A Neural Network Training Processor With 8-Bit Shared Exponent Bias Floating Point and Multiple-Way Fused Multiply-Add Trees

Jeongwoo Park, *Graduate Student Member, IEEE*, Sunwoo Lee, *Graduate Student Member, IEEE*, and Dongsuk Jeon, *Member, IEEE*

***Abstract*— Recent advances in deep neural networks (DNNs) and machine learning algorithms have induced the demand for services based on machine learning algorithms that require a large number of computations, and specialized hardware ranging from accelerators for data centers to on-device computing systems have been introduced. Low-precision math such as 8-bit integers have been used in neural networks for energy-efficient neural network inference, but training with low-precision numbers without performance degradation have remained to be a challenge. To overcome this challenge, this article presents an 8-bit floating-point neural network training processor for state-of-the-art non-sparse neural networks. As naïve 8-bit floating-point numbers are insufficient for training DNNs robustly, two additional methods are introduced to ensure high-performance DNN training. First, a novel numeric system which we dub as 8-bit floating point with shared exponent bias (FP8-SEB) is introduced. Moreover, multiple-way fused multiply-add (FMA) trees are used in FP8-SEB's hardware implementation to ensure higher numerical precision and reduced energy. FP8-SEB format combined with multiple-way FMA trees is evaluated under various scenarios to show a trained-from-scratch performance that is close to or even surpasses that of current networks trained with full-precision (FP32). Our silicon-verified DNN training processor utilizes 24-way FMA trees implemented with FP8-SEB math and flexible 2-D routing schemes to show 2.48× higher energy efficiency than prior low-power neural network training processors and 78.1× lower energy than standard GPUs.**

***Index Terms*— Computational efficiency, digital integrated circuits, learning systems, neural network accelerators, very large-scale integration.**

I. Introduction

THE advance of deep neural networks (DNNs) in machine learning has triggered many research and commercial application-specific integrated circuits (ASICs) or processors with a domain-specific architecture that specialize in processing computation-heavy DNNs, such as convolutional neural networks (CNNs) and their variants [1]–[9]. These designated processors are often designed on the ground that neural networks are generally error-tolerant, implementing a low-precision arithmetic such as 8-bit integers (INT8) for lower power consumption in both combinational logic and memory accesses [7], [10]. As such, hardware designers have adopted quantization to make maximum use of the error tolerance, with some works providing a wide range of numerical precision on the same hardware with flexible arithmetic units. Another common technique employed for efficient DNN processing in hardware is utilizing numerical sparsity, in either activations, weights, or both. For instance, some of the works on mobile DNN processors utilize sparsity in activations by skipping zero-multiply computations, dynamically reducing latency at model runtime [1], [2], [7], or simply gate local computation units [6].

However, the grounds for which prior DNN accelerators were designed—error tolerance of neural networks that allowed low-precision arithmetic such as INT8 and sparsity of neural networks exploited in zero-skip architecture—have recently changed. For example, due to the shift in the deep learning algorithm community toward using non-sparse activation functions such as Leaky ReLU or Swish, state-of-the-art models no longer exhibit the sparsity found in conventional ReLU-based models [11]. This shift is due to two main reasons: better training convergence properties of gradients in these new nonlinear functions, and the growing evidence that smooth gradient in nonlinear activations provides more robustness against adversarial attacks [12]. Our analysis on various state-of-the-art models on five different tasks—image classification, image recognition, image generation, language modeling, and speech recognition—shows that the average sparsity of models has decreased significantly for both inference and training compared to their predecessors (see Fig. 1). This change prompts accelerator designs that do not rely on sparsity and rather perform more reliably on dense inputs. Moreover, contrary to error-tolerant image classification tasks, more challenging tasks such as image super-resolution require higher precision than plain 8-bit integers not just for training, but for inference to avoid large accuracy degradation. Fig. 2 shows a sample image for a super-resolution task showing large signal-to-noise ratio (SNR) drop as well as qualitative degradation when implemented in naïve 8-bit integer inference compared to its full-precision (IEEE Float32) counterpart.

Manuscript received April 3, 2021; revised July 13, 2021; accepted August 4, 2021. Date of publication August 17, 2021; date of current version February 24, 2022. This work was supported in part by the National Research Foundation of Korea under Grant NRF-2019R1C1C1004927, in part by the Korea Institute of Science and Technology (KIST) Institutional Program under Grant 2E31031-21-038, and in part by the IC Design Education Center. This article was approved by Associate Editor Vivek De. *(Corresponding author: Dongsuk Jeon.)*

The authors are with the Graduate School of Convergence Science and Technology, Seoul National University, Seoul 08826, South Korea, and also with the Inter-University Semiconductor Research Center, Seoul National University, Seoul 08826, South Korea (e-mail: djeon1@snu.ac.kr).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/JSSC.2021.3103603.

Digital Object Identifier 10.1109/JSSC.2021.3103603

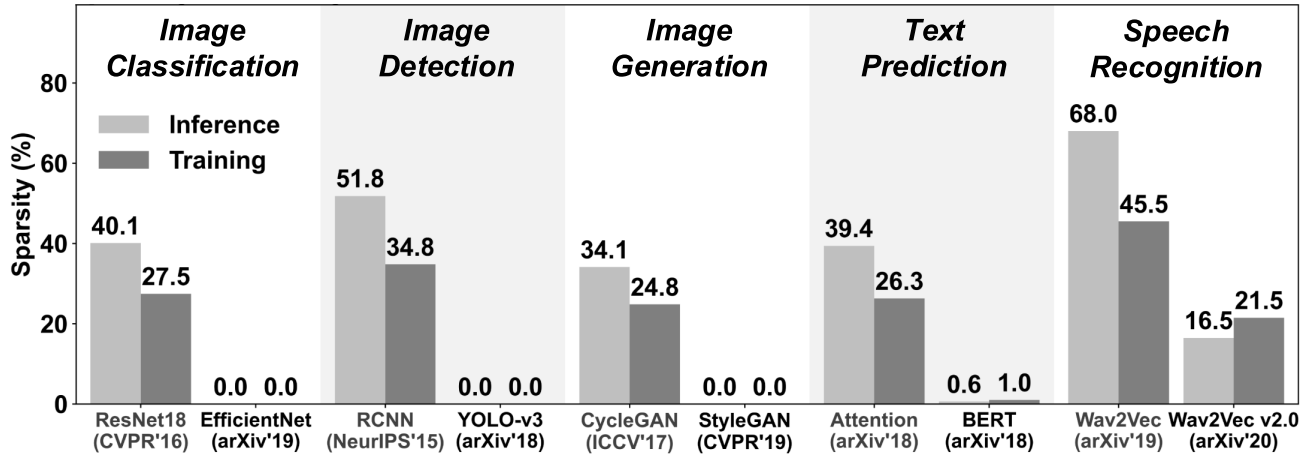

Fig. 1. Sparsity of widely used neural networks and their predecessors.

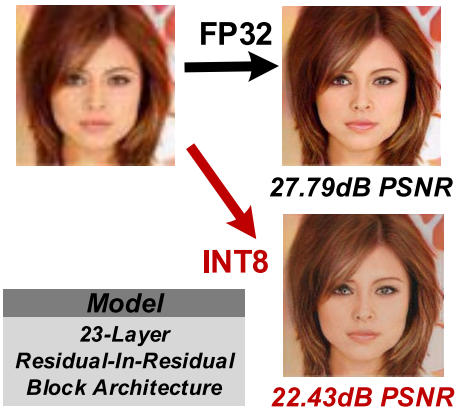

Fig. 2. More error-sensitive models such as image super-resolution may show quality degradation with 8-bit integer inference.

Another aspect of the DNN accelerators that must be considered is whether they are intended for inference, where models are run with fixed parameters, or for training where the gradients for the parameters are computed to determine their changes for their respective inputs. Even in mobile DNN accelerators, the training feature could be useful for implementing novel training paradigms such as Federated Learning [13], which allows a single collective model to be trained without users' private data being sent to centralized data centers or for adjusting each end user's model according to their preferences. Typically, training is thought of as requiring more numerical precision compared against inference and requires more hardware resources when implemented in DNN accelerators [14].

These environmental changes offer new challenges faced by mobile deep learning processors; they must process non-sparse networks efficiently, maintain higher precision for more challenging tasks, and provide features for training neural networks with minimal hardware overhead. In our processor design, these challenges are overcome with three novel approaches that complement each other. These approaches are as follows.

1) 8-bit floating-point number system with shared exponent bias.
2) N-way fused multiply-add (FMA) trees for lossless processing.
3) Routing at input and output datapoints for area-efficient homogeneous datapath.

The 8-bit floating-point number system, which we dub float8 with shared exponent bias (FP8-SEB), shows a higher precision that does not degrade with more challenging tasks such as the image super-resolution task mentioned above. It is also shown that FP8-SEB successfully trains various models from scratch with a performance that is similar to, or even outperforms full-precision baselines. This higher precision of FP8-SEB format does not come at the cost of more energy; the implementation style we discuss in more detail in the latter part of this article, N-Way FMA Trees, leads to 88% lower energy consumption compared against conventional implementation of our FP8-SEB datapath with multiply-accumulate (MAC) units. Lastly, this processor implements a datapath that routes only at the input and output data points and not at the intermediary datapoints. This allows low-overhead routing units, both in terms of area and energy, while showing flexibility for training on N-way FMA trees introduced in this article.

The remainder of this article expands on the three novel approaches in more detail and elaborates on points discussed in our conference paper [15]. Section II introduces FP8-SEB, discusses the mathematical formation behind the number system, and analyzes its training performance. Section III discusses various components of the processor, validating the efficiency of the proposed processing with N-Way FMA trees and depicting the exact datapath for computing convolution feedforward and backward steps. Section IV analyzes energy efficiency, external memory access (EMA), and final performance on various benchmark DNN models of our processor as well as compare them against prior works. Section V concludes this article.

II. FP8 With Shared Exponent Bias

For robust training with lower precision in our processor, we propose a new arithmetic system, 8-bit floating point with shared exponent bias (FP8-SEB). We compare this format against prior works on training neural networks with limited

precision [1], [2], [16]–[18]. Some of the works suffer from training accuracy degradation on large, complex networks [16], [19]. Some other works are more costly in terms of hardware compared to our work, due to using mixed precision of FP8/FP16 [1], [2], or because they require using different FP8 formats in different computation paths of neural network training [17], which is realized as FP9 datapath [5]. We find that an independent work [18] published after our conference work is very similar to our proposed FP8 format, using similar core ideas such as using shared exponent biases though differs in terms of the scale of the bias-sharing. Our proposed FP8-SEB overcomes these limitations through using a homogeneous datapath for dominant matrix multiplication-like operations, while showing training performance close to that of full precision training on a wide range of machine learning tasks.

In FP8-SEB, the elements of a tensor are represented in 1-4-3 (1-bit sign, 4-bit exponent, and 3-bit mantissa) FP8 format, with the exponent being biased differently for each tensor according to the dynamic range of the tensor, as illustrated in Fig. 3. In this work, we group all output activation elements of a single layer, and all weight elements of a single layer in a DNN as a tensor, respectively, although implementations may vary to accommodate different types of grouping mechanisms for sharing a bias. The real value for a single-element x represented in FP8-SEB with sign s, 4-bit unsigned exponent e, 3-bit unsigned mantissa m, in a tensor with an unsigned 8-bit bias b, is given by

$$x_Q = (-1)^s 2^{e-127+b}\left(1+\frac{m}{8}\right). \tag{1}$$

The extra cost of using separate biasing should be carefully analyzed for comparison against other quantization methods. The first cost associated with the separate biasing is the extra memory it incurs for storing the shared bias. However, our analysis shows that this cost is negligible. In ResNet-18 training, we find that the shared biases only occupy 0.003% in terms of memory during ResNet-18 training.

Other costs may be incurred from implementing the hardware logic while being aware of the fact that any biases could be taken in the input tensors. At first glance, it may appear that extra logic is required, using flexible biasing logic during normalization step in floating-point arithmetic. While this is true for some operations (e.g., this holds in the case for addition between two FP8 tensors with different exponent biases), in the case of matrix multiplications between two different tensors, we could formulate a mathematical rearrangement which minimizes hardware logic. In order to perform dot product or matrix multiplications in FP8-SEB tensors, bias should be decoupled from the tensor elements themselves to maximize computational efficiency. The accumulated value Y as the result of matrix multiplication between tensor X_1 and tensor X_2 is given by

$$Y = 2^{-254+b_1+b_2}\sum(-1)^{S_1+S_2}2^{e_1+e_2}\left(1+\frac{m_1}{8}\right)\left(1+\frac{m_2}{8}\right). \tag{2}$$

After accumulation is completed, the resulting tensor is re-cast back to the FP8 format. Denoting the accumulated tensor in the 1-6-23 format by Y_{acc} [represented with 1-bit

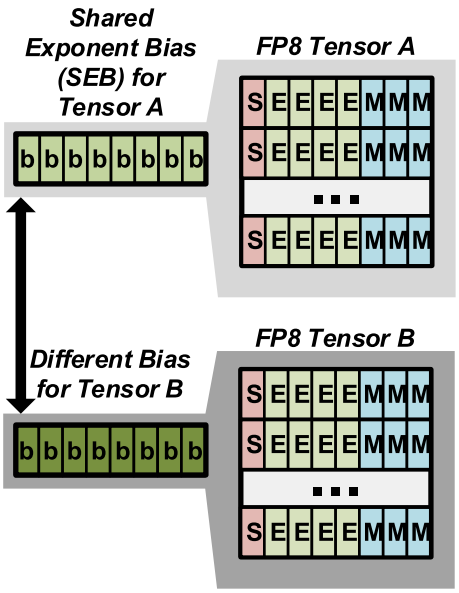

Fig. 3. Illustration of FP8-SEB format.

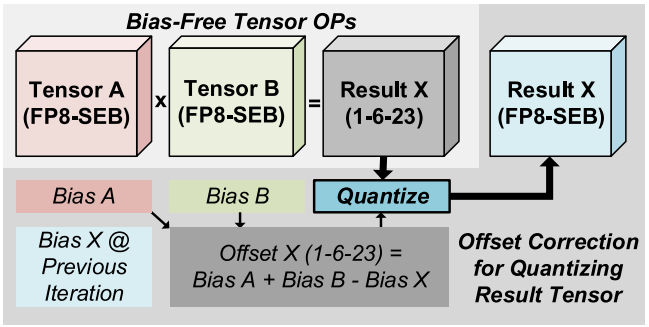

Fig. 4. Matrix multiplication in FP8-SEB.

S_{acc}, 6-bit e_{acc}, and 23-bit m_{acc} as in (3)] and the re-quantized 1-4-3 tensor by Y_Q with bias b_{Y_Q}, we obtain the following re-quantization equations:

$$Y_{\text{acc}} = (-1)^{S_{\text{acc}}}2^{e_{\text{acc}}-(b_1+b_2)+127}\left(1+\frac{m_{\text{acc}}}{2^{23}}\right) \tag{3}$$

$$S_{Y_Q} = S_{\text{acc}} \tag{4}$$

$$e_{y_Q} = \text{clamp}(e_{\text{acc}} - (b_1+b_2+b_{Y_Q}) + 254, 0, 15) \tag{5}$$

$$m_{Y_Q} = \text{round}\left(\frac{m_{\text{acc}}}{2^{20}}\right) \tag{6}$$

where

$$\text{clamp}(x, \min, \max) = \begin{cases} \min, & x < \min \\ \max, & x > \max \\ x, & \text{otherwise} \end{cases}$$

and round refers to rounding to the nearest integer. Therefore, the only extra logic would be for the re-quantization function, and any bias value (in our formulation, 0 is chosen) could be chosen for the adders and the accumulators. This matrix multiplication is depicted in Fig. 4.

The last cost that could be associated with separate exponent biasing is with determining the actual value for the exponent bias. The work in [18] determines this value by finding the

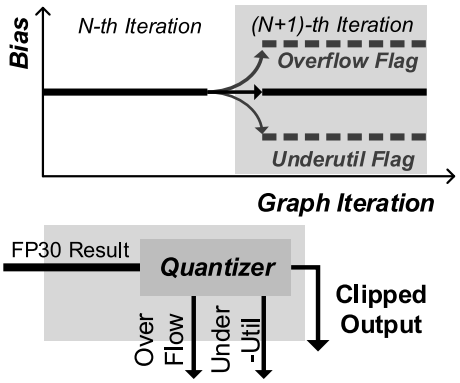

Fig. 5. Bias tracking and update algorithm.

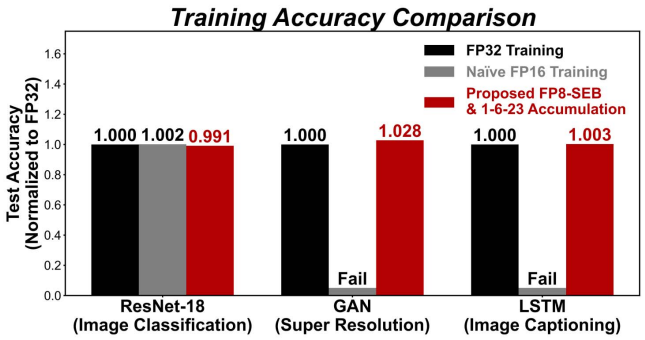

Fig. 6. Training accuracy comparison.

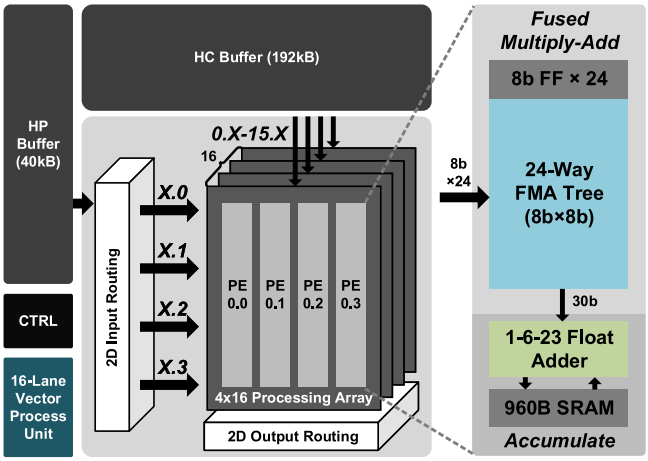

Fig. 7. Architecture diagram of our processor.

max/min values for the entire block during run-time; this will incur $O(n)$ overhead for each biasing and may incur significant more overhead if the bias value of the final accumulated tensor has not been determined and the high-precision accumulated tensor needs to be stored to off-chip memory before finding the actual exponent biases. Contrarily, we only require this max/min value search for the first iteration of training steps; after the first biases are determined, it is assumed that the biases of the prior training step remains the same unless it has overflowed or underutilized. As shown in Fig. 5, one would simply increment or decrement the bias value after the computation loop for the next training batch. In the case of overflow or underflow, the strategy is to simply clamp each of the overflowed or underflowed elements to the maximum and minimum representable values in FP8 formats, respectively, as in (5). Although more complicated bias tracking algorithms could be considered, similar to an algorithm predicting the next bias overflow in [20], we find that this simple algorithm depicted could be applied without any accuracy degradation. Therefore, by using the implementation techniques discussed above, the hardware cost of the shared exponent biasing could be mitigated.

The first advantage of using separate bias for each tensor is that it allows different tensors such as activation and weight gradients, which have vastly different dynamic ranges [20], [21], to be represented accurately while avoiding overflow. Moreover, the separate exponent biasing in FP8-SEB allows an extra mantissa bit, leading to higher precision in computation compared against the 1-5-2 FP8 format proposed in [16] and leads to 1.6% higher top-1 accuracy in ResNet-18 ImageNet classification task.

The numerical precision that is used for the intermediary accumulation is crucial for training neural networks without loss in final training performance, as suggested in [16]. For instance, our experiments on a small-scale network (LeNet on CIFAR-10) show that using the same 8-bit accumulation precision results in 5% accuracy loss in final trained accuracy. On deeper networks such as ResNet-18, we find that the training simply fails to converge when using 8-bit accumulations. Moreover, sometimes even half-precision (FP16, 1-5-10 format) training may show failure on models that are more difficult to optimize. Our experiments on generative adversarial network (GAN) for image super-resolution task and long short-term memory (LSTM) network for language modeling show failure in training with FP16, as shown in Fig. 6, although different numeric formats in FP16 such as brainfloat (1-8-7 format) may mitigate this training accuracy degradation. We conservatively use FP30 accumulation with the 1-6-23 format (such that it would attain the same number of mantissa bits as full-precision numbers, which also consists of 23 mantissa bits), which successfully shows training performance that does not degrade from the full-precision baseline in our simulations. In this numeric pipeline, storing the high precision intermediary value to off-chip memory is heavily penalized in an energy-constrained system. Therefore, it is imperative for the computational flow to be arranged such that the intermediary accumulation values should be avoided being stored in off-chip memory and rather reside in the local on-chip memory.

III. Processor Design

A. Top-Level Architecture

The key principle that shapes our architecture is minimizing the cost of data movements. In order to realize this principle, the cost of memory access is reduced through a computing architecture that minimizes on-chip memory access, as well as carefully choosing the computing order which minimizes memory access to both on-chip memory and off-chip memory.

When selecting the order of computations, off-chip memory accesses are more heavily penalized as they are more costly in orders of magnitude compared to on-chip memory access [22].

The top-level architecture of the processor consists of a 4×16 processing element array, each of which processes 24 FMAs per cycle, 2-D routing at the input–output points of the processing elements, two main buffers with 40 and 192 kB each storing activations and weights, and a 16-lane vector processor for handling auxiliary operations required in neural network training (see Fig. 7). Each processing element contains a 24-way FMA tree, a high-precision bias-free 1-6-23 adder, and a 960 B of scratchpad memory for storing accumulated values. Contrary to conventional works that make use of MAC units, our processing logic is based on multiple-way FMA trees that could amortize the cost of memory access associated with partial sums accumulation.

The 16-lane vector processor supports various operations, including linear vector operation in FP8/FP16 mixed precision, various pooling and activation functions, and scalar FP32 arithmetic operations to support error-sensitive computing such as computation on determining batch normalization constants. The execution pipeline and arithmetic units of the processor is shown in Fig. 8. The processor consists of three main execution blocks: 1) floating-point vector execution unit; 2) integer execution unit; and 3) matrix multiplication and convolution unit. After the fetched instruction is decoded, either one of the execution unit processes on the instruction according to their processing category. For example, convolution-feedforward instructions would be ran in the main processing logic, while linear AXPY operations (A*X+Y) would be executed by the FP vector execution unit.

The FP vector execution unit handles the auxiliary operations required for end-to-end neural network training. The mixed-precision AXPY unit, as shown in Fig. 8, could be reconfigured to use FP16 with 1-bit sign, 8-bit exponent, and 7-bit mantissa (known as brainfloat) or to use FP16 with 1-bit sign, 6-bit exponent, and 9-bit mantissa. They are implemented with a single fused-multiply-addition of FP18 (1-8-9 format) for efficient resource sharing.

In addition to simplified RISC-style instructions, our processor supports custom instructions that pipeline some of the commonly used functions that are utilized during neural network training or inference. For example, by using an optimized instruction for stochastic gradient descent (SGD) in place of combinations of vector instructions, the runtime for weight updates is reduced by an average of 31.7%. The sample custom pipeline and FSM that implements SGD instruction is shown in Fig. 9. The SGD instruction takes current weights W_t, current weight gradients gW_t, momentum M_t, and three constants d, μ, and lr as input and updates the weights and the moving momentum with a combination of three equations; weight decay [see (7)], momentum update [see (8)], and weight update [see (9)]

$$gW'_t = d * W_t + gW_t \tag{7}$$

$$M_{t+1} = \mu * M_t + gW'_t \tag{8}$$

$$W_{t+1} = W_t - lr * M_{t+1}. \tag{9}$$

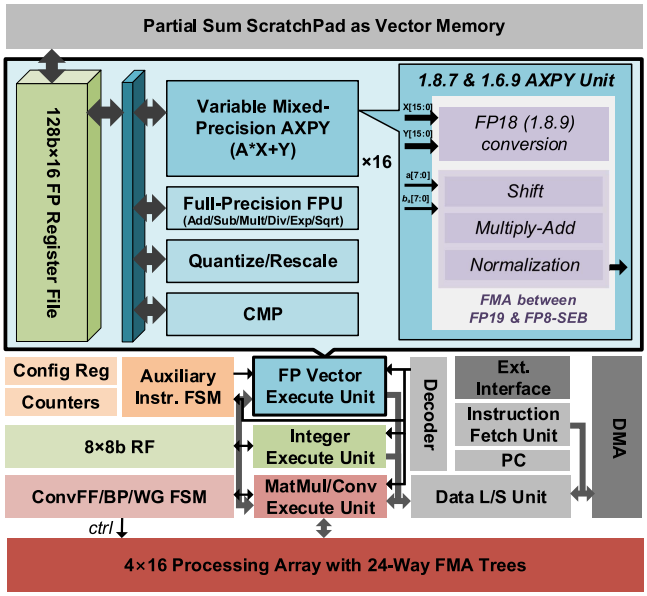

Fig. 8. Detailed view of the vector processor.

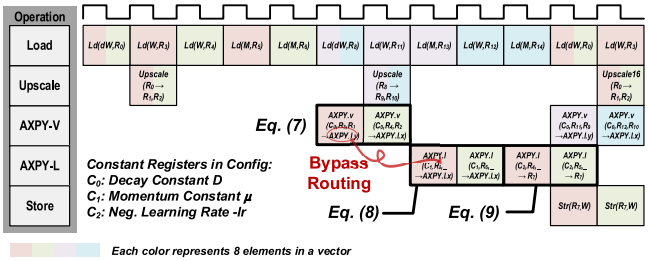

Fig. 9. Cycle-level view of SGD instruction.

The optimized instruction allocates the registers (denoted by R_X in Fig. 9) with different elements of the input tensor (denoted by the colors in Fig. 9, with each color holding eight elements) and utilizes different arithmetic units in the same cycle, allowing higher utilization compared against separate instructions that could only utilize one of the arithmetic units at a cycle. For example, as shown in Fig. 9, two different AXPY units (AXPY-L for AXPY unit with stochastic LFSR rounding and AXPY-V for vanilla AXPY unit) and an upscaler that upscales 1-4-3 FP8 elements to either 1-8-7 or 1-6-9 FP16 formats are utilized in the same cycle using the optimized instruction. Other custom optimized instructions include softmax, channel-wise mean/var, maxpool, and average pools.

B. N-Way FMA Trees

We have adopted an N-way FMA tree depicted in Fig. 10, rather than conventional MAC-based processing schemes for implementing our numeric system (FP8-SEB as inputs, FP30 for intermediary accumulations). Implementing low-precision floating-point matrix multiplications in FMA trees suppress precision loss compared to MAC-based processing, as small normalization errors ("swamping") displaying non-associative properties [16], [23] could be avoided. This

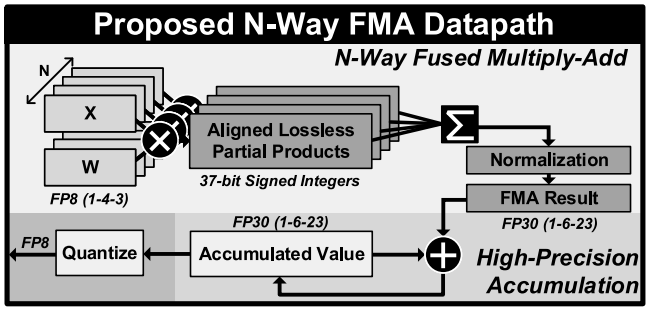

Fig. 10. Pipeline of the proposed N-way FMA trees.

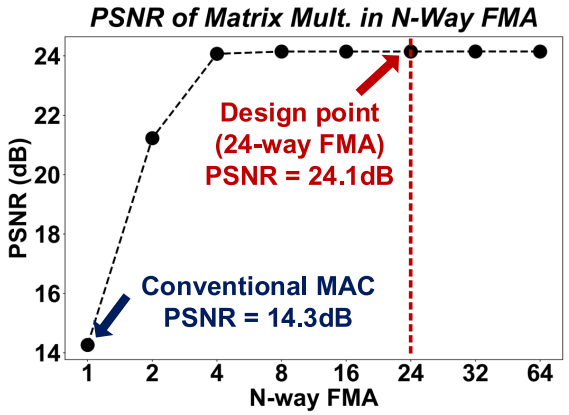

Fig. 11. PSNR in matrix multiplication using different configurations of N in N-way FMA trees.

TABLE I
ENERGY CONSUMPTION AND AREA*, SYNTHESIZED RESULTS

	1-Way MAC, FP8	24-Way FMA, FP8
Comb. (pJ)	5.8	1.5
Seq. & Memory (pJ)	10.1	0.4
Area**(μm^2)	3081.6	848.9

*Costs are normalized per MAC
**Excludes SRAM

swamping effect is caused by the absolute difference in magnitude between the operands of addition. The output of a typical MAC is much smaller than the partial sum, and hence it losses a significant amount of information when added to a much larger partial sum. However, as more values are accumulated in the FMA tree before being added to the partial sum, the increment value displays less chance of swamping. Fig. 11 shows the effective peak signal-to-noise ratio (PSNR) in matrix multiplication using various N-way FMA configurations, clearly demonstrating the trend that the wider the FMA trees, the higher the precision. Results were obtained on a fixed matrix size of 1024 × 1024, and PSNR was calculated against full-precision matrix multiplication result.

These improvement in accuracy is not a simple trade-off between accuracy and hardware costs such as more logic or energy; the entire energy consumption associated with each MAC actually reduces from 15.9 pJ (one-way FMA) to 1.9 pJ (24-way FMA). This large reduction is possible due to amortizing the high cost of intermediary 1-6-23 accumulation, both in the logic itself and with the cost of accessing the memory for loading and storing the intermediary partial sums. In the case of N-way FMA trees, this cost could be amortized over N, as only one high-precision accumulation is required per MAC and the combinational logic could be optimized more in the FMA tree itself as the value of N gets larger, since implementing an N-way adder tree provides more space for logic optimization compared against the cost of a single adder implementation multiplied by N. Table I summarizes various synthesized results in different N values for the energy consumption and area, implemented in a 40-nm LP CMOS process.

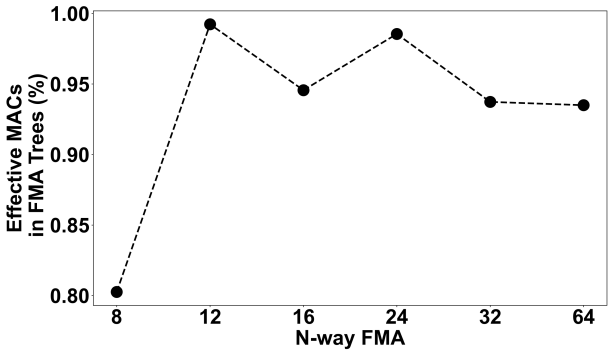

Fig. 12. Ratio of valid MACs in N-way FMA trees. The results are shown in the feedforward stage of ResNet-18.

The reason for choosing a 24-way FMA tree, and not a number of N-ways, where N is some power of 2 (16, 32, etc.) is that the effective ratio of active MACs in the FMA trees is reliant on being divisible by the width of the convolution kernels. As seen in Fig. 12, as 12 and 24 provide plenty of divisors, they have a better ratio of effective MACs compared against power-of-2 N choices. Although 12-way FMAs show slightly better utilization over 24-way, 24-way is chosen as they show better area and energy efficiency through sharing the resources required for accumulation, both in terms of combinational logic and the implicit cost of reading and writing the accumulation values to memory.

C. On-Chip Buffer Organization

The on-chip memory is grouped as high-performance buffer (HP buffer, 40 kB), high-capacity buffer (HC buffer, 192 kB), and local scratchpads (spad, 60 kB). The HP buffer, consisting of 32 SRAMs with 40-bit word size (for four 8-bit elements that are used by the 4 × 16 processing array and a prefetching element), is mainly used for highly accessed data such as activations and gradients. Prefetchers are implemented in the HP buffer to reduce the memory stall cycles during computation, so that the prefetched elements of the tensors are loaded before they are needed in computation. Effective utilization of the computation units in various ResNet-18 layers is shown in Fig. 13, compared against inference without prefetching.

TABLE II
ON-CHIP AND OFF-CHIP MEMORY ACCESS IN ALEXNET LAYERS

Layer Types	This Work				Eyeriss [6]		
	On-Chip Buffer			Off-Chip DRAM	On-Chip Buffer		Off-Chip DRAM
	HP Buffer	HC Buffer	ScratchPad		Global	ScratchPad*	
Conv1: InCh=3, OutCh=64, Kernel=(11,11)	3.1MB	6.1MB	486.4MB	1.0MB	18.5MB	1938.6MB	5.0MB
Conv2: InCh=64, OutCh=192, Kernel=(5,5)	10.7MB	7.0MB	320.4MB	0.3MB	77.6MB	4038.5MB	4.0MB
Conv3: InCh=192, OutCh=384, Kernel=(3,3)	8.9MB	7.6MB	133.7MB	0.3MB	50.2MB	2594.6MB	3.0MB
Conv4: InCh=384, OutCh=256, Kernel=(3,3)	11.9MB	10.1MB	178.2MB	0.6MB	37.4MB	1904.3MB	2.1MB
Conv5: InCh=256, OutCh=256, Kernel=(3,3)	7.9MB	6.8MB	118.8MB	0.4MB	24.9MB	119.2MB	1.3MB
Total	**42.4MB**	**37.64MB**	**1238.5MB**	**2.6MB**	**208.5MB**	**11665.2MB**	**15.4MB**

*Inferred from original paper, using 2*16b*(# of Active PEs)*(# of Processing Cycles) as ScratchPad access count

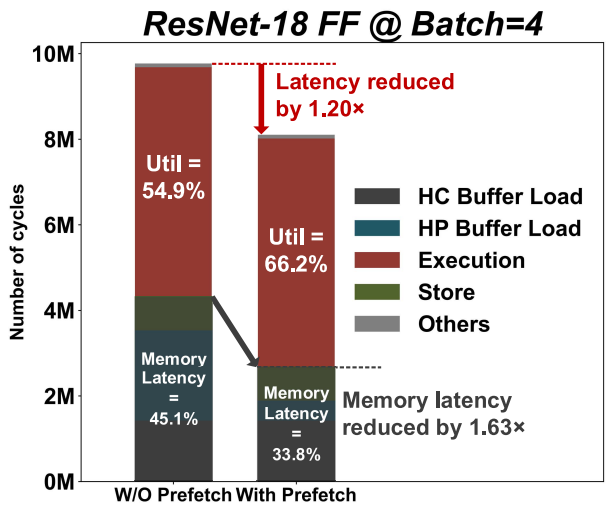

Fig. 13. Breakdown of latencies in ResNet-18 feedforward stage.

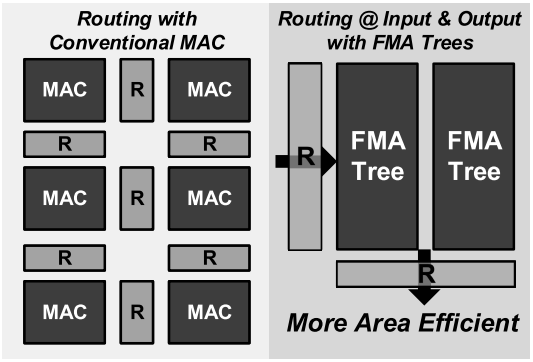

Fig. 14. Routing on input–output data points instead of between intermediary data points.

Input data are organized carefully in HP buffers, HC buffers, and local scratchpads to maximize data re-use, both when they are read from off-chip memory to on-chip buffers, and when they are read out from on-chip buffers to the computation units. For example, during convolutional feedforward (conv-ff) operations, weights are loaded to the HC buffer and the activations are loaded to the HP buffer. For reducing on-chip buffer accesses, weights read out from the HC buffer are shared across four processing rows of the height (H)-axis of input tensors, reducing HC buffer access by 4×. The activations read out from the HP buffer are aligned according to the kernel widths (KW) and heights (KH) to reduce data read/writes by KW × KH. In terms of off-chip memory access, only the inputs may be read more than once, so that redundant loads are avoided as much as possible and redundant stores do not occur. The case of iterative input loads occur if the weight tensor does not fit into the HC buffer; weight buffers are then split along the output channel axis so that the output tensors could be stored in the off-chip memory after they are computed completely, minimizing off-chip memory access in output tensors. In this case, input tensors need to be fetched iteratively by the number of "splits" required in the HC buffer.

The average on-chip memory access count and off-chip memory access count are analyzed on the configurations of AlexNet [24] layers, compared against Eyeriss CNN accelerator [6], which is shown in Table II. Our architecture with N-way FMA trees achieves the goal of reducing high powers of local scratchpads, reducing scratchpad access by 89.3% (94.3% if data widths of partial sums are normalized to 16 b) compared against the design in [6]. Moreover, it does not come at the cost of more memory access in other types of buffers—on-chip buffer access for weights and activations amount to 80.08 MB, which is a reduction of 61.6% (23.2% if data widths of weights and activations are normalized to 16 b). Total EMA is reduced by 83.4% (66.8% if data widths of data access are normalized to 16 b). As scratchpads took up to 42.5% of power in [6], the buffer organization and FMA tree-based processing used in this processor is expected to yield 40.0% of total power reduction if implemented on the same processor, just from the reduced energy in accessing scratchpad memory.

D. Dataflow for CNN Training

Fig. 14 depicts the spatial-locality-aware routing on input–output data points that are implemented for supporting both training and inference of neural networks efficiently with N-way FMA trees. By restricting routing units to

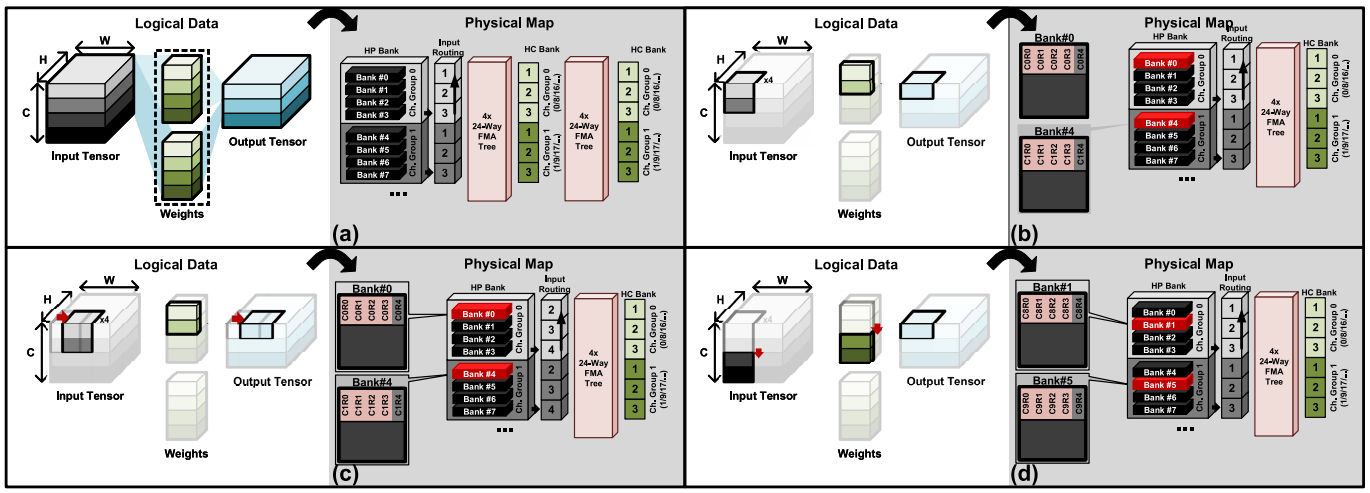

Fig. 15. Illustration of the conv-feedforwad functionality implemented in our processor. (a) General processing in Conv-FF step. (b) In the case of KW = 3, eight channels are processed at a single time-step. (c) Pixel values read out from the HP buffers are shifted in the next cycle for minimized data read from HP buffers. (d) After the accumulation along the W-axis is finished, different sets of channels are selected for computation.

inputs/outputs of the processing arrays instead of on intermediary datapoints, routing resources only take up 0.34% of the total area and 0.72% of the total energy.

These routings are enough to provide flexibility for different types of convolution-like computations required in training; during feedforward stage of convolution layers, routings on input points are used to make use of the spatial locality between input data. During the convolution feedbackward stage and deconvolution operations, routings on output points are used for making use of the spatial locality in output data. For instance, in the feedforward stage of convolutional layers, each of the 24-way FMA trees takes pixel values of different input channels read from high-performance buffers. The 32 SRAMs in the HP buffer are grouped into different channel groups, with the number of channel groups and the channel group index for the input data with channel c given by the equations below

$$N_{\text{chGroup}} = \text{floor}\left(\frac{24}{\text{KW}}\right) \tag{10}$$

$$N_{\text{bankPerGroup}} = \text{floor}\left(\frac{32}{N_{\text{chGroup}}}\right) \tag{11}$$

$$\text{index}_{\text{chGroup}}(c) = c \bmod N_{\text{chGroup}} \tag{12}$$

$$\text{index}_{\text{bank}}(c) = c \bmod (N_{\text{chGroup}} * N_{\text{bankPerGroup}}). \tag{13}$$

The case of convolution with a KW of 3 is shown in Fig. 15. The input activation tensor, as shown in Fig. 15(a), is split along the channel (C)-axis and mapped to the HP banks with the group index given by (12). Moreover, as an HP bank may not be large enough to accommodate all of the elements in the input activation tensor, maximum of only five rows along the H-axis are physically stored in the HP bank. Fig. 15(b) shows the elements in the input activations that could be read out and processed in a single cycle. In bank #0, the first five rows of the first channel is stored (C0R0 to C0R4), along with other channels that are also mapped to bank index #0. Of the five rows that are mapped to the bank #0, a single element from each of the four rows (one row is excluded as it is used for prefetching) is read out in a single cycle. The read out values are then aligned in the input width dimension by the shift registers in the input routing unit such that the consecutive three-pixel elements in the window of convolution could be sent to the FMA trees along with seven different input channels. In this way, the input router maps eight different input channels from eight different banks and three consecutive elements to a single FMA tree, whereas four parallel rows of the input activations are mapped to the "four" rows of the FMA trees that share HC banks. This mapping is

Algorithm 1 Conv-FF Implementation

$N_{OCSplit} = Ceil(\frac{WeightSize}{HCBufferSize})$
Initialize $O[Batch, O_{Ch}, O_H, O_W] \leftarrow 0$
for $O_{chSplit} \leftarrow 1$ to $N_{OCSplit}$ **do**
  Load split output channel weights to HC Buffer
  $StartO_{ch} = \frac{O_{ch}}{N_{OCSplit}} * (O_{chSplit} - 1)$
  $EndO_{ch} = \frac{O_{ch}}{N_{OCSplit}} * O_{chSplit}$
  for $b \leftarrow 1$ to $Batch$ **do**
    for $r \leftarrow 1$ to $InputHeight$ **do**
      for $o_{ch} \leftarrow StartO_{Ch}$ to $EndO_{Ch}$ **do**
        for $kh \leftarrow 1$ to $KernelHeight$ **do**
          for $IC_{Split} \leftarrow 1$ to $N_{bankPerGroup}$ **do**
            Change channel selection mux
            Read Weights to FMA Tree
            $i_{Ch} = N_{chGroup} * IC_{Split}$
            for $p \leftarrow 1$ to $InputWidth$ **do**
              $O[b, o_{ch}, row, p] +=$
$\sum I[b, i_{ch} : +N_{chGroup}, r, p : +KW]$
$*W[o_{ch}, i_{ch} : +N_{chGroup}, kh, :]$
            end for
          end for
        end for
      end for
      Store Output Row to Ext. Memory
    end for
  end for
end for

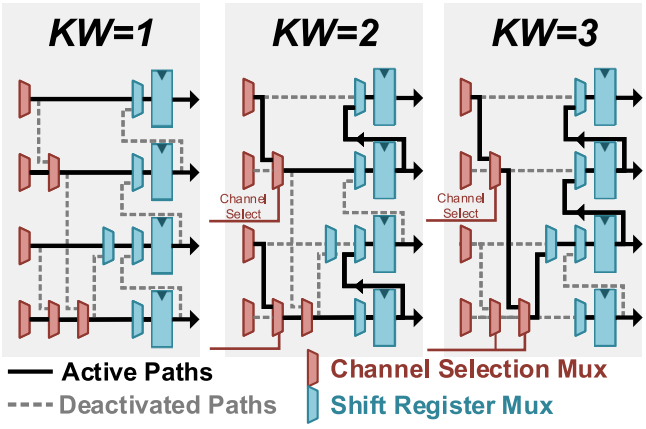

Fig. 16. Input routing supports row alignment, channel selection, and spatial shifts. Three different configurations with the KW from 1 to 3 are shown.

shared for a maximum of 16 different output channels along the 16 columns of the FMA trees. The elements of the input, weight, and output tensors that are mapped in a single cycle is highlighted in the left part of Fig. 15(b). In the following cycle, the pixel values read from eight different channels are then shifted along the W-axis to make the maximum use of the spatial locality between these pixels [see Fig. 15(c)]. After accumulation along the W-axis of the input tensor is finished, a different bank from the same channel group is selected for next processing [see Fig. 15(d)]. When all the different channels are accumulated, the next rows along the H-axis of input tensors are processed until all of the output channels and the batches are computed.

The pseudo-code for this computational flow in conv-ff is shown in Algorithm 1. To maximize data reuse on a restricted size of on-chip memory, we split the output channels by the maximum allowable output channels in the on-chip memory, iterating for N_{OCSplit} times, as shown in Algorithm 1, line 1. At each of the iteration, output channels from $\text{Start}O_{\text{ch}}$ to $\text{End}O_{\text{ch}}$ are completely processed without storing intermediary accumulation values to the external memory, as in lines 7–21. After the corresponding weights have been loaded on-chip, the order of processing is from the innermost loop to the outermost loop as shown in line 15 and lines 11–7, starting from: 1) pixels along the input feature's width (p); 2) chunks of input channels (IC_{split}); 3) KH; 4) a portion of the split output channels (o_{ch}); 5) input dimension's height (r); and 6) batch (b). To allow this type of processing, pixel values read out from the HP buffer go through the global input routing unit consisting of barrel shifters, three-stage channel selection mux, and shift registers, which, respectively, align data according to their rows, channels, and spatial locations in accordance with Algorithm 1, as shown in Fig. 16. The channel selection muxes, shown in red, are selected according to the KW and the current IC_{Split} (as in Algorithm 1) to select the current bank that is being read out for processing. The shift register muxes, shown in blue, are selected according to the KW in such a way that the shift registers have a chain length that is equal to the KW.

Algorithm 2 Conv-BP Implementation

1: $N_{O_{Ch}} = 16//KernelWidth$
2: $N_{ICSplit} = Ceil(\frac{WeightSize}{HCBufferSize})$
3: Initialize $I_{grad}[Batch, O_{Ch}, O_H, O_W] \leftarrow 0$
4: **for** $I_{chSplit} \leftarrow 1$ to $N_{ICSplit}$ **do**
5: $StartI_{ch} = \frac{I_{ch}}{N_{ICSplit}} * (I_{chSplit} - 1)$
6: $EndI_{ch} = \frac{I_{ch}}{N_{ICSplit}} * I_{chSplit}$
7: Load split input channel weights to HC Buffer
8: **for** $b \leftarrow 1$ to $Batch$ **do**
9: **for** $r \leftarrow 1$ to $InputHeight$ **do**
10: Load Row to HP Buffer
11: **for** $i_{ch} \leftarrow StartI_{Ch}$ to $EndI_{Ch}$ **do**
12: **for** $kh \leftarrow 1$ to $KernelHeight$ **do**
13: **for** $OC_{Split} \leftarrow 1$ to $N_{bankPerGroup}$ **do**
14: Change BankSelect
15: Read Weights to FMA Tree
16: $o_{Ch} = N_{O_{ch}} * OC_{Split}$
17: **for** $p \leftarrow 1$ to $InputWidth$ **do**
18: $I_{grad}[b, o_{ch}, row, p] += \sum O_{grad}[b, o_{ch} : +24, r, p] * W[o_{ch} : +n_{oc}, i_{ch} : +24, kh, :]$
19: **end for**
20: **end for**
21: **end for**
22: **end for**
23: Store I_{grad} Row to Ext. Memory
24: **end for**
25: **end for**
26: **end for**

Similarly, the feedbackward stage of convolutional layers (conv-bp), required for the training stage of CNNs, could be realized in our processor using only output data point routing. As illustrated in Fig. 17, convolutional backward shares spatial correlations between the output data, as conv-bp is a transpose of conv-ff. The accumulated value of the second FMA tree in Fig. 17(a) is time-delayed before being added in the final accumulator of the first FMA tree in Fig. 17(b), making use of spatial locality and avoiding redundant computations. During the computation of conv-bp steps, additional adders are not required for addition between a time-delayed accumulated value and the current accumulated value; the accumulation adders required during conv-ff could be utilized as shown in Fig. 18 to provide the functionality required by conv-bp.

The pseudo-code for the computational flow in conv-bp is shown in Algorithm 2. Similar to the convolutional feedforward stage, input channels are split by the maximum number of input channels that could be loaded to the on-chip memory, iterating over N_{ICSplit} times for maximum data reuse. Note that the annotation "I_{grad}" represents the actual output of the conv-bp function, and "O_{grad}" represents the actual input. Efficient implementation of the conv-bp functionality may be of interest even for CNN accelerators that only target inference, as conv-bp is mathematically equivalent to transposed convolution (deconvolution) which are often implemented in image-to-image networks or generative networks.

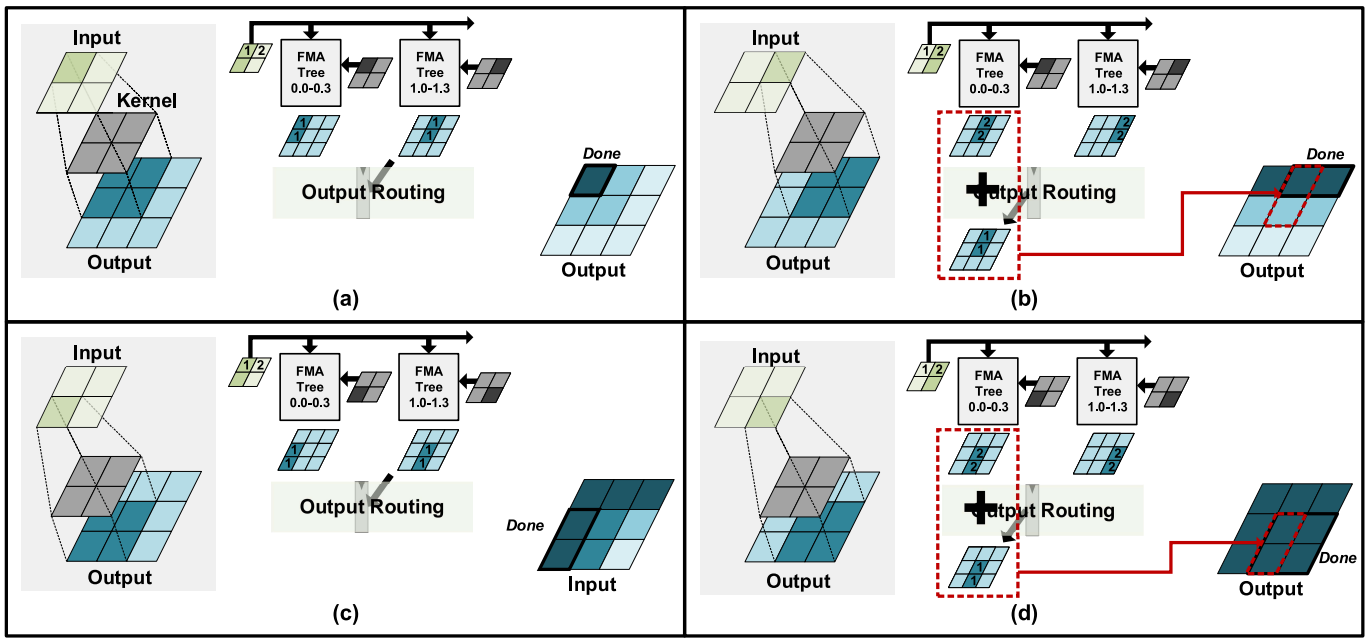

Fig. 17. Illustration of conv-feedbackwad functionality. (a) First step of conv-bp in the case of KW = 2. (b) In the next processing cycle, the computed value in the second FMA tree of step (a) is added to current accumulated value in the first FMA tree, making use of spatial correlation. (c) and (d) Next rows are processed likewise to maximize data re-use.

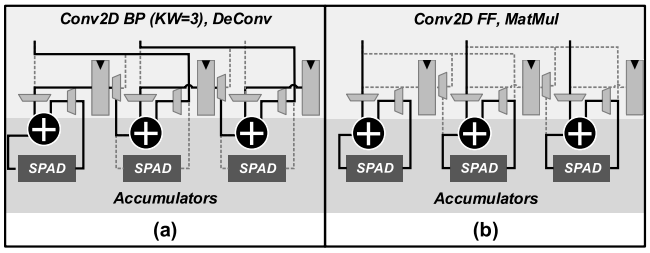

Fig. 18. Illustration of the output routing unit. (a) Output routing unit in the configuration of KW = 3 in conv-bp step. (b) Output routing unit is shown during conv-ff operations.

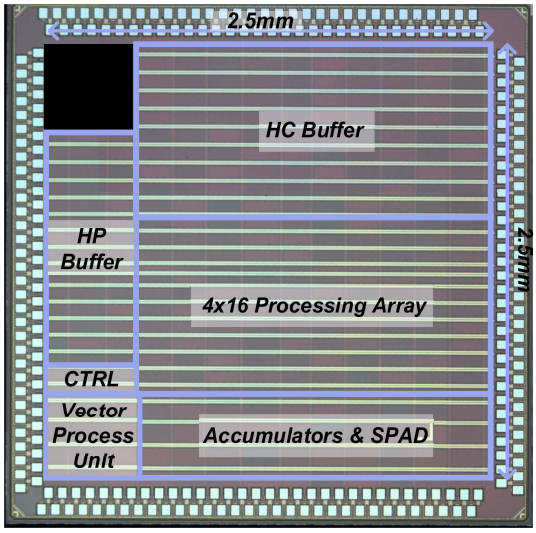

Fig. 19. Micrograph of the fabricated processor.

IV. Results and Discussion

We verify the designed processor by fabricating the design in a 40-nm LP CMOS technology. A micrograph of the 2.5 mm × 2.5 mm core with 293 kB of on-chip SRAM and 4 × 16 24-way FMA trees is shown in Fig. 19. We have also built a test environment with an FPGA integrated with 1-GB DDR3 DRAM to verify the functionality, performance, and power consumption of the fabricated processor. This test environment is shown in Fig. 20.

We verify the merits of the processor using two different measurement points: one for the maximum performance, using 1.1-V core voltage with 180-MHz clock, and another for the maximum efficiency, using 0.75-V core voltage with 20-MHz clock. We first measure the energy efficiency–performance tradeoff across different core voltages, as shown in Fig. 21. Our processor shows a maximum performance of 567 GFLOPS at 1.1 V, 180 MHz, and shows the maximum energy efficiency of 4.81 TFLOPS/W at 0.75 V, 20 MHz. We then measure our processor on three different benchmark models: ResNet-18 for image classification [25], one-layer LSTM for character prediction [26], and dc-GAN network for image generation [27]. The results for these benchmark models are summarized in Table III.

Inference operations are generally more energy-efficient, due to two main reasons: 1) there are additional memory-bound operations required during training, such as weight updates and keeping track of weight momentums and 2) there is more room for optimization during inference, such as merging batch-normalization layers to the convolutional layers, whereas this is not possible during training. LSTM networks show lower energy efficiency due to the operations

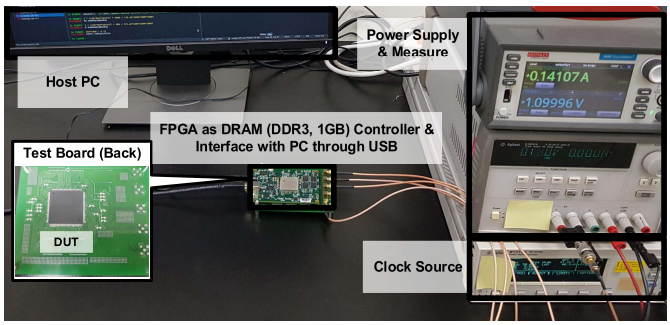

Fig. 20. Our verification system setup.

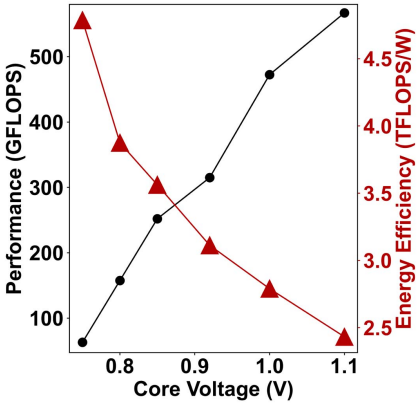

Fig. 21. Performance–efficiency trade-off graph.

TABLE III
PERFORMANCE OF THE PROCESSOR ON BENCHMARK MODELS

Model	ResNet-18		DC-GAN		LSTM	
Config	224×224 Image		512 to 32×32 Image		128×128 1-layer	
Train/Infer	Train	Infer	Train	Infer	Train	Infer
Max Energy Efficiency (TFLOPS/W)	1.64	2.05	1.63	2.05	0.31	0.34
Throughput (FPS)	27.2	92.1	220.5	766.0	82K	264K
EMA (bytes)	26.1M	5.2M	3.1M	0.6M	89.1K	39.9K
MACs	5.46G	1.82G	0.63G	0.21G	0.32M	0.13M

mainly being bottlenecked at memory accesses. The measurement and analysis of the EMA and the computational requirement, represented in MACs, are also shown at the bottom of Table III, which demonstrate that the ratio of EMA versus MACs is higher in training compared against inference phases.

To validate the final efficiency and performance of our processing system, we compare our work against prior works [1]–[5] in Table IV. At 0% sparsity, our processor shows 1.64 TFLOPS/W energy efficiency, outperforming prior work [1] using the same ResNet-18 training configuration by 2.48×. The training curve for ImageNet classification with ResNet-18 is shown in Fig. 22.

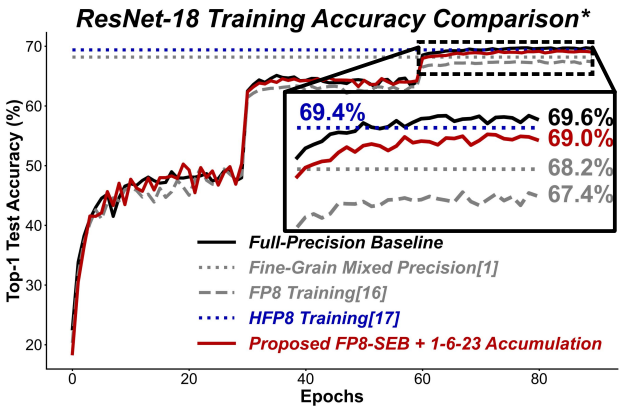

Fig. 22. Training ResNet-18 on ImageNet classification task from scratch using different low-precision training methods [1], [16], [17].

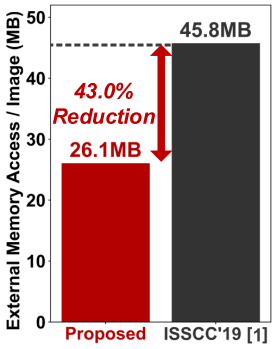

Fig. 23. EMA comparison.

We re-implement the 8-bit training proposed in [16] and use the reported figures from [1] and [17]. Compared against [1], our proposed numeric pipeline performs better even though it uses 8-bit floating contrary to using mixed precision of FP8 and FP16. At top-1 accuracy of 69.0%, our results are comparable to the training performance accomplished by hybrid-FP8 (HFP8) proposed in [17] (69.39%), which is effectively implemented as FP9 [5]. Note that we used GPU-based simulation to extract training accuracy up to 90 epochs (= 90 M images) in a reasonable amount of time, and the fabricated chip was confirmed to exactly match the simulation model in measurements.

In order to further evaluate the costs and effectiveness of different FP8 formats, we implemented combinational adders, multipliers, and MAC units for 1-4-3 (our format), 1-5-2 [16], and hybrid-FP8 (implemented as FP9 with 1-5-3) [5], [17] in Table V.

Including the cost of memory access provides more extensive information on the actual performance of the processor. Despite having smaller on-chip memory size compared against other designs in Table IV, due to the benefit of FP8-SEB tensor format and optimized datapath control for reduced memory access, our processor requires 43.0% less memory access per

TABLE IV
COMPARISON OF NEURAL NETWORK TRAINING PROCESSORS

	Our Work	[1]	[2]	[3]	[4]	[5]
Technology	**40nm**	65nm	65nm	40nm	14nm	7nm
Training	**CNN/DNN/RNN**	CNN/DNN/RNN	DNN(GAN)	CNN/DNN/RNN	CNN/DNN/RNN	CNN/DNN/RNN
Data Format	**FP8-SEB**	FP8/FP16	FP8/FP16	BFLOAT16	FP16/FP32	HFP8
Peak Performance	**567GFLOPS @180MHz, 1.1V**	300-600GFLOPS	540-1080GFLOPS	204GFLOPS	3000GFLOPS	25600GFLOPS
Max Energy Efficiency (@sparsity=0%)	**4.81TFLOPS/W @20MHz, 0.75V**	1.74-3.48TFLOPS/W	1.81-3.62TFLOPS/W	2.16TFLOPS/W	1.41TFLOPS/W	3.5TFLOPS/W
Real Model Training Efficiency	**1.64TFLOPS/W (ResNet-18)**	0.66TFLOPS/W (@sparsity=0%) 0.87TFLOPS/W (@sparsity=43%) (ResNet-18)	0.57TFLOPS/W (@sparsity=0%) 1.00TFLOPS/W (@sparsity=41%) (CycleGAN)	N/A	N/A	N/A
On-Chip Memory	**293kB**	372kB	676kB	448kB	2MB	8MB
Core Area	**6.25mm^2**	16mm^2	32.4mm^2	16mm^2	9.24mm^2	19.6mm^2

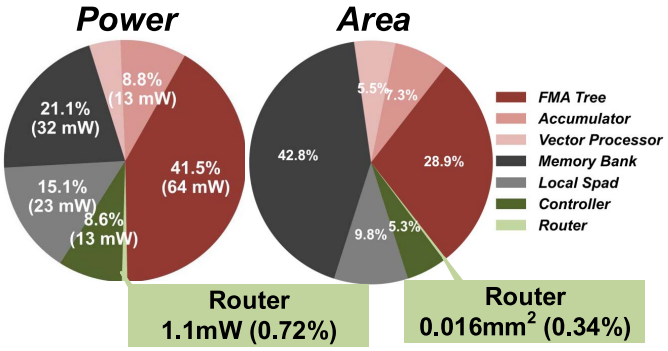

Fig. 24. Area and power breakdown simulated for ResNet-18 conv-ff layer with input channel = 256, output channel = 256, kernel sizes = (3,3), and feature size = (14,14).

TABLE V
ADDERS, MULTIPLIERS, AND MAC UNITS WITH VARIOUS FP8 CONFIGURATIONS*

Logic Type		**FP8(1.4.3)**	**FP8(1.5.2)**	**HFP8**
Adder	Area(μm^2)	273.773	259.426	298.469
	Energy(pJ)	0.65	0.57	0.67
Multiplier	Area(μm^2)	408.307	354.92	455.35
	Energy(pJ)	0.755	0.61	0.84
MAC	Area(μm^2)	675.494	584.237	735.706
	Energy(pJ)	1.81	1.42	1.87

*Synthesis results in 40nm LP CMOS technology.

image of ResNet-18 training compared against the design in [1] (see Fig. 23). This was measured under the same DRAM settings for a fair comparison, constraining our processor to only use 256 MB of the available 1 GB of memory.

Compared against NVIDIA Titan X GPU, when training ResNet-18 on 64 batches of 224 × 224 images, our processor achieves 1.64 TFLOPS/W computing efficiency, improving by 78.1× from GPU efficiency of 0.021 TFLOPS/W. DRAM usage is also reduced by 81.6%, requiring a mere 573.9 MB of memory compared against 3127.0 MB required by GPUs.

The breakdown of area and power consumption using the final place-and-routed netlist is shown in Fig. 24. The routing units required on our processor only takes up 0.72% of power and 0.34% of the area. The total costs in computation take up 55.3% (85.85 mW) of power and memory-related costs take up 44.7% (69.50 mW) of total power.

V. CONCLUSION

The environment shifts in DNN models incur new challenges that DNN processors face today—higher standards for numerical precision as models become more sensitive to errors in difficult tasks, a departure from ReLU activations that provided ample sparsity, and new training paradigms that require on-device gradient calculations and training. Our work contributes to overcoming these challenges through co-optimization of hardware and the numerical representation algorithms; an FP8-SEB data format is introduced along with its efficient implementation in N-way FMA trees, showing higher numerical precision and lower hardware costs compared to implementation in conventional MAC. Coupled with routing in only input–output datapoints, data re-use is maximized in both off-chip and on-chip memory reads, further increasing the energy efficiency of our processor architecture.

Fabricated in 40-nm LP CMOS, the processor consumes 13.1 mW at 0.75 V, 20 MHz with the maximum energy efficiency of 4.81 TFLOPS/W, and 230 mW at 1.1 V, 180 MHz with the maximum performance of 567 GFLOPS and area efficiency of 90.7 GFLOPS/mm^2. The fabricated processor is measured to consume 43.0% less EMA compared to prior work for the same ResNet-18 training on ImageNet due to using only 8-bit representation for communicating with

external DRAMs. In addition, the processor provides 2.48× higher energy efficiency for training models with non-sparse activations by using the proposed FP8-SEB format combined with its efficient implementation of 24-way FMA trees, high-precision accumulators, and flexible 2-D routing.

REFERENCES

[1] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-J. Yoo, "7.7 LNPU: A 25.3TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2019, pp. 142–144.

[2] S. Kang *et al.*, "7.4 GANPU: A 135TFLOPS/W multi-DNN training processor for GANs with speculative dual-sparsity exploitation," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2020, pp. 140–142.

[3] C. Kim, S. Kang, D. Shin, S. Choi, Y. Kim, and H.-J. Yoo, "A 2.1TFLOPS/W mobile deep RL accelerator with transposable PE array and experience compression," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2019, pp. 136–138.

[4] B. Fleischer *et al.*, "A scalable multi- TeraOPS deep learning processor core for AI trainina and inference," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2018, pp. 35–36.

[5] A. Agrawal *et al.*, "A 7 nm 4-core AI chip with 25.6TFLOPS hybrid FP8 training, 102.4TOPS INT4 inference and workload-aware throttling," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, vol. 64, Feb. 2021, pp. 144–146.

[6] 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.

[7] J. Song *et al.*, "7.1 An 11.5TOPS/W 1024-MAC butterfly structure dual-core sparsity-aware neural processing unit in 8 nm flagship mobile SoC," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2019, pp. 130–132.

[8] S. Kim, J. Lee, S. Kang, J. Lee, and H.-J. Yoo, "A 146.52 TOPS/W deep-neural-network learning processor with stochastic coarse-fine pruning and adaptive input/output/weight skipping," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2020, pp. 1–2.

[9] J. Oh *et al.*, "A 3.0 TFLOPS 0.62V scalable processor core for high compute utilization AI training and inference," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2020, pp. 1–2.

[10] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.

[11] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2017, *arXiv:1710.05941*. [Online]. Available: http://arxiv.org/abs/1710.05941

[12] C. Xie, M. Tan, B. Gong, A. Yuille, and Q. V. Le, "Smooth adversarial training," 2020, *arXiv:2006.14536*. [Online]. Available: http://arxiv.org/abs/2006.14536

[13] J. Konečný, H. Brendan McMahan, F. X. Yu, P. Richtárik, A. Theertha Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," 2016, *arXiv:1610.05492*. [Online]. Available: http://arxiv.org/abs/1610.05492

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

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

[16] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 1–10.

[17] X. Sun *et al.*, "Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–10.

[18] S. Fox, S. Rasoulinezhad, J. Faraone, D. boland, and P. Leong, "A block minifloat representation for training deep neural networks," in *Proc. Int. Conf. Learn. Represent.*, 2021, pp. 1–16. [Online]. Available: https://openreview.net/forum?id=6zaTwpNSsQ2

[19] 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*. [Online]. Available: http://arxiv.org/abs/1606.06160

[20] U. Köster *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," 2017, *arXiv:1711.02213*. [Online]. Available: http://arxiv.org/abs/1711.02213

[21] P. Micikevicius *et al.*, "Mixed precision training," 2017, *arXiv:1710.03740*. [Online]. Available: http://arxiv.org/abs/1710.03740

[22] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2014, pp. 10–14.

[23] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 1–41, May 2008.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 25, 2012, pp. 1097–1105.

[25] 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.

[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[27] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2015, *arXiv:1511.06434*. [Online]. Available: http://arxiv.org/abs/1511.06434

Jeongwoo Park (Graduate Student Member, IEEE) received the B.S. degree from the Department of Electrical and Computer Engineering, Seoul National University, Seoul, South Korea, in 2017, where he is currently pursuing the Ph.D. degree.

His research interests include neuromorphic algorithms and systems, quantized neural network training, and efficient application-specific integrated circuit (ASIC) design for deep learning inference/training.

Sunwoo Lee (Graduate Student Member, IEEE) received the B.S. degree from the Department of Electrical and Computer Engineering, Seoul National University, Seoul, South Korea, in 2020, where he is currently pursuing the Ph.D. degree.

His research interests include quantized neural networks, low-precision neural network training algorithms and systems, and low-power application-specific integrated circuit (ASIC) design for deep learning inference/training.

Dongsuk Jeon (Member, IEEE) received the B.S. degree in electrical engineering from Seoul National University, Seoul, South Korea, in 2009, and the Ph.D. degree in electrical engineering from the University of Michigan at Ann Arbor, Ann Arbor, MI, USA, in 2014.

From 2014 to 2015, he was a Post-Doctoral Associate with the Massachusetts Institute of Technology, Cambridge, MA, USA. He is currently an Assistant Professor with the Graduate School of Convergence Science and Technology, Seoul National University. His current research interests include hardware-oriented machine learning algorithms, hardware accelerators, and low-power circuits.