et al.*, "TinyBERT: Distilling BERT for natural language understanding," in *Proc. Findings Assoc. Comput. Linguistics*, 2020, pp. 4163–4174.

[14] M. Haroush, I. Hubara, E. Hoffer, and D. Soudry, "The knowledge within: Methods for data-free model compression," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 8491–8499.

[15] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[16] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 269–284, Apr. 2014.

[17] J. Wang, J. Lin, and Z. Wang, "Efficient hardware architectures for deep convolutional neural network," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 6, pp. 1941–1953, Nov. 2018.

[18] T. Yuan, W. Liu, J. Han, and F. Lombardi, "High performance CNN accelerators based on hardware and algorithm co-optimization," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 1, pp. 250–263, Jan. 2021.

[19] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, "SNAP: An efficient sparse neural acceleration processor for unstructured sparse deep neural network inference," *IEEE J. Solid-State Circuits*, vol. 56, no. 2, pp. 636–647, Feb. 2021.

[20] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Process. Mag.*, vol. 37, no. 3, pp. 50–60, May 2020.

[21] S. Choi, J. Sim, M. Kang, Y. Choi, H. Kim, and L. Kim, "An energy-efficient deep convolutional neural network training accelerator for *in situ* personalization on smart devices," *IEEE J. Solid-State Circuits*, vol. 55, pp. 2691–2702, 2020.

[22] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Netw.*, vol. 113, pp. 54–71, May 2019.

[23] F. Tu *et al.*, "Evolver: A deep learning processor with on-device quantization–voltage–frequency tuning," *IEEE J. Solid-State Circuits*, vol. 56, no. 2, pp. 658–673, Feb. 2021.

[24] P. Micikevicius *et al.*, "Mixed precision training," in *Proc. 6th Int. Conf. Learn. Represent. (ICLR)*, 2018, pp. 1–12.

[25] D. Kalamkar *et al.*, "A study of BFLOAT16 for deep learning training," 2019, *arXiv:1905.12322*.

[26] Y.-K. Yang, S. Wu, L. Deng, T. Yan, Y. Xie, and G. Li, "Training high-performance and large-scale deep neural networks with full 8-bit integers," *Neural Netw.*, vol. 125, pp. 70–82, Oct. 2019.

[27] K. Zhao *et al.*, "Distribution adaptive INT8 quantization for training CNNs," 2021, *arXiv:2102.04782*.

[28] G. Yang, T. Zhang, P. Kirichenko, J. Bai, A. Wilson, and C. D. Sa, "SWALP: Stochastic weight averaging in low precision training," in *Proc. 36th Int. Conf. Mach. Learn., (ICML)*, vol. 97, 2019, pp. 7015–7024.

[29] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, "Evaluations on deep neural networks training using posit number system," *IEEE Trans. Comput.*, vol. 70, no. 2, pp. 174–187, Feb. 2021.

[30] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. 32nd Int. Conf. Mach. Learn. (ICML)*, 2015, pp. 448–456.

[31] F. Zhu *et al.*, "Towards unified INT8 training for convolutional neural network," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Oct. 2020, pp. 1966–1976.

[32] J. Hormigo and J. Villalba, "New formats for computing with real-numbers under round-to-nearest," *IEEE Trans. Comput.*, vol. 65, no. 7, pp. 2158–2168, Jul. 2016.

[33] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Oct. 2016, pp. 2818–2826.

[34] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.

[35] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. 32nd Int. Conf. Mach. Learn. (ICML)*, 2015, pp. 1737–1746.

[36] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," 2016, *arXiv:1603.01025*.

[37] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," 2016, *arXiv:1606.06160*.

[38] W. Zhao *et al.*, "F-CNN: An FPGA-based framework for training convolutional neural networks," in *Proc. IEEE 27th Int. Conf. Appl.-specific Syst., Archit. Processors (ASAP)*, Jul. 2016, pp. 107–114.

[39] Z. Liu, Y. Dou, J. Jiang, Q. Wang, and P. Chow, "An FPGA-based processor for training convolutional neural networks," in *Proc. Int. Conf. Field Program. Technol. (ICFPT)*, 2017, pp. 207–210.

[40] T. Geng *et al.*, "FPDeep: Acceleration and load balancing of CNN training on FPGA clusters," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2018, pp. 81–84.

[41] S. Dey *et al.*, "A highly parallel FPGA implementation of sparse neural network training," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2018, pp. 1–4.

[42] K. Guo *et al.*, "Compressed CNN training with FPGA-based accelerator," in *Proc. 2019 ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2019, p. 189.

[43] S. K. Venkataramanaiah *et al.*, "Automatic compiler based FPGA accelerator for CNN training," in *Proc. Int. Conf. Field Program. Log. Appl. (FPL)*, 2019, pp. 166–172.

[44] S. K. Venkataramanaiah *et al.*, "Fpga-based low-batch training accelerator for modern CNNs featuring high bandwidth memory," in *Proc. 39th Int. Conf. Comput.-Aided Design (ICCAD)*, 2020, pp. 1–8.

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

[46] S. Singh and S. Krishnan, "Filter response normalization layer: Eliminating batch dependence in the training of deep neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 11234–11243.

[47] M. Wang, S. Rasoulinezhad, P. H. W. Leong, and H. K. H. So, "NITI: Training integer neural networks using integer-only arithmetic," 2020, *arXiv:2009.13108*.

[48] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li, "Bag of tricks for image classification with convolutional neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Oct. 2019, pp. 558–567.

[49] Z. Yang, L. Wang, X. Zhang, D. Ding, C. Xie, and L. Luo, "PRBN: A pipelined implementation of RBN for CNN training," in *Proc. Conf. Adv. Comput. Archit.*, 2020, pp. 117–131.

[50] S. P. Fox, J. Faraone, D. Boland, K. Vissers, and P. H. W. Leong, "Training deep neural networks in low-precision with high accuracy using FPGAS," in *Proc. Int. Conf. Field-Program. Technol. (ICFPT)*, 2019, pp. 1–9.

[51] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, 2016, pp. 26–35.

**Jinming Lu** received the B.S. degree in microelectronics from Nankai University, Tianjin, China, in 2018. He is currently pursuing the Ph.D. degree in information and communication engineering with Nanjing University, Nanjing, China.

His current research interests include automatic speech recognition and deep learning, especially its hardware acceleration and compression algorithms.

**Chao Ni** received the B.S. degree in microelectronics from Jilin University, Changchun, China, in 2019. He is currently pursuing the M.S. degree with Nanjing University, Nanjing, China.

His current research interests include model compression algorithms and hardware acceleration for machine learning.

**Zhongfeng Wang** (Fellow, IEEE) received the B.S. and M.S. degrees from the Department of Automation, Tsinghua University, Beijing, China, in 1988 and 1990, respectively, and the Ph.D. degree from the University of Minnesota, Minneapolis, MN, USA, in 2000.

He worked at Oregon State University, Corvallis, OR, USA, and National Semiconductor Corporation, Santa Clara, CA, USA. He worked at Broadcom Corporation, San Jose, CA, USA, from 2007 to 2016, as a leading VLSI Architect. He has been working with Nanjing University, Nanjing, China, as a Distinguished Professor since 2016. He has published over 200 technical articles with multiple best paper awards received from the IEEE technical societies, among which is the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS Best Paper Award of 2007. He has edited one book *VLSI* and held more than 20 U.S. and China patents. His current research interests are in the area of optimized VLSI design for digital communications and deep learning.

Dr. Wang has served as a TPC member and various chairs for tens of international conferences. In 2015, he was elevated to the fellow of IEEE for contributions to VLSI design and implementation of forward error correction (FEC) coding. He is a world-recognized expert on low-power high-speed VLSI design for signal processing systems. In the current record, he has had many papers ranking among top 25 most (annually) downloaded manuscripts in IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS. In the past, he has served as an Associate Editor for IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—II: EXPRESS BRIEFS, and IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEM for many terms. Moreover, he has contributed significantly to industrial standards. So far, his technical proposals have been adopted by more than 15 international networking standards.