BOOST: Block Minifloat-Based On-Device CNN Training Accelerator with Transfer Learning

Chuliang Guo[1], Binglei Lou[2], Xueyuan Liu[2], David Boland[2], Philip H.W. Leong[2], Cheng Zhuo[1,3,*]
[1] Zhejiang University, Hangzhou, China
[2] The University of Sydney, Sydney, Australia
[3] Key Lab of CS&AUS of Zhejiang Province
*Email: czhuo@zju.edu.cn

***Abstract*—Adapting CNNs to changing problems is challenging on resource-limited edge devices due to intensive computations, high precision requirements, large storage needs, and high bandwidth. This paper presents BOOST, a novel block minifloat (BM)-based parallel CNN training accelerator on memory- and computation-constrained FPGAs for transfer learning (TL). By updating a small number of layers online, BOOST enables adaptation to changing problems. Our approach utilizes a unified 8-bit BM datatype (`bm(2,5)`), i.e., with a sign bit, 2 exponent bits, and 5 mantissa bits, and proposes unified Conv and dilated Conv blocks that support non-unit stride and enable task-level parallelism during back-propagation to minimize latency. For ResNet20 and VGG-like training on CIFAR-10 and SVHN datasets, BOOST achieves near 32-bit floating point accuracy, reducing latency by 21%-43% and BRAM usage by 63%-66% compared to back-propagation training without TL. Notably, BOOST outperforms the prior SOTA works to achieve per-batch throughput of 131 and 209 GOPs for ResNet20 and VGG-like respectively.**

I. Introduction

Convolutional neural networks (CNNs) are widely used in various deep learning scenarios at the edge, including object tracking [1], image classification [2], natural language processing [3], and autonomous driving [4]. However, most edge deployments focus on inference only, involving offline training of the CNN model on GPUs and fixed network parameters for execution. This approach lacks the adaptability required to handle changes in the environment or new tasks, necessitating re-training and re-deployment. Moreover, transferring network parameters to/from GPUs for edge-based machine learning incurs additional data transmission, memory bandwidth, and power consumption. The communication between edge devices and cloud GPUs also raises concerns about data privacy and latency. Consequently, there are significant challenges in deploying highly parallelized back-propagation algorithms on memory- and computation-constrained edge devices, limiting the scale of training workloads and performance at the edge. *This issue poses a significant obstacle to the adoption of machine learning at the edge under varying requirements or changing environments.*

Training at the edge faces strict limitations in terms of memory and computation resources. Consequently, only small neural networks with high precision or relatively large networks with low-bit data formats can be deployed. However, this compromises both task accuracy and latency performance. As a result, there is growing interest in *low-precision* and *low-batch* techniques for edge training, which minimize memory and computation requirements while enabling higher parallelism with limited resources. Recent studies have shown that even with low batch sizes as small as one and reduced bit-width (sub-8 bits) [5], [6], similar convergence and accuracy to training with floating-point precision and large batches can be achieved, even on large datasets like ImageNet. However, determining the optimal data format for CNN training at the edge remains a challenge, as it involves balancing accuracy and execution efficiency. Previous research has explored datatypes such as FP8, INT8, and the more recent block floating-point (BFP) [7]. However, ensuring adequate task accuracy with these techniques typically requires additional statistical calculations (e.g., standard variance and underflow rate) to determine scaling factors for each layer, posing a critical hurdle [8], [9]. In this context, block minifloat (BM) [10] has emerged as a promising solution, delivering near 32-bit floating-point accuracy with 8-bit computational complexity and memory bandwidth requirements for low-precision training at the edge.

To this end, block minifloat (BM) [10] delivers near 32-bit floating point accuracy with 8-bit computational complexity and memory bandwidth requirement, which becomes promising for low-precision training at the edge.

However, low-precision training using the back-propagation algorithm [11] remains relatively slow, as it requires three times more multiply-and-add (MAC) operations than inference, resulting in three times longer latency for training. This phenomenon is due to (1) The workload of low-precision training remains the same amount of MACs as full-precision floating-point training; and (2) Most prior works adopted the same architecture (such as systolic arrays) for forward Conv, backward transposed Conv, and dilated Conv, despite their different computation patterns, resulting in low efficiency and hence long latency in the generic architecture for gradient generation.

To address these obstacles, **we propose BOOST, an efficient accelerator that utilizes block minifloat (BM)-based transfer learning (TL) to accelerate CNN training on FPGAs.** TL reduces the training workload by fine-tuning only the last several layers of a pre-trained model, rather than starting from scratch, to achieve higher accuracy with fewer training epochs. In particular, BOOST *explores channel parallelism and adopts unified `bm(2,5)` precision* for end-to-end stochastic gradient descent (SGD)-based back-propagation CNN training, featuring task-parallelism of error back-propagation and gradient generation. This novel low-precision arithmetic eliminates the need for extra computations for layer-wise scaling factors and implements simplified computation units with fewer on-FPGA resources compared to FP32/FP16. Additionally, *low-batch training combined with TL* allows relaxed memory requirements for on-chip storage of input feature maps, eliminating time- and energy-consuming external data access for forward activations.

In addition to the above, we design *a unified Conv block that handles both forward Conv and backward transposed Conv, along with a dilated Conv block* for gradient generation and weight update. These Conv blocks enable task-level parallelism and significantly reduce the back-propagation latency. Notably, our Conv blocks support non-unit stride, a feature rarely discussed in prior works [12], [13]. In short, the major contributions of this work are summarised as follows:

- To the best of our knowledge, this is the first FPGA-based CNN training accelerator for TL using 8-bit BM arithmetic. BOOST addresses the memory and computation demands of FP32 while

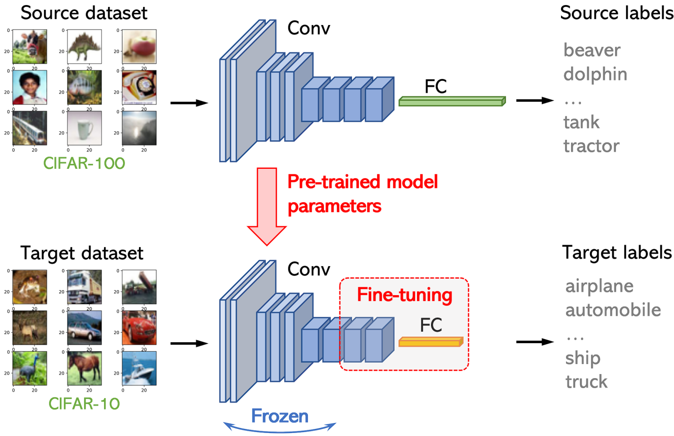


Fig. 1: Inductive TL of CNN for image classification. i. training from scratch on the source dataset (e.g., CIFAR-100); ii. initialization of the same CNN with pre-trained parameters; iii. fine-tuning FC (including several final Conv layers) as a new linear classifier for the target dataset (e.g., CIFAR-10).

achieving similar accuracy at much lower latency than training from scratch.

- We propose a unified BM precision for CNN training, `bm(2,5)`, with modified CNN building blocks, which eliminates the necessity of using different precision for the forward and backward paths. This narrower bit-width allows for the efficient reuse of computation units and saves logic resources on the FPGA.
- We design a novel unified Conv block for both forward Conv and backward transposed Conv, as well as a dilated Conv block with a weight kernel partition scheme for gradient generation. These blocks support non-unit stride and enable task-level parallelism, minimizing back-propagation latency. [1]

We demonstrate the performance and benefits of the proposed BOOST accelerator on VGG-like [14] and ResNet20 [15] networks, using unified `bm(2,5)` precision for end-to-end back-propagation training. Experimental results show that BOOST achieves near 32-bit floating point accuracy with a latency reduction of 21%-43% and BRAM reduction of 63%-66% compared to back-propagation training without TL. Furthermore, BOOST outperforms prior state-of-the-art works with per-batch throughput of 131 and 209 GOPs for ResNet20 and VGG-like respectively [12], [13], [16]–[20].

II. Preliminaries and Related Works

In this section, we introduce the preliminaries of the proposed accelerator, including the TL training workflow and the arithmetic operations for SGD-based back-propagation training of CNNs. We then review the BM number system and related works.

A. CNN Transfer Learning

With features learned from the source dataset, TL generally updates parameters in the latter layers of the CNN on the target dataset, as shown in Fig. 1. The pre-trained CNN is fine-tuned using the back-propagation algorithm [11] with the SGD optimizer [21], as illustrated in Algorithm 1, including four stages: (1) forward path, (2) backward path, (3) gradient generation, and (4) weight update.

[1]Reference designs have been made available from `https://github.com/chuliang007/resnet20_training`.

Algorithm 1: Conv patterns in SGD training with momentum, including Conv, transposed Conv, and dilated Conv.

1 **Input:** image samples, X; categorical labels, y;
2 **Variables:** activation, A; weight, W; error, E; gradient, G; velocity, V;
3 **Parameters:** stride, s; padding, p; momentum, α; learning rate, η;
4 **for** $n = 1$ to N **do**
5 Choose k uniformly at random from $\{1, 2, 3, ..., N\}$
$A^1 = X^k$;
`/* Forward- Conv */`
6 **for** $l = 2$ to L **do**
7 $A^l[c_o][h][w] = \sum_{h,w,c_i,c_o,h_k,w_k=0}^{H_{out},W_{out},C_{in},C_{out},H_{wt},W_{wt}}$
8 $W^l[c_o][c_i][h_k][w_k] \times A^{l-1}[c_i][h \times s + h_k - p][w \times s + w_k - p]$;
9 **end**
10 Calculate error E^L using A^L and label $y\{X^k\}$
`/* Backward- Transposed Conv */`
11 **for** $l = L - 1$ to M **do**
12 $E_{dil}^{l+1}[c_i][h \times s][w \times s] = E^{l+1}[c_i][h][w]$;
13 $E^l[c_o][h][w] = \sum_{h,w,c_i,c_o,h_k,w_k=0}^{H_{out},W_{out},C_{in},C_{out},H_{wt},W_{wt}}$
14 $W_{rot}^{l+1}[c_o][c_i][h_k][w_k] \times E_{dil}^{l+1}[c_i][h + h_k - p][w + w_k - p]$;
15 **end**
`/* Gradient- Dilated Conv */`
16 **for** $l = L$ to M **do**
17 $E_{dil}^l[c_o][h \times s][w \times s] = E^l[c_o][h][w]$;
18 $G^l[c_o][c_i][h][w] = \sum_{h,w,c_i,c_o,h_k,w_k=0}^{H_{out},W_{out},C_{in},C_{out},H_{wt},W_{wt}}$
19 $E_{dil}^l[c_o][h_k][w_k] \times A^{l-1}[c_i][h + h_k - p][w + w_k - p]$;
`/* Weight update */`
20 $V^l = \alpha V^l + \eta G^l$;
21 $W^l = W^l - V^l$;
22 **end**
23 **end**

Forward path: In the forward path, input activations pass through each layer to produce the numeric output. Convolutional layers slide weight kernels over input feature maps to generate output feature maps. Batch normalization (BN) performs an affine transformation on the output feature maps, while ReLU replaces negative values with zeros. AvgPool reduces the feature map size through average pooling. The fully-connected (FC) layer applies a linear transformation between input activations and weights to produce outputs matching the categorical labels.

Backward path: The backward path for unit-stride Conv is the same as the forward path, except that weights are rotated by 180° in the height and width dimensions, while filter and channel dimensions are exchanged [22]. For non-unit stride, the transposed Conv acts as an up-sampling layer, retaining the same connectivity pattern between weights and input activations as the forward Conv. Normalized input activations are reused with respect to the cost function to calculate weight gradients, bias gradients, and back-propagated errors in BN layers. ReLU applies the same 0/1 mask as in the forward path, and AvgPool replaces each pixel with multiple pixels of the same averages. FC maintains the same computational pattern, except for transposed linear weights. In TL scenarios, the backward path can be shortened by back-propagating errors amongst several of the final layers, i.e., from L to M layers, where $M \geq 2$.

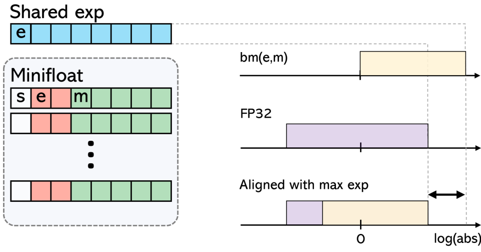


Fig. 2: Minifloat and BM representations. The shared exponent is common for minifloats of the same tile group and aligning the maximum minifloat with the maximum floating point.

Gradient generation: Gradients in the back-propagation algorithm are calculated using dilated Conv between forward input activations and back-propagated errors. This approach avoids repeatedly deriving gradients with the chain rule for each layer. In the case of non-unit stride, errors are dilated with zeros between adjacent pixels and expanded to match the receptive field size of the input activations in the previous layer. Dilated Conv operates independently on feature maps from different channels.

Weight update: Network parameters, such as weights and biases, can be immediately updated once gradients have been generated. The SGD optimizer with momentum is used, considering both current and past gradients to determine the next direction of gradient descent. Momentum velocities accumulated by weight gradients per iteration are stored as well.

B. BM Number System

The BM number system, denoted as bm(e,m) [10], is a novel floating-point representation designed for low-precision training, as depicted in Fig. 2. It employs an 8-bit shared exponent for each tile group consisting of 48×48 minifloats (with a tile size of 48). The shared exponent serves as a tile-wise scaling factor, ensuring that the maximum 8-bit minifloat aligns with the maximum floating point value. This approach avoids overflow issues and provides finer-grained scaling compared to traditional layer-wise loss/gradient scaling methods [8], [9].

C. Related Work

1) Low-precision training: Prior works in low-precision training have explored INT8 [17]–[19] or INT16 [23] formats, which often suffer from accuracy degradation due to the limited dynamic range of gradients in integer training [24]. Alternative approaches, such as the Efficient Training Accelerator (ETA), introduced the PINT8 format, achieving close-to-baseline training accuracy [13]. Block Floating-Point (BFP) training has also gained attention for achieving a balance between INT-like performance and floating-point accuracy. Hybrid BFP-FP number representations, such as HBFP, have been proposed for DNN training with FPGA implementations [7]. The Fast First, Accuracy Second Training (FAST) system [25] supports variable precision BFP with bit-parallel implementations, achieving similar accuracy to HFP8 [26] on CNN and transformer models at faster speeds. FlexBlock [27] supports various floating point precisions, including 4-bit, 8-bit, and 16-bit mantissa with an 8-bit exponent, utilizing hierarchical sub-word parallelism. Bitlet [28] employs bit-level sparsity parallelism and bit-interleaved processing elements but lacks acceleration for back-propagation in training. *While Block Minifloat (BM) incurs higher complexity than fixed-point logic, it has demonstrated the best-reported accuracy for sub-8-bit training.*

2) FPGA-based training accelerators: Existing FPGA-based CNN training accelerators often prioritize throughput improvements and rely on large batches (e.g., from 16 to 128). However, such designs consume substantial resources and lack flexibility, particularly for deep networks. This approach is not suitable for training on edge devices with limited memory and where latency is critical. Most existing CIFAR-10 training techniques only support older CNNs with unit stride, and their Conv blocks are unable to handle non-unit stride in the forward path or transposed Conv in the backward path. Additionally, many works neglect the inclusion of normalization layers, such as batch normalization (BN), due to the computation complexity and latency concerns. However, BN is known to be crucial for achieving high CNN accuracy [29]. Some approaches, such as ETA [13] and FlexBlock [27], incorporate L1-norm filter response normalization (L1-FRN) layers or RangeBN [30] to address these challenges and improve accuracy.

Moreover, prior FPGA-based training accelerators have not explored the use of TL for CNNs, which can offer additional benefits in terms of latency and memory usage. While FPGA-based inference accelerators have been proposed [31], which utilize pre-trained weights and efficient Conv blocks from top layers, they still rely on complete back-propagation and do not effectively reduce latency.

III. Algorithmic Optimizations for CNN Training

In this section, we present algorithmic optimizations for CNN training to increase hardware efficiency on FPGAs.

A. Unified `bm(2,5)` Precision for Forward/Backward Operations

The CNN task accuracy is strongly dependent on the numerical precision and dynamic range during training. Given a certain floating-point bit-width, such as 8 bits, there exists a bit-width balance between the mantissa (numerical precision) and the exponent (dynamic range). Hybrid precision is favored in GPU training due to the different preferences for dynamic range and numerical precision between the forward and backward paths [26]. However, on an FPGA, accommodating a multiplier/adder/MAC capable of handling all these data formats or separate ones would result in idle components for a significant proportion of the time. To avoid this, we adopt a unified Block Minifloat (BM) precision for on-FPGA training.

From a hardware perspective, fewer exponent bits are preferred due to a narrower bit shifter and reduced critical path delay for accumulators and multipliers. Specifically, the required bit-width of the Kulisch accumulator, which converts floating points into full-length fixed points and performs error-free addition for dot products, grows exponentially with the exponent bits of minifloats. The limited dynamic range often leads to significant accuracy degradation, especially in the backward path and gradient generation, where errors and gradients prioritize dynamic range over precision, as reported in [26]. However, as shown in the visualization presented in [32], the dynamic range of errors within each layer can be adequately captured by a 2-bit exponent, even if their absolute magnitudes are very small (e.g., smaller than 2^{-10} in ResNet18 training on ImageNet).

Using BM arithmetic, we can effectively capture the dynamic range within a tile group (typically smaller than a layer) by utilizing a small minifloat exponent, such as 2 bits, and scaling it to tiny absolute magnitudes with the assistance of a shared exponent of sufficient bit-width. This approach eliminates the need for a 40-bit accumulator in the backward path, as required in the original hybrid `bm(2,5)`/`bm(4,3)` precision [10], resulting in saved computation resources and reduced critical latency [10]. In our practical implementation, we allocate 8 bits for shared exponents and choose

TABLE I: Top-1 accuracy on CIFAR-10 and SVHN training from scratch.

Model	Precision (FP/BP)	CIFAR-10 Acc.	SVHN Acc.
VGG-like	FP32	86.46%	92.60%
	BFP8	85.63%	91.77%
	bm(2,5)/bm(4,3)	86.39%	92.05%
	bm(2,5)	**86.43%**	**92.12%**
ResNet20	FP32	90.04%	93.34%
	BFP8	87.56%	90.17%
	bm(2,5)/bm(4,3)	89.38%	92.13%
	bm(2,5)	**89.71%**	**92.57%**

a tile size of 32 to ensure adequate dynamic range. Experimental results on CIFAR-10 and SVHN datasets, as presented in Table I, demonstrate that modified ResNet20 and VGG-like networks utilizing the unified bm(2,5) precision achieve comparable accuracy to FP32 and outperform hybrid BM and BFP8 [7] precision methods.

B. TL-Oriented Improvements to CNN Building Blocks

Layer fusion [33] is commonly used to combine Batch Normalization (BN) with the previous Convolutional (Conv) layer to reduce latency during inference. However, in backward propagation, BN requires channel-wise gradients for the BN weights and bias, which can only be computed after the mini-batch Conv operation is completed. This makes fusing backward BN into Conv layers challenging. To address this, we perform layer fusion by moving the second Rectified Linear Unit (ReLU) function in the ResNet basic building block before the shortcut addition. This fuses ReLU and BN, enabling each basic block to be processed with just Conv and the fused BN&ReLU functions. In the VGG-like network, we replace the max-pooling layers with stride-2 Conv layers to maximize computation block reuse. This ensures efficient hardware utilization.

However, due to the limited feature representations of the modified ResNet20 and VGG-like networks, their performance on the ImageNet dataset during a 90-epoch training is subpar, achieving a top-1 accuracy of 40.23% and 28.10%, respectively. Therefore, for the purpose of proof-of-concept, we pre-train the ResNet20 and VGG-like networks on CIFAR-100 using unified bm(2,5) precision and then transfer the network parameters to the CIFAR-10 and SVHN datasets as a starting point for fine-tuning. Fully-connected (FC) layer is randomly initialized and trained from scratch due to the different number of classes in CIFAR-100. As presented in Table II, fine-tuning the pre-trained CNN models on CIFAR-100 achieves higher accuracy compared to training from scratch with randomly initialized network parameters. This illustrates an opportunity for the trade-off between accuracy and hardware efficiency, i.e., only activations in the final several trainable layers are kept for back-propagation. As our TL strategy, we update the last 6 and 2 Conv layers of ResNet20 (batch size 4) for SVHN and CIFAR-10 respectively, and the last 3 Conv layers of VGG-like (batch size 8) for both datasets.

TABLE II: Top-1 accuracy of TL on CIFAR-10 and SVHN from CIFAR-100. Unified bm(2,5) precision is utilized.

Model	Layer Update	CIFAR-10 Acc.	SVHN Acc.
VGG-like	Full update	87.12%	92.75%
	4 Conv + FC	86.57%	91.29%
	3 Conv + FC	**86.62%**	**91.31%**
	2 Conv + FC	83.76%	84.62%
ResNet20	Full update	90.89%	94.16%
	6 Conv + FC	90.37%	**92.26%**
	4 Conv + FC	89.39%	91.23%
	2 Conv + FC	**88.97%**	90.34%

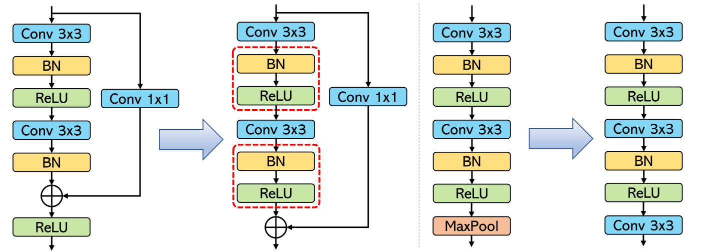


Fig. 3: Modifications to the basic building block of ResNet20 and VGG-like.

IV. Proposed Accelerator

With the proposed unified bm(2,5) precision and improvements on the CNN building blocks, this section presents the details of the proposed accelerator BOOST for CNN training, implemented in high-level synthesis (HLS) using a coding style inspired by FraBNN [34].

A. Overall Architecture

The BOOST training accelerator can support the most commonly used CNN layers. For example, the modified ResNet20 accelerator requires the following operation types: 3×3 Conv and transposed Conv with stride 1 and 2, 1×1 Conv and transposed Conv with stride 2, 3×3 dilated Conv with stride 1 and 2, 1×1 dilated Conv with stride 2, 8×8 AvgPool, FC, BN&ReLU, and shortcut addition.

Fig. 4 illustrates the execution of basic building blocks for the forward path, backward path, gradient generation, and weight update. To maximize hardware utilization, we generate at most one instance of each computational block and process the CNN sequentially, layer by layer. In both forward and backward paths, the layer-by-layer processing is decoupled into the sequential execution of Conv, BN_ReLU, and shortcut addition in basic building blocks, as shown in Fig. 4 (a) and (b). Since BN layers and shortcut addition cannot be fused into adjacent Conv layers, their sequential processing contributes to a significant portion of the overall latency.

To address the abovementioned performance degradation, in contrast to the prior works that process error back-propagation and weight gradient generation sequentially, we employ task-level parallelism by processing transpose Conv and dilated Conv for the same Conv layer in parallel (Fig. 4 (b)). This approach reduces the latency of weight update to approximately $2\times$ the latency for inference, this being a significant improvement over references [12], [13] which require $3\times$.

Processing elements (PEs) implement a sliding window Conv, as shown in Fig. 4 (c), where BM arithmetic is applied in the dot products. Gradual underflow with denormals and Kulisch accumulation are crucial to achieving high accuracy in BM training [10]. Here we use unified bm(2,5) precision for activations, weights, errors, gradients, and momentum velocities during training, which supports denormals, normals, and zero as depicted in Eq. (1):

$$X\langle 2,5\rangle = \begin{cases} (-1)^s \times 2^{1-b} \times \left(0 + m \times 2^{-5}\right), & (e = 0) \\ (-1)^s \times 2^{e-b} \times \left(1 + m \times 2^{-5}\right), & (e \neq 0) \end{cases} \quad (1)$$

where the minifloat X is represented by sign s, exponent e, mantissa m, and exponent bias $b = 2^{2-1} - 1$. 8-bit minifloats in bm(2,5) format are converted to 20-bit fixed points for error-free Kulisch accumulation in dot products.[2] Updated weights and velocities are

[2]To save DSP resources, multipliers and adders are synthesized with LUTs while only MAC units utilize DSP48E2s.

transferred back to off-chip DDR after data conversion from fixed point to BM, where the leading-zero counter is implemented by `hls::log2()` inherently supported in the HLS math library, and stochastic rounding is implemented by a linear feedback shift register.

Algorithm 2: Channel tiling for ResNet20 training with BM

```
1  Variables: activation, A; weight, W; error, E; gradient, G;
     velocity, V;
2  Parameters:
3  #input/output channel, in_channels, out_channels;
4  input/output CPF, ch_in_t = 8, ch_out_t = 8;
5  ...
   /* layer_1_0 Conv2 forward                    */
6  in_channels = 16;
7  out_channels = 16;
8  for c_out = 0 to out_channels/ch_out_t − 1 do
9      activation index update;
10     for c_in = 0 to in_channels/ch_in_t − 1 do
11         weight index update;
12         function load_weight (W);
13         function Conv (A, W);
14         shared exponent update;
15     end
16     function BN&ReLU (A);
17 end
18 function shortcut_addition (A, A);
19 ...
   /* layer_1_0 Conv2 backward                   */
20 in_channels = 16;
21 out_channels = 16;
22 for in_channels/ch_in_t − 1 to c_in = 0 do
23     activation index update;
24     function BN&ReLU (A, E);
25     for out_channels/ch_out_t − 1 to c_out = 0 do
26         weight index update;
27         function load_weight_momentum (W, V);
28         function Conv (E, W) and Conv_grad (A, E);
29         function write_weight_momentum (W, V);
30         shared exponent update;
31     end
32 end
33 ...
```

In a high-level construction of the accelerator, channel parallelism is adopted for input and output channels, with a channel parallelism factor (CPF) of 8 (as shown in Algorithm 2). This allows the parallel processing of feature maps from 8 input and output channels. Thus, it iterates 4, 16, and 64 times for Conv layer stacks with 16, 32, and 64 channels, respectively. The activation, weight, and 1-bit ReLU mask indexes are updated when switching channels and layers. The shared exponent of the BM is separately stored and aligns the 8-bit minifloat with the maximum number. By employing channel tiling, the maximum number along feature map dimensions is determined in pipelined function blocks, and the shared exponent is updated accordingly. The tile size is chosen to accommodate the maximum feature map size of the datasets (e.g., 32 for CIFAR-10 and SVHN), enabling the division of the minifloat group along the feature map dimension (i.e., width and height) of the activations in channel tiling. We store the forward activations of the final layers on-chip while reading/writing weights and momentum velocities from/to off-chip DDR, which provides several benefits:

- It allows for larger-scale CNNs supported within the FPGA's memory resource budget by keeping the weights off-chip.
- On-chip activations eliminate the time-consuming DDR transfer of variable-sized feature maps, which is typically slower than fetching and unpacking weights in previous works [12], [13].
- On-chip activations contribute to higher throughput as batch size increases, thanks to weight reuse across batches. This effect becomes more pronounced as the proportion of external data access latency grows.

B. Unified Conv Block

The unified Conv block is a critical component responsible for performing forward Conv and backward transposed Conv operations with support for non-unit stride (e.g., stride 1 and 2 for ResNet20 and VGG-like). Its design aims to optimize memory bandwidth and maximize hardware efficiency.

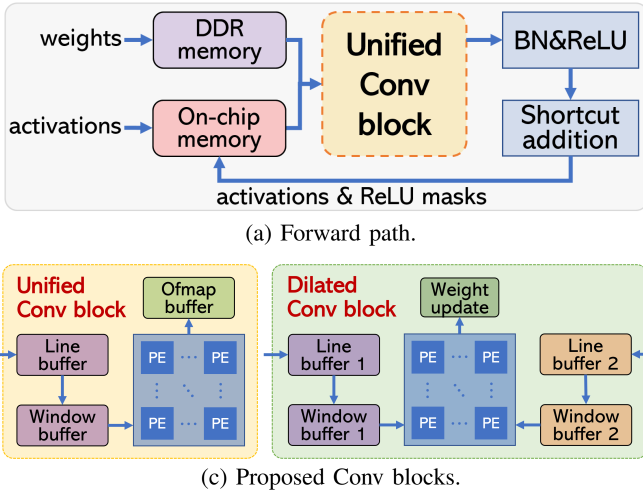
To achieve efficient memory access, the block utilizes a combination of local buffering techniques, namely the line buffer and window buffer. The line buffer stores lines from the input feature map and updates pixels vertically. It has the same number of rows as the input feature map and a number of columns equal to the height of the weight kernel. This configuration allows the sliding window of the weight kernel to efficiently compute with the locally stored pixels of the input feature map horizontally. In addition, channel-level parallelism is achieved by employing an array of line buffers, where each line buffer corresponds to an input feature map. The window buffer follows the shifting pattern of the weight kernel over the image, and it copies the necessary pixels from the line buffer for processing by the kernel. By utilizing multiple window buffers, corresponding to different input channels, and applying parallelism across them, the block can generate multiple output feature maps simultaneously. For example, for the CPF of 8, 8 window buffers can sample 8 line buffers that correspond to 8 input channels, and 8 kernels can be applied to these window buffers in parallel; this results in 8 output feature maps per clock cycle.

The structure of the line buffer and window buffer architecture is depicted in Fig. 5. It demonstrates how the line buffer, with the same number of columns as the input feature map, slides down along the column, while the window buffer slides right along the row of the line buffer. This coordinated movement allows efficient computation of Conv and transposed Conv operations with minimal memory bandwidth requirements.

In terms of specific Conv operations, the block supports both unit-stride Conv and transposed Conv. They share the same computational pattern, with the only difference being the 180°rotation of weights and exchange between the filter and channel dimensions for transposed Conv. For stride-2 Conv in the forward path, where the output feature map size is halved compared to the input, a strategy is employed to discard output pixels every two steps, taking into account the limited capacity of the line buffer. Similarly, for stride-2 transposed Conv in the backward path, zeros are inserted between adjacent pixels during the buffering process to ensure consistent processing of Conv and transposed Conv operations with arbitrary strides.

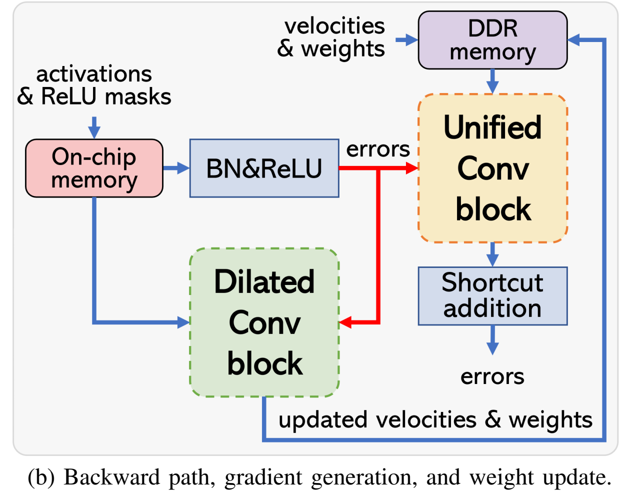
C. Dilated Conv Block

In Conv layers, errors are computed using error and rotated weights, while gradients are generated using input activations and errors. We apply task-level parallelism for error back-propagation and weight gradient generation to ensure similar latency for the unified Conv block and the dilated Conv block. However, the error, serving



(a) Forward path.

(c) Proposed Conv blocks.



(b) Backward path, gradient generation, and weight update.

Fig. 4: Overall architecture of the CNN training accelerator BOOST for layer-by-layer processing in ResNet training. (a) The proposed unified Conv block and dilated Conv block utilize line buffer and window buffer for activations (and errors). (b) In the forward path, Conv, BN_ReLU, and shortcut addition are sequentially processed for each layer with the same block instances. (c) During the backward path, transposed Conv and dilated Conv are deployed in task-level parallelism to hide the latency of gradient generation during error back-propagation.

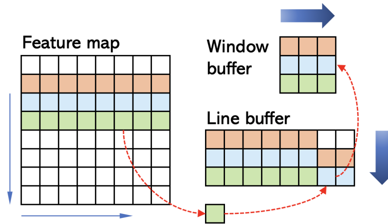


Fig. 5: Line buffer and window buffer architecture in unified Conv block. The line buffer with the same column as the input feature map slides down along the column of the feature map, and the window buffer with the same row as the line buffer slides right along the row of the line buffer.

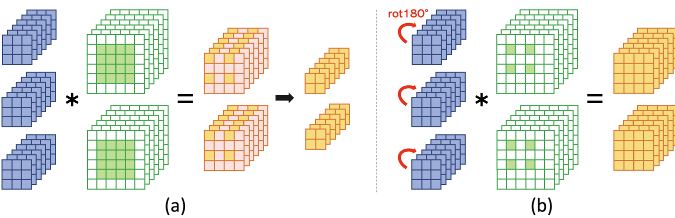


Fig. 6: Stride-2 Conv and transposed Conv. We perform stride-2 Conv in unit-stride and discard output pixels every two steps in the forward path, and dilate input feature maps with zeros when loaded into the line buffer in the backward path.

as the weight, has the same size as the input, which is larger than the size in the unified Conv block. This prevents the practical use of local buffering for the input feature map since the window buffer and line buffer would need to be the same size as the complete feature map, such as from 8×8 to 32×32 for CIFAR-10 and SVHN datasets.

To overcome this limitation, we propose a weight kernel partition scheme for dilated Conv. As illustrated in Fig. 7, we partition the weight kernel, such as the 8×8-sized weight kernel in the last Conv layer of ResNet20, into multiple 4×4 regions. These regions can be arranged in the window buffer of a similar size as the unified Conv

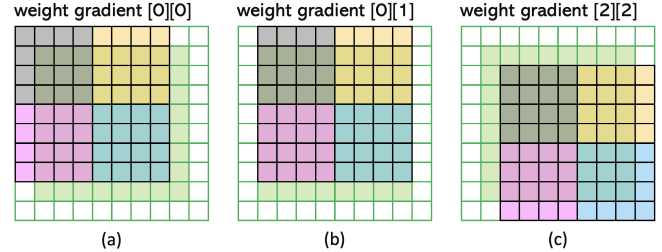


Fig. 7: Weight kernel partition in dilated Conv. The error as weight kernel is partitioned into four 4×4 small kernels for dilated Conv.

block. The size of the partitioning region can be adjusted, but a small size would require frequent data communication and under-utilization of the MAC units, slowing down gradient generation. On the other hand, a large size would require a significant number of BRAM ports, making it challenging to achieve a non-trivial initiation interval.

In this approach, we allocate separate groups of line buffers and window buffers for errors and input activations, respectively (Fig. 4 (c)). For each computation, 4×4 partitioned weights are loaded. The decision to use partial sums as part of the weight gradient pixel is determined by the pixel indexing of feature maps, which naturally occurs in sliding windows of dilated Conv. To avoid duplicate summations, partial sums are added to the same weight gradient every four Conv steps. For example, the partial sum at step 0 (grey area in Fig. 7(a)) contributes to the pixel of weight gradient [0][0], while the partial sum at step 1 (grey area in Fig. 7(b)) contributes to the pixel of weight gradient [0][1]. After four steps, the partitioned weight kernel contributes again to the pixel of weight gradient [0][0] (yellow area in Fig. 7(a)). Weight gradients in local buffers can be released after the velocity accumulation during the weight update phase, which is inlined in pipeline processing following gradient generation.

Although dilated Conv typically exhibits batch-level parallelism instead of channel-level parallelism in Conv and transposed Conv, the dilated Conv block for gradient generation requires nearly the same latency as the unified Conv block when the feature map size is the same. This is due to the channel-pipelined processing of

local buffering and unit-stride Conv. Therefore, neither of the Conv blocks degrades the latency of the task-level parallelism between the transposed Conv and dilated Conv.

V. Experimental Results

In this section, we first describe the experimental setup and then present our TL results on FPGAs.

A. Experimental Setup

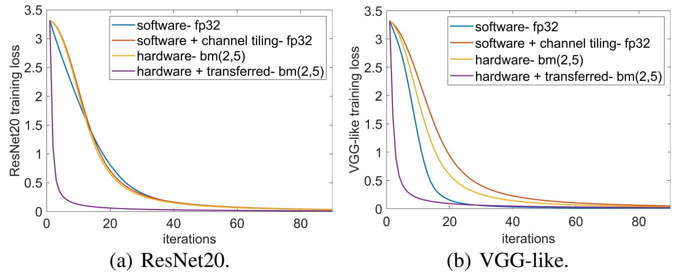
We implement the training accelerator BOOST in HLS, synthesize, implement, place and route using Vivado design suite 2019.2, and evaluate the performance on a Xilinx ZCU102 FPGA. This board uses the Zynq UltraScale+ MPSoC, which contains an embedded ARM CPU. The programmable logic fabric has 274K LUTs, 2520 DSPs, and 64 Mb BRAMs. We have implemented accelerators with all parameters updated, and accelerators for TL using different layer update strategies as demonstrated in section III-B with forward activations kept on-chip. ResNet20 and VGG-like networks are first pre-trained on CIFAR-100 and then fine-tuned on CIFAR-10 and SVHN respectively with a re-initialized 10-class FC layer. Weights include off-chip 3×3 and 1×1 kernels in Conv layers, on-chip scalar weights in BN layers, and 2-dimensional linear weights in the FC layer. Biases in BN and FC layers are also updated on-chip. Inputs are 32×32 RGB images loaded from off-chip DDR4 through an AXI-lite bus under the control logic executed by the ARM-based processor. Activations/errors and gradients are stored on-chip for forward/backward operations and weight update, and discarded from double buffers after processing of the current layer. The remaining training workloads are executed on the FPGA logic part, running at a system frequency of 225MHz.

Note that our BOOST implementation is also capable of complete back-propagation training since we support all arithmetic operators in ResNet20 and VGG-like. It is in principle scalable to different CNNs and datasets, but in this paper we implement CIFAR-10 and ResNet20 for comparison with the literature.

B. FPGA Implementation

To validate the functionality of the ResNet20 and VGG-like training accelerators, we demonstrate the training loss curve of CIFAR-10 with streaming images. The learning rate is fixed as 0.005 and 0.02 for ResNet20 and VGG-like respectively, both with a momentum of 0.9. As shown in Fig. 8, the *'software'* curve sequentially processes all layers with generic training algorithms (e.g., generic Conv, BN, and FC) regardless of the channel number, while the *'channel tiling'* curve additionally takes the same processing order as the hardware design shown in Algorithm 2, where BN is operated inside output channel tiling but outside input channel tiling. This difference makes the convergence speed of these two curves not exactly the same. The *'hardware'* curve utilizes the proposed unified Conv and dilated Conv blocks as well as channel tiling. As for the *'transferred'* curve, we first train the networks with CIFAR-100 images, re-initialize the FC layer, freeze Conv layers except for the last two of ResNet20 and the last three of VGG-like, and then fine-tune the networks with the CIFAR-10 images for another 90 iterations.

The CPF parameter is set at a maximum of 8 due to HLS tool limitations on pipeline regions although it is in principle right to increase above 8. The Conv block has a peak throughput of around 250 GOPs and requires memory bandwidth higher than 38.4 Gb/s from the ZCU102 for off-chip weight access. Therefore, the design is memory-bound and takes additional DDR transfer time. Since 3×3 and 1×1 Conv weights would not be read/written from/to DDR



(a) ResNet20. (b) VGG-like.

Fig. 8: Training loss of the accelerators using CIFAR-10 images.

TABLE III: Resource utilization and power consumption of ResNet20 and VGG-like TL accelerators.

	LUT	DSP	BRAM	Power (W)
Full ResNet20	189K	502	1735	8.7
6 Conv+FC	142K	468	588	7.3
2 Conv+FC	131K	381	650	7.2
Full VGG-like	147K	373	1255	7.7
3 Conv+FC	130K	372	477	6.7

simultaneously in layer-by-layer processing, they are packed in 64 bits and sent through the same bundle and bus. Momentum velocities utilize different bundles and buses from weights during weight update phase. Writing updated weights and velocities takes less than twice the time as reading weights due to the much higher frequency of the DDR clock compared to the system clock.

The post-implementation reports of resource utilization and power consumption are summarised in Table III. The BRAM usage for storing forward activations greatly reduced in TL, i.e., 66% and 63% for ResNet20 (CIFAR-10 accelerator utilizing more LUTRAM and thus fewer BRAM than SVHN) and 62% for VGG-like. The FPGA board power, including the FPGA, fans, and all other system power, was estimated using the Xilinx Vivado Tool and measured with an EcoFlow RIVER Max Portable Power Station.

Fig. 9 and 10 demonstrate the latency breakdown of BOOST for batch 1 operation, including the forward path, backward path, gradient generation, and total iteration time through the Vivado HLS co-simulation reports and real hardware tests on the Zynq PL-PS device. The latency in the forward path includes reading in Conv weights and input images from off-chip DDR4 memory, sequential processing of Conv layer stacks, AvgPool, FC, and calculating the cross-entropy loss. The computation in the backward path (including gradient generation and weight update) exhibits similar latency in Conv layers due to task-level parallelism except for doubled latency in BN layers because their gradients are first calculated before using them for error back-propagation. The total latency also accounts for Zynq system operations, such as communications with the ARM processor through the AXI-lite bus. If without task-level parallelism, this would result in a total throughput much lower than the peak throughput of 250 GOPs during back-propagation.

In TL scenarios with all network parameters updated, training on the source dataset simply serves to initialize the weights. Using a source dataset of CIFAR-100 and a target dataset of CIFAR-10, we observed an accuracy improvement of 1.18% (0.69%) for ResNet20 (VGG-like) networks compared with training from scratch. When updating several of the final Conv and FC layers during TL, an accuracy degradation of 0.74% for ResNet20 was observed. Similar results are achieved for the SVHN target dataset. Freezing most of the network parameters during back-propagation in TL brings benefits to

TABLE IV: Performance comparison of FPGA-based CNN training accelerators.

	[16]	[17]	[18]	[12]		[20]		[19]	[13]			**Ours**	
Device	ZU19EG	KCU1500	ZCU111	Stratix 10 MX		ZCU102		MAX5	VC709			ZCU102	
Data Format	FP32	INT8	INT8	FP16		FP32		INT8	PINT8			`bm(2,5)`	
Freq. (MHz)	200	250	180	185		100		200	200			225	
Dataset	CIFAR-10	CIFAR-10	CIFAR-10	CIFAR-10		ImageNet		CIFAR-10	CIFAR-10		ImageNet	CIFAR-10 & SVHN	
CNN Network	LeNet10	VGG-like	VGG16	VGG-like	ResNet20	VGG16		VGG-like	VGG16	ResNet20	ResNet18	VGG-like	ResNet20
DSP	1500(76%)	1030(19%)	1037(25%)	1046(26%)	1040(26%)	1508(60%)	1680(67%)	6241(91%)	1728(48%)			373(15%)	502(20%)
LUT/ALM	33K(63%)	199K(30%)	73K(17%)	221K(31%)	239K(34%)	-	-	679K(57%)	132K(30%)			147K(54%)	189K(69%)
BRAM/M20K	174(18%)	1060(49%)	1045(97%)	2998(44%)	2558(37%)	787(86%)	812(89%)	1232(29%)	240(14%)			1255(69%)	1735(95.12%)
Norm	-	-	-	-	-	-	BN	-	L1-FRN	L1-FRN	L1-FRN	BN	BN
Batch Parallel	-	-	1	1	1	1	1	**128***	**16***	**16***	**16***	1	1
Tput. (GOPs)	86	641	20	160	180	47	40	1417	611	659	811	209	131
Power (W)	14.2	26.8	-	20	20	7.71	8.2	13.5	8.4	8.6	8.6	7.7	8.7
Eff. (GOPs/W)	6.1	23.9	-	8	9	6.1	4.9	105.0	72.7	76.6	94.3	27.1	15.1

* Throughput of parallel multiple mini-batches.

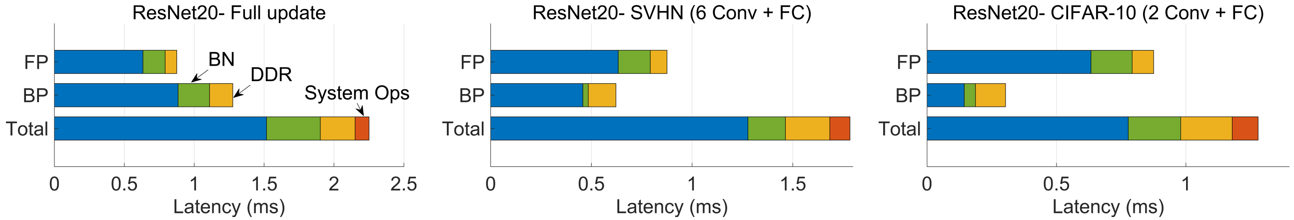


Fig. 9: Latency breakdown of ResNet20 training accelerators running at 225 MHz.

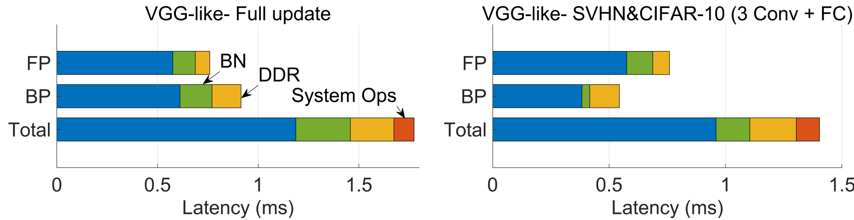


Fig. 10: Latency breakdown of VGG-like training accelerators running at 225 MHz.

the latency of error back-propagation and weight gradient generation. With the ResNet20 architecture, for SVHN, by updating only the last 6 Conv layers and the FC layer, we can halve the back-propagation time; for CIFAR-10, updating only the last 2 Conv layers and FC layer takes less than a quarter of the full back-propagation time. This translates to reducing the total overall latency by 30% and 43% respectively. Similarly, for the VGG-like architecture, a proper update strategy for both CIFAR-10 and SVHN is to freeze all but the last 3 Conv and FC layers. This reduces the total latency by 21%.

Table IV compares our accelerators with prior CNN training works. Since the modified ResNet20 and VGG-like networks for CIFAR-10 and SVHN training are relatively compact, we are able to store entire forward activations on the FPGA logic to avoid bandwidth limitation incurred by off-chip DDR transfers of variable-sized input feature maps. The DSP48E2 is primarily used for MAC units in Conv blocks and index calculations. Although the overall throughput may be lower than works of batch parallelism [13], [19], when considering normalized throughput per batch, we achieve the (second) highest performance of 131 and 209 GOPs for ResNet20 and VGG-like respectively. Compared with the highest reported reference with high-bandwidth memory (HBM2) [12], BOOST with complete back-propagation requires 51% fewer DSPs and 21% fewer LUTs, but 22% more BRAM utilization (basically for the on-chip storage of forward activations). In comparison with an Nvidia GeForce GTX 1080 Ti GPU, we observed throughputs of 24 and 49 GOPs on ResNet20, and 77 and 139 GOPs on VGG-like for batch 1 and 2 respectively, with a power consumption of 55 Watt reported by the `nvidia-smi` tool. In such low-batch training scenarios of sequential processing for each batch, the FPGA implementation demonstrates better energy efficiency and outperforms the memory-bounded GPU in both throughput and power.

VI. Conclusion

We presented an FPGA-based TL accelerator, BOOST, for efficient CNN back-propagation training with BMs. Our design utilized a unified `bm(2,5)` precision for all data (e.g., weights, activations, errors, gradients, and momentum velocities) and achieved similar accuracy to FP32 for ResNet20 and VGG-like training on CIFAR-10 and SVHN. Our TL implementation utilizes fine-grained parallelism via pipelining and task-level parallelism to achieve high performance. Moreover, off-chip memory accesses are minimized by restricting off-chip DDR accesses to Conv layer weight and momentum velocity parameters, with other values including gradients and activations being kept on-chip. While our implementation supports full back-propagation, by applying TL where only several of the final Conv and FC layers are trainable, we demonstrated that significant latency and BRAM reduction can be achieved. Our future work will study how high-bandwidth memory can be utilized to further improve performance on large datasets and networks.

Acknowledgment

This work is supported by NSFC (Grant No. 62034007, 61974133, and 62141404), SGC Cooperation Project (Grant No. M-0612), and Zhejiang Provincial NSF (Grant No. LD21F040003).

REFERENCES

[1] Samuel Schulter, Paul Vernaza, Wongun Choi, and Manmohan Chandraker. Deep network flow for multi-object tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6951–6960, 2017.

[2] Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of tricks for image classification with convolutional neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 558–567, 2019.

[3] Andrea Galassi, Marco Lippi, and Paolo Torroni. Attention in natural language processing. *IEEE transactions on neural networks and learning systems*, 32(10):4291–4308, 2020.

[4] Bichen Wu, Forrest Iandola, Peter H Jin, and Kurt Keutzer. Squeezedet: unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 129–137, 2017.

[5] Donghyeon Han, Jinsu Lee, Jinmook Lee, and Hoi-Jun Yoo. A low-power deep neural network online learning processor for real-time object tracking application. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(5):1794–1804, 2018.

[6] Donghyeon Han, Dongseok Im, Gwangtae Park, Youngwoo Kim, Seokchan Song, Juhyoung Lee, and Hoi-Jun Yoo. Hnpu: an adaptive dnn training processor utilizing stochastic dynamic fixed-point and active bit-precision searching. *IEEE Journal of Solid-State Circuits*, 56(9):2858–2869, 2021.

[7] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point. *Advances in Neural Information Processing Systems*, 31, 2018.

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

[9] Ruizhe Zhao, Brian Vogel, Tanvir Ahmed, and Wayne Luk. Reducing underflow in mixed precision training by gradient scaling. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 2922–2928, 2021.

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

[11] Catherine F. Higham and Desmond J. Higham. Deep learning: an introduction for applied mathematicians. *SIAM Review*, 61(4):860–891, 2019.

[12] Shreyas K Venkataramanaiah, Han-Sok Suh, Shihui Yin, Eriko Nurvitadhi, Aravind Dasu, Yu Cao, and Jae-sun Seo. Fpga-based low-batch training accelerator for modern cnns featuring high bandwidth memory. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–8, 2020.

[13] Jinming Lu, Chao Ni, and Zhongfeng Wang. Eta: an efficient training accelerator for dnns based on hardware-algorithm co-optimization. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2014.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[16] Zhiqiang Liu, Yong Dou, Jingfei Jiang, Qiang Wang, and Paul Chow. An fpga-based processor for training convolutional neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 207–210. IEEE, 2017.

[17] Kaiyuan Guo, Shuang Liang, Jincheng Yu, Xuefei Ning, Wenshuo Li, Yu Wang, and Huazhong Yang. Compressed cnn training with fpga-based accelerator. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 189–189, 2019.

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

[19] Cheng Luo, Man-Kit Sit, Hongxiang Fan, Shuanglong Liu, Wayne Luk, and Ce Guo. Towards efficient deep neural network training by fpga-based batch-level parallelism. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 45–52. IEEE, 2019.

[20] Yue Tang, Xinyi Zhang, Peipei Zhou, and Jingtong Hu. Ef-train: enable efficient on-device cnn training on fpga through data reshaping for online adaptation or personalization. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2022.

[21] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[22] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[23] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 26–35, 2016.

[24] Maolin Wang, Seyedramin Rasoulinezhad, Philip HW Leong, and Hayden K-H So. Niti: training integer neural networks using integer-only arithmetic. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):3249–3261, 2022.

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

[26] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Viji Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. *Advances in neural information processing systems*, 32, 2019.

[27] Seock-Hwan Noh, Jahyun Koo, Seunghyun Lee, Jongse Park, and Jaeha Kung. Flexblock: A flexible dnn training accelerator with multi-mode block floating point support. *IEEE Transactions on Computers*, 2023.

[28] Hang Lu, Liang Chang, Chenglong Li, Zixuan Zhu, Shengjian Lu, Yanhuan Liu, and Mingzhe Zhang. Distilling bit-level sparsity parallelism for general purpose deep learning acceleration. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 963–976, 2021.

[29] Sergey Ioffe and Christian Szegedy. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[30] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. *Advances in neural information processing systems*, 31, 2018.

[31] Ruizhe Zhao, Ho-Cheung Ng, Wayne Luk, and Xinyu Niu. Towards efficient convolutional neural network for domain-specific applications on fpga. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 147–1477. IEEE, 2018.

[32] Brian Chmiel, Liad Ben-Uri, Moran Shkolnik, Elad Hoffer, Ron Banner, and Daniel Soudry. Neural gradients are near-lognormal: improved quantized and sparse training. In *International Conference on Learning Representations*, 2020.

[33] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[34] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. Fracbnn: accurate and fpga-efficient binary neural networks with fractional activations. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 171–182, 2021.