A 28-nm 8-bit Floating-Point Tensor Core-Based Programmable CNN Training Processor With Dynamic Structured Sparsity

Shreyas Kolala Venkataramanaiah, *Member, IEEE*, Jian Meng, *Student Member, IEEE*, Han-Sok Suh, *Student Member, IEEE*, Injune Yeo, *Member, IEEE*, Jyotishman Saikia, *Student Member, IEEE*, Sai Kiran Cherupally, *Member, IEEE*, Yichi Zhang, *Student Member, IEEE*, Zhiru Zhang, *Fellow, IEEE*, and Jae-Sun Seo, *Senior Member, IEEE*

***Abstract*— Training deep/convolutional neural networks (DNNs/CNNs) requires a large amount of memory and iterative computation, which necessitates speedup and energy reduction, especially for edge devices with resource/energy constraints. In this work, we present an 8-bit floating-point (FP8) training the processor which implements: 1) highly parallel tensor cores (fused multiply–add trees) that maintain high utilization throughout forward propagation (FP), backward propagation (BP), and weight update (WU) phases of the training process; 2) hardware-efficient channel gating for dynamic output activation sparsity; 3) dynamic weight sparsity (WS) based on group Lasso; and 4) gradient skipping based on the FP prediction error. We develop a custom instruction set architecture (ISA) to flexibly support different CNN topologies and training parameters. The 28-nm prototype chip demonstrates large improvements in floating-point operations (FLOPs) reduction (7.3×), energy efficiency (16.4 TFLOPS/W), and overall training latency speedup (4.7×), for both supervised and self-supervised training tasks.**

***Index Terms*— Activation sparsity, CNNs, convolutional neural network (CNN) training processor, energy-efficient accelerator, weight sparsity (WS).**

I. Introduction

ARTIFICIAL intelligence technology has been adopted in many applications such as autonomous driving, smart drones, face recognition, natural language processing, and so on. Training deep/convolutional neural networks (DNNs/CNNs) for these applications involves a very large amount of memory accesses and computations and has been typically performed in high-performance cloud servers with graphics processing unit (GPUs). Once DNN/CNN models are trained on the cloud server, such models are downloaded on edge devices to perform inference. But what if we want to fine-tune the models with user-specific data? Then user-specific or personal data would have to be uploaded to the cloud, which raises privacy and security concerns. In addition, the updated model would have to be communicated again to the edge devices, adding considerable latency. To avoid this, if we can train the model on edge devices, then we obtain enhanced personalization and the data would be secure, as it never leaves the device. However, training the complex DNN/CNN models on resource-/energy-constrained platforms is challenging, and this requires an energy-efficient hardware design for the CNN training processor.

A number of prior works reported DNN/CNN training processor designs [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]. In [4], a DNN training processor with 8-bit floating-point (FP8) precision and shared exponent bias was reported, targeting nonsparse neural networks. A programmable CNN learning processor is proposed in [5] that utilizes off-the-shelf static random access memory (SRAMs) to perform nontranspose and transpose operations in 16-bit fixed-point precision. However, the proposed SRAM demands complex read/write control and incurs area overhead compared to conventional SRAMs. Bit-slice input–output sparsity is exploited in [6], but did not consider weight sparsity (WS) during training, and variable fixed-point precision was used, which typically shows lower accuracy than floating-point counterparts. In [7], the sparse channels are randomly selected, and the hardware design is simplified without sparsity similarity comparison, resulting in training accuracy loss. A global magnitude threshold is used in [8] to generate the element-wise sparse masks without sorting, but the nonstructured sparse elements are not skippable. An input load balancer is proposed in [9] to support irregular sparsity but did not consider low utilization of PEs during small kernel weight gradient computation.

Fu et al. [12] presented a filtering scheme to achieve structured/unstructured sparsity and compress the sparse activation

Manuscript received 9 January 2023; revised 14 April 2023; accepted 17 April 2023. Date of publication 15 May 2023; date of current version 28 June 2023. This article was approved by Associate Editor Massimo Alioto. This work was supported in part by the National Science Foundation (NSF) under Grant 1652866; and in part by the Center of Brain-Inspired Computing (C-BRIC) in JUMP 1.0 and the Center for the Co-Design of Cognitive Systems (CoCoSys) in JUMP 2.0, Semiconductor Research Corporation (SRC) programs sponsored by the Defense Advanced Research Projects Agency (DARPA). *(Corresponding author: Jae-Sun Seo.)*

Shreyas Kolala Venkataramanaiah was with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281 USA. He is now with Apple, Cupertino, CA 95014 USA.

Jian Meng, Han-Sok Suh, Jyotishman Saikia, and Jae-Sun Seo are with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281 USA (e-mail: jaesun.seo@asu.edu).

Injune Yeo was with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281 USA. He is now with the Department of Electrical Engineering, Chosun University, Gwangju 61452, South Korea.

Sai Kiran Cherupally was with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ 85281 USA. He is now with TI Kilby Labs, Dallas, TX 75243 USA.

Yichi Zhang and Zhiru Zhang are with the School of Electrical and Computer Engineering, Cornell University, Ithaca, NY 14850 USA.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/JSSC.2023.3269148.

Digital Object Identifier 10.1109/JSSC.2023.3269148

data for DNN training, but they did not generate/exploit WS. Tu et al. [13] propose on-device quantization voltage frequency (QVF) tuning to improve the performance and energy efficiency of the deep-learning processor. In addition, block-floating-point precision was employed in [12] where the actual computation occurs in fixed-point precision and integer precision was employed in [13], which generally limits the achievable training accuracy compared to floating-point precision. TSUNAMI [14] introduces dynamic pruning to create sparsity in the weights and activations of CNNs during training. However, the activation pruning unit they propose involves processing all activations through a multilayer perceptron (MLP) unit, which incurs hardware/computation overhead. SRAM-based processing-in-memory architecture has been presented for on-device DNN training [15], but employs custom 8T SRAM which incurs area overhead compared to off-the-shelf foundry SRAMs. Denser RRAMs have been employed in [16] for AI training, but RRAM is not available in the latest CMOS technologies and exhibits limitations in endurance, on–off ratio, reliability, and so on compared to SRAM counterparts. IBM presented a series of AI processors that can support both training and inference workloads with different precision values [17], [18], [19], [20], but these works did not exploit sparsity during training.

In this work, we present a new sparse CNN training accelerator [21] with FP8 precision that exploits structured activation sparsity, structured WS, and gradient skipping to dynamically reduce the unimportant operations and achieves high hardware speedup. Hardware-efficient sparse training algorithms have been developed corresponding to the custom sparse training accelerator with programmable instructions. The sparse training accelerator has been evaluated for six different CNN models from <0.3 to >11 M parameters and for both supervised training and self-supervised training tasks. The 28-nm prototype chip demonstrates large improvements in floating-point operations (FLOPs) reduction (7.3×), energy efficiency (16.4 TFLOPS/W), and overall training latency speedup (4.7×).

II. Hardware-Efficient Sparse Training Algorithm

We present a novel and comprehensive hardware-aware efficient training algorithm (see Fig. 1) that is highly compatible with the proposed hardware training accelerator. Different from the prior works that solely considered WS or activation sparsity, the proposed algorithm simultaneously exploits weight, activation, and gradient sparsity in all phases of the training process.

A. Stochastic Gradient Descent (SGD)

SGD [22] minimizes the local loss with respect to different mini-batches. Instead of optimizing the model based on the entire dataset, deploying data to the training hardware with dedicated batch sizes leads to high memory efficiency and comparable accuracy. In general, each iteration of the SGD-based batch-by-batch training can be divided into *Forward Propagation* (FP), *Backward Propagation* (BP), and *Weight Update* (WU) phases. In the FP phase, the output activations

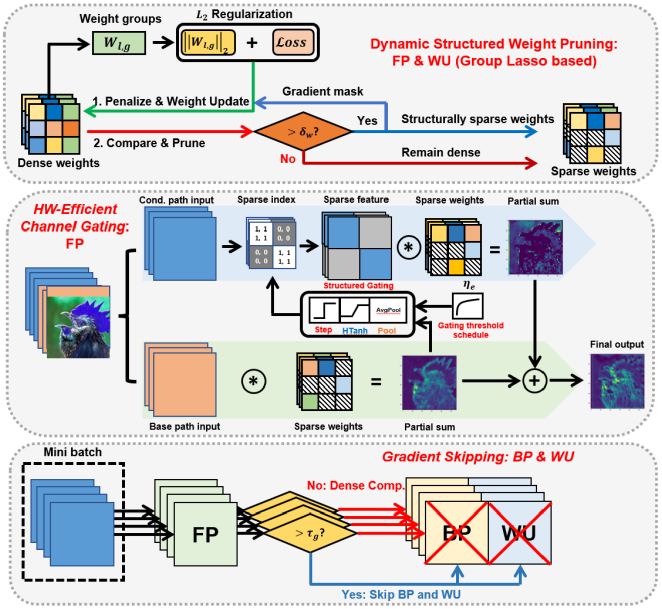


Fig. 1. Proposed efficient sparse training techniques.

are computed sequentially in a layer-by-layer manner, while the training loss is computed based on the distance between the final output logits and the target label. In the BP phase, the local gradients are computed at every layer in the backward direction. During BP, the incoming gradients are convolved with the transposed kernels, and the pooling (downsampling) operations are replaced by upsampling units. In the WU phase, weight gradients are computed with respect to all the components of the loss function, including both vanilla loss (e.g., cross-entropy and mse) and regularization penalty term. Based on the local gradient of the current mini-batch, the weights are updated to minimize the loss.

B. Regularization-Based Weight Sparsification (WS)

Prior works have shown that DNNs can still retain performance even if many network weights are removed [23], [24], [25], [26]. Pruning the neural network entails generating a sparse model $f(W \odot M)$, where $M \in \{0, 1\}$ is the binary mask that forces a certain amount of "unimportant" weights to zero and W is the weight of the DNN model.

The granularity of the sparsity has a significant impact on the efficiency of the hardware computation. Unstructured pruning generates high element-wise sparsity with the cost of a large amount of sparse index storage. Subsequently, fine-grained redundancy skipping requires frequent memory access, leading to high energy consumption and increased computation latency. Such performance overhead will be amplified during the on-device training due to the iterative and intensive computation for both forward and backward computation. On the other hand, structured pruning exploits the WS in a group-wise fashion. The coarse-grained sparsity enables the hardware to skip computation with respect to the size of the processing element (PE), elevating the energy efficiency and accelerating the training.

Evaluating the redundancy of the weight groups has been widely investigated in prior works [23], [24]. Score-based sparsification prunes the weights based on the deterministic or relative L_1 norm threshold. The recently proposed sparse training algorithm removes the unimportant weight channels and feature map pixels based on the Top-K L_1 norm weight scores [27] or gradient scores [28]. However, the popular Top-K selection requires complex sorting and percentile computation, especially for element-wise feature map sparsification. On the hardware side, the "sorting-and-pruning" algorithm would introduce a massive amount of comparators for the accelerator design, which is computationally expensive for resource-constrained hardware.

Unlike the score-based redundancy evaluation, this work formulates the structured in-training WS based on the regularization-aided weight penalty. Specifically, the regularization process penalizes the weights with the small magnitude (L_1 or L_2 norm) without using the magnitude sorting. Furthermore, the weight penalty can be asserted in both a structured and unstructured manner. Group Lasso [23] penalizes the weight groups based on the group-wise L_2-regularization during DNN training. The loss function can be formulated as

$$\hat{\mathcal{L}} = \mathcal{L}\left(f\left(x, \{W\}_{l=1}^{L}\right), y\right) + \lambda \sum_{l=1}^{L} \sum_{g=1}^{G_l} \left\| W_{l,g} \right\|_2 \tag{1}$$

where $\mathcal{L}$ represents the loss of the DNN model f respect to the ground-truth y, and G_l represents the number of predefined groups in layer l. The intensity of the structured pruning is controlled by the tunable parameter λ. Penalizing the weight groups $W_{l,g}$ reduces the weight magnitude but generating the hardware-skippable group sparsity requires the subsequent sparsification operation to set the penalized weight values to "0." In this work, we sparsify the weights by comparing the weight elements with the predefined threshold value δ, resulting in the sparsification mask M. The structurally sparse masks are applied to both FP convolution and backward weight gradient computation during training. Mathematically, we have

$$\text{Forward convolution: } Y = X * W \odot M \tag{2}$$

$$\text{Weight gradient: } \nabla W = \nabla W \odot M. \tag{3}$$

Specifically, for each individual weight group $W_{l,g}$, adding the regularization terms to the loss function introduces the extra penalty in the gradient computation. For an individual weight element $w_{i,g} \in W_{l,g}$, the gradient $\nabla w_{i,g}$ is

$$\nabla w_{i,g} = \frac{\partial \hat{\mathcal{L}}}{\partial \nabla w_{i,g}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \nabla w_{i,g}}}_{\text{Normal gradient}} + \underbrace{\lambda \frac{\partial \left\| W_{l,g} \right\|_2}{\partial \nabla w_{i,g}}}_{\text{Group Lasso penalty}}. \tag{4}$$

The weight gradient with respect to the loss $\mathcal{L}$ can be computed directly in the normal fashion, while the group Lasso penalty can be expanded as

$$\frac{\partial \left\| W_{l,g} \right\|_2}{\partial \nabla w_{i,g}} = \frac{\partial}{\partial \nabla w_{i,g}} \sqrt{\sum_i w_{i,g}^2} = \frac{1}{\sum_i w_{i,g}^2} \cdot w_{i,g}. \tag{5}$$

Combining (4) and (5), the WU process can be formulated with the learning rate η as

$$w_{i,g}^* = w_{i,g} - \eta \left(\frac{\partial \mathcal{L}}{\partial \nabla w_{i,g}} + \frac{\lambda}{\sum_i w_{i,g}^2} \cdot w_{i,g} \right) \tag{6}$$

$$= \underbrace{\left(1 - \eta \frac{\lambda}{\left\| W_{l,g} \right\|_2} \right)}_{\text{GL}_{\text{scale}}} w_{i,g} - \underbrace{\eta \frac{\partial \mathcal{L}}{\partial \nabla w_{i,g}}}_{\text{Normal gradient}}. \tag{7}$$

In addition to the normal gradient computation, the old weight value $w_{i,g}$ will be scaled down by the decay factor, namely GL_{scale}. Since the L_2 norm of the weight group $W_{l,g}$ is constant at each iteration, the scaling is mainly controlled by the magnitude of $w_{i,g}$. In other words, the small magnitude weights tend to be gently scaled down, while the large weight values will be penalized harder. As a result, the overall magnitude of $W_{l,g}$ decreases during training, and the element-wise threshold can easily force the entire group to zero. The simplicity of the designed group Lasso pruning algorithm creates high feasibility in the hardware design, leading to superior computation efficiency.

C. Structured Channel Gating

In addition to the weight sparsification, our proposed design also exploits the activation sparsity by skipping the unsalient features during the forward pass of training. Some prior works employed an auxiliary neural network to predict the uninformative features or channels [29], [30], [31], [32], which is equivalent to projecting the high-dimensional input, to the latent feature space. The resultant low-dimensional salience vectors will be used to formulate the binary feature masks during supervised training. However, during the FP phase of the training, the convolution operation has to wait until the salience prediction is complete. Implementing the additional neural networks (e.g., CNN) retards the on-device training with the increased FP phase latency and complicated backward computation.

Motivated by CGNet [33], this work proposes structured channel gating (SCGNet), which divides the total convolution into "base" and "conditional" paths. The base path computation involves the fully dense input feature map channels and the corresponding weights. The resultant partial sum will be gated for the computation skipping map of the conditional path. The high correlation between convolution channels implies the commutative property between the base path and conditional path. Since the feature salience prediction is part of the convolution operation, the latency and energy overhead are minimal.

On the hardware side, embedding the nonlinear Sigmoid gating function and the learnable gating threshold of the CGNet could be expensive for the hardware designation. In this work, we mimic the nonlinearity of the Sigmoid gating function with the scaled and shifted HardTanh

$$\mathcal{G}\left(y_b^*\right) = \underset{\text{min}=0,\text{max}=1}{\text{HardTanh}} \left(0.25 \times \left(y_b^* + 2\right) - 0.5\right) \tag{8}$$

where y_b^* represents the normalized and shifted base path output, and the offset of the shifting is the learnable gating

TABLE I

HIGH HARDWARE COMPATIBILITY AND NEGLIGIBLE ACCURACY DROP OF THE SCGNET WITH RESNET-18 MODEL ON CIFAR-10

Method	Gating Func.	Threshold	Cond. Path Sparsity	Granularity	Acc.
Baseline	-	-	0.0%	-	94.78%
CGNet [33]	Sigmoid	Learnable	70.21%	Element-wise	94.45%
SCGNet	**HardTanh**	**Scheduled**	68.71%	Structured	94.41%

threshold τ of the CGNet, which can further control the conditional path sparsity

$$y_b^* = \text{normalize}(y_b - \tau). \tag{9}$$

In practice, we found out that the magnitude of τ is gradually ramping throughout the training, which can be modeled as an epoch-based deterministic schedule function

$$\tau_e = \tau_{\text{final}} - \left(\frac{e_{\text{total}} - 1 - e_{\text{current}}}{e_{\text{total}}}\right)^p (\tau_{\text{final}} - \tau_{\text{initial}}). \tag{10}$$

We set the initial threshold initial to zero and keep the final threshold final and the ramping power p as the tunable parameters. The deterministic threshold schedule is employed as the lookup table for each epoch of the training. Combining (8)–(10), the binary feature salience mask can be computed as

$$M_c = \mathcal{H}\left(\mathcal{G}\left(y_b^*\right)\right) \in \{0, 1\} \tag{11}$$

$$y_b^* = \text{normalize}(y_b - \tau_e) \tag{12}$$

where $\mathcal{H}$ is the Heaviside step function for the binary mask computing.

To avoid the expensive fine-grained sparse indexes of the element-wise sparsity, we generate structured sparsity of feature maps by computing the *group salience* scores of the base path output features y_b^* via 3-D average pooling and HardTanh (see Fig. 1). The group salience scores determine the structured computation skipping of the conditional path. Table I shows the negligible accuracy degradation of the highly hardware-compatible SCGNet.

D. Confidence-Based Gradient Computation Skipping (GS)

Under the context of supervised learning with the classification tasks, the output logits of FP represents the confidence with respect to each class. Since the convergence time and sample complexity varies between different inputs, the softmax score of the logits indicates the prediction confidence of the given sample, which can be utilized to justify whether the image should be learned again.

Specifically, inputs with high confidence scores during FP will have minimal WU in the training process and thus can be skipped from the BP and WU phases [34]. As shown in Fig. 1, we exclude inputs with high softmax confidence from the BP and WU phases. Combined with the structured weight/gradient sparsity, the proposed scheme achieves high energy efficiency in both gradient accumulation and the gradient itself.

TABLE II

DETAILED DATA PRECISION OF THE PROPOSED FP8 PRECISION TRAINING SCHEME

Variables	Exp. Bits	Man. Bits
FP		
Weight	5	2
Activation	5	2
Conv. Accumulation	5	10
Loss	5	2
Learning Rate	5	2
SCG Threshold	5	2
Grad. Skipping Threshold	5	2
BP & WU		
Gradient Accumulation	5	10
L2 Norm Accumulation	5	10
L2 Norm	5	2
Weight Gradient	5	2

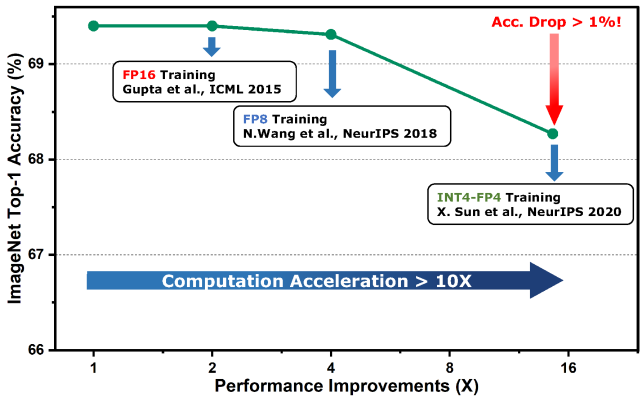


Fig. 2. Continuous progress in low-precision FP training. FP8 precision provides minimal accuracy drop with sizable computation reduction/acceleration.

E. Low-Precision Training

For the entire training process, we use FP8 precision (FP8) to represent weight, activation, gradient, and other hyperparameters (e.g., learning rate and thresholds). The FP8 data is configured with a 1-bit sign, 5-bit exponent, and 2-bit mantissa (1-5-2) [35]. The 16-bit floating-point precision (FP16) has a (1-5-10) format and is employed to represent the intermediate accumulation results of convolution in both FP and BP phases. Table II summarizes the detailed data precision in FP, BP, and WU phases. Fig. 2 summarizes the performance of the recent SoTA low precision training algorithms, where the employed FP8 training [35] scheme balances the speedup-accuracy trade-off for DNN training.

F. Self-Supervised Learning

In addition to the supervised sparse learning algorithm, the proposed training accelerator also supports the recent contrastive self-supervised learning (SSL), achieving both high energy efficiency and low labor intensity. In this work, we adopt the SimCLR [36] as the basic SSL training method to exploit the hardware-aware weight and feature sparsity during training. Specifically, SimCLR [36] preprocesses the original input into two separate images with different augmentation techniques ($X \rightarrow X^*$). Two different versions of the same image are considered as positive pairs (X_i^*, X_j^+), while the

Algorithm 1 The Proposed SimSkip Algorithm for SSL

Require: Encoder f**, projector** g**, temperature of loss function** τ**, skipping threshold** γ**. Batch size** N

Initialize Fixed skipping threshold γ
for sampled minibatch X_i **do**
  Forward Pass:
  for contrastive branch $a_t \in \{1, n\}$ **do**
    Draw data augmentation $t_{a_i} \sim \mathcal{T}$
    $X_i^{a_t} = t_{a_t}(X_k)$
    Get the encoded output: $v_i = g(f(X_i))$
  end for
  Compute the similarity matrix S between latent vectors of two contrastive paths (effective batch size $= 2N$).
  $S^* = \textbf{Softmax}(S) \in \mathbb{R}^{(2N,2N-1)}$
  Skip the batch with positive pair similarity $> \gamma$.
  Compute the contrastive loss based on Eq. 13 with the **truncated** similarity matrix.
  Backward Pass: Skip the gradient computation with respect to the skipped contrastive samples.
  Weight Update: Skip the gradient averaging with respect to the skipped contrastive samples.
end for

rest of the mini-batch is considered as negative pairs (X_i^*, X_k^-). The augmented images are encoded separately with a shared DNN, leading to the latent vector v.

Compared to conventional supervised learning, SSL learns visual representation by increasing the similarity between the positive pairs while repelling the negative pairs. Mathematically, we have

$$\mathcal{L} = -\log \frac{\exp\left(\text{sim}\left(v_i, v_j^+\right)/\tau\right)}{\sum_{k=1}^{2N} \mathbf{1}_{k \neq i} \exp(\text{sim}(v_i, v_k)/\tau)} \tag{13}$$

where sim indicates the similarity between the encoded latent vectors and τ represents the temperature hyper-parameter.

Compared to other recent SSL algorithms [37], [38], SimCLR [36] utilizes a shared encoder for both augmented images, and the loss is computed collectively based on similarity. Therefore, the proposed group-Lasso-based weight pruning and gating-based feature skipping are compatible with SimCLR. As a result, it is feasible to deploy the shared encoder to the proposed accelerator while exploiting sparsity during training.

Since the deterministic labels are absent in SSL, the raw confidence becomes inaccessible in SSL. To enable gradient skipping in SSL, we propose SimSkip, a similarity-based training sample skipping algorithm designed with simplicity and high hardware compatibility. As introduced in [36], the *NT-Xent* loss of SimCLR [36] is computed based on the similarity. The high similarity between the positive pairs indicates the well-learned matches during training, which are safe to be skipped for the current training iteration. Algorithm 1 summarizes the proposed SimSkip method based on the SimCLR [36] with a shared encoder.

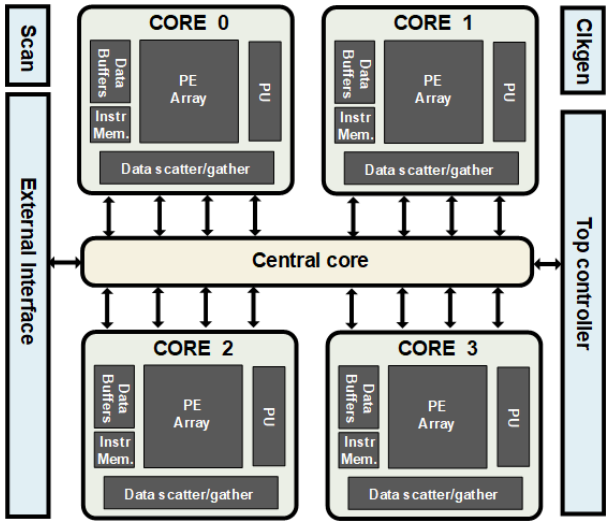


Fig. 3. Overall architecture of the CNN training processor.

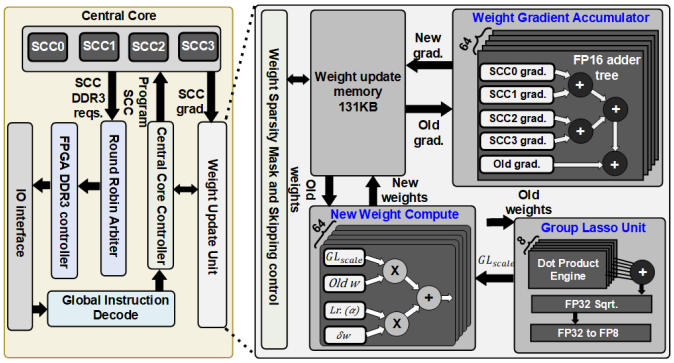


Fig. 4. Central core architecture with a detailed view of the WU unit.

III. Overall Architecture

Fig. 3 shows the overall architecture of the sparse training processor, which consists of four sparse compute cores (SpCC), a central core (CC), a global controller, and an external I/O interface to FPGA. Each SpCC is a programmable core that can be configured individually using the custom ISA.

A. Central Core

Fig. 4 depicts the top-level architecture of the CC, which coordinates all the SpCCs and processes off-chip DDR3 access requests. When the SpCC encounters a DDR3 memory access instruction, it decodes all the required parameters, such as the number of memory access, start address, and so on and sends a DDR3 access request to the CC. The CC processes the DDR3 access requests from the four SpCCs employing a round-robin arbiter. Once the DDR3 resource is granted, the central core sends an acknowledgment signal to the granted SpCC to process the incoming/outgoing data. The FPGA DDR3 controller divides the requested total memory accesses into smaller blocks that the IO interface can handle. It includes a state machine that monitors each small block transaction and initiates a new transaction when the prior transaction is complete.

The central core configures all SpCCs by loading instructions to their respective instruction memories. The instructions generated from the training compiler (more details in Section III-C) are sent to the central core through the chip IO interface. The central core processes the WU instructions (compute and memory) and SpCCs process all the other instructions. The global instruction decoder interprets the incoming instructions and dispatches them to one of the SpCCs or the central core.

Fig. 4 shows the detailed view of the WU unit. The WU unit: 1) performs the weight gradient (WG) accumulation; 2) generates the structured WS by adopting the group Lasso algorithm; and 3) performs the WU at the end of the batch based on stochastic gradient descent. The WGs are efficiently computed in the SpCCs and stored in their weight buffers. After all four SpCCs complete the WG computations, a gradient accumulation instruction is dispatched to the central core controller. The central core controller loads the WU memory with the previous WGs and reads all the computed WGs from the SpCCs. The WG accumulator obtains all the gradients (old WGs and WGs from all four SpCCs) and accumulates them using an FP16 adder tree, as shown in Fig. 4. The WU memory of the central core and weight buffers of the SpCC store 8×8 ($K \times C$) weights in each row. These 64 weight gradients are accumulated in parallel using the WG accumulator array.

The group Lasso unit (GLU) computes the inverse $L2$ norm of the weight group (8×8) and scales it with the $\mathrm{GL}_{\mathrm{scale}}$, as described in Section II-B. GLU reuses the dot product engines of the SpCC PE array. By sending the same inputs to the multipliers of the dot product engine, GLU produces a squared sum of 8 weights. The squared sum of the entire weight group is computed in parallel using an eight-dot product engine cascaded with an FP16 adder tree. Finally, an FP32 inverse square root module computes the $L2$ norm to generate a $\mathrm{GL}_{\mathrm{scale}}$.

The WS controller controls and schedules all the submodules in the WU unit. The controller generates the WS mask when the data (weights or weight gradients) is being written to the WU memory. The memory accesses and the computations of the sparse weight group are entirely skipped in the WU unit, thereby saving latency and power.

B. Sparse Compute Core

Fig. 5 shows the overall architecture of the SpCC. The SpCC includes: 1) a 16×8 PE array where each PE is a dot product engine with eight FP8 multipliers and one FP16 adder tree; 2) SRAMs to store activations, weights, instructions, and sparsity masks; 3) vector processing units for non-MAC operations; 4) sparsity controllers to exploit the structured sparsity of output activations/weights; and 5) data scatter/gather units to store the data in the buffers in the required format.

1) PE Array: As shown in Fig. 6, the PE array consists of eight PE columns, where each PE column has 16 PEs and a PE load balancer. Each PE is a configurable dot-product engine with eight FP8 (1-5-2) multipliers and one FP16 (1-5-10) adder tree. The PE column shares the weight vectors obtained from the weight register and the PE row shares the input activations during FP and BP. The dot product engine comprises

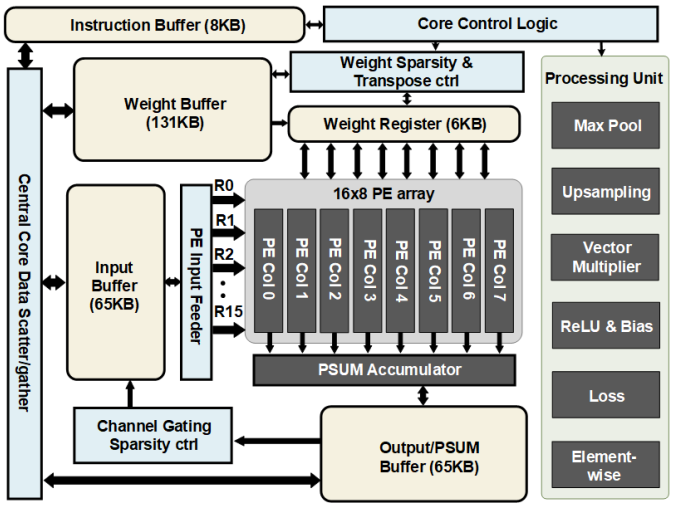


Fig. 5. Architecture of the sparse compute core (SpCC).

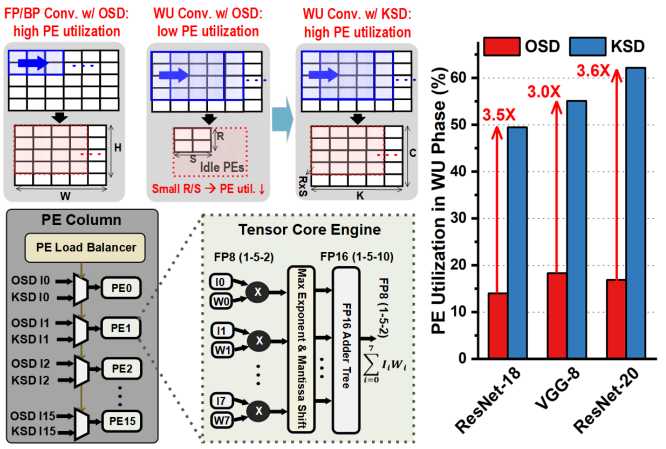


Fig. 6. PE array consisting of 16×8 PEs and a PE load balancer. The PE array follows OSD during FP/BP and KSD during WU, overall achieving high PE utilization.

eight FP8 multipliers and the products are aligned by matching the exponents and shifting the mantissa. The aligned mantissa products are sent to an 8-way FP16 adder tree. The output of the adder tree is normalized and quantized back to FP8 precision using nearest neighbor rounding.

The PE array supports standard, transposed, and WU convolutions required for training. During FP and BP phases, it follows output stationary dataflow (OSD) and computes 16 output pixels of eight output channels in parallel with high PE utilization. However, OSD leads to low PE utilization for weight gradient computation in the WU phase, because the typically small kernel size (e.g., 3×3) makes it difficult for the 4-D tensor weight gradient [K, C, R, S] (K/C: number of output/input channels, $R \times S$: kernel size) to efficiently utilize the PE array. To overcome this, we propose a kernel stationary dataflow (KSD) during WU, where each PE computes one output kernel ($R \times S$) and the PE array computes 8/16 output/input channels (K/C) in parallel. The PE load balancer unit dynamically switches from OSD to KSD during the WU phase, improving the PE utilization (see Fig. 6).

2) Structured WS and Skipping: Fig. 7 shows the FP/BP operation with structured WS. WS is dynamically generated during training by group Lasso regularization. Weights are

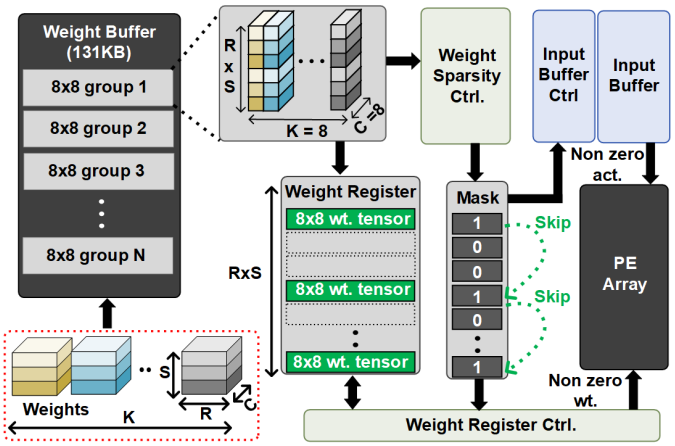


Fig. 7. Dynamic weight zero skipping dataflow.

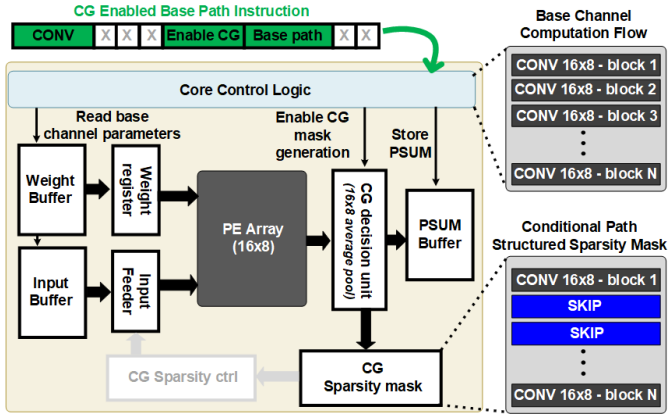


Fig. 8. Proposed SCG base path dataflow for dynamic activation sparsity.

divided into $8 \times 8 \times 1 \times 1$ ($K \times C \times R \times S$) groups and a regularizer term is added to the loss function, which scales the weight groups based on the group $L2$ norm. The CC generates 8×8 group sparsity in weights during the WU phase using the GLU. The weight memory stores the weights in groups of 8×8 with $R \times S$ elements. WS controller selects a weight group, stores it in the weight register, and compares the weights with a deterministic threshold to generate the sparsity mask. The input/weight buffer controllers use the sparsity mask to generate the read addresses by skipping the sparse weight groups.

3) Structured Channel Gating (SCG) Dataflow: Figs. 8 and 9 illustrate the SCG dataflow. We first compute all outputs of a given CNN layer using the base input channels in the SCG-enabled convolutions. Upon a base path instruction, the SpCC reads the parameters required to compute only the base path from input/weight memory and enables the SCG decision unit to compute an SCG sparsity mask, a 16×8 average pooling module, and a comparator, as depicted in Fig. 8. Structured activation sparsity is achieved by performing 16×8 average pooling instead of element-wise comparison. The SpCC reads the sparsity mask for the conditional path and enables structured output activation skipping, as shown in Fig. 9. The conditional path outputs are accumulated with base channel outputs stored in the partial sum memory. Since

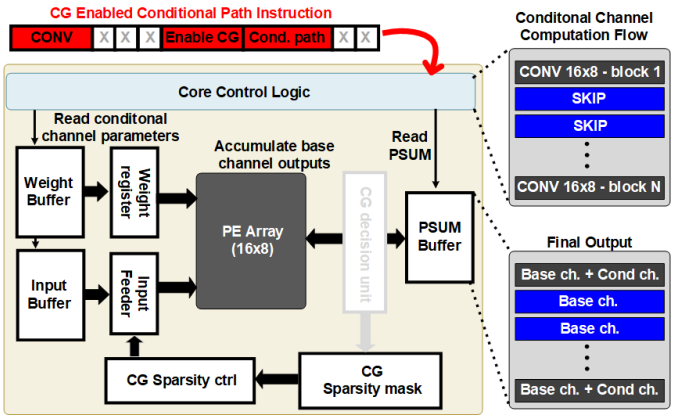


Fig. 9. Proposed SCG conditional path dataflow for dynamic activation sparsity.

Instruction	Width	Explanation
DDR3 Access	1	Enable DDR3 read/write
	6	Feature map height/width
	8	Number of access (NA)
	7	Number of loops for NA
	16	On-chip mem. address
	32	DDR3 access address
Conv. Layers	12	# of channels, kernel size
	2	Stride / transpose
	2	Memory read modes
	10	Channel gating, threshold
	3	Weight sparsity, KSD
Other Layers	2	Max pooling/upsampling
	2	Fully-connected weight
	2	Element-wise operations
Weight Update	1	Gradient accumulation
	12	On-chip mem. address
	2	Weight sparsity/update
	16	Group Lasso / learn. rate

Fig. 10. Proposed custom instruction set architecture that configures the SpCC.

the 16×8 block structure exactly matches the PE array size, the SCG sparsity controller efficiently eliminates input/weight memory accesses as well as the computations associated with the skipped 16×8 conditional path blocks, largely reducing training latency and energy.

C. Custom ISA

Fig. 10 shows the 128-bit instruction set architecture (ISA) used in CNN training, which is highly flexible and capable of supporting all operations needed for end-to-end CNN training. Although most instructions are shorter than 128 bits, we chose to support instruction lengths up to 128 bits in our design, which is the nearest power of 2 numbers to the longest instruction. This decision was made for the sake of design simplicity, but it can be optimized in future work.

The DDR3 access instruction facilitates off-chip memory accesses during training. With DDR3 access, we can easily

read and write activations, gradients, and weights stored in off-chip memory. When accessing big chunks of data, we can specify the burst length and the number of bursts required using the "number of access" and the "number of loops" fields, respectively. Additionally, the DDR3 access instruction provides feature map and kernel size information that the data scatter/gather unit uses to reorganize the DDR3 data in the on-chip SRAMs.

The convolution instruction supports normal convolutions, transpose convolutions, and convolutions with intratile accumulations (for weight gradient computation). With each of these convolutions, we have the option to enable or disable activation sparsity, channel gating, WS, and to specify the data flow, such as KSD. To efficiently process big convolution layers in CNN models, each layer is divided into smaller tiles based on the size of the on-chip SRAMs. Each tile is then represented using a single convolution instruction. Depending on the size of the layer, there may be multiple convolution instructions cascaded between several DDR3 access instructions to read/write new data from/to the DRAM. This approach is particularly useful for mapping large convolution layers whose parameters cannot fit into the on-chip SRAMs. By dividing the layer into smaller tiles and using multiple convolution instructions cascaded between DDR3 access instructions, we can efficiently process the convolution layer without requiring the entire set of parameters to be loaded into on-chip memory.

Nonconvolution layers, such as residual connections, pooling layers, fully connected WUs, and upsampling layers, typically have shorter instruction lengths than convolution layers, as they require less information to be processed. These layers consume less than 1% of the overall training latency. The majority of the training latency is consumed by convolutions and memory accesses. Like convolution instruction, each nonconvolution layer instruction is padded with DDR3 access instructions. For element-wise addition operations, we read one row of 128 bits (32 pixels) from both the input and output buffers, where the two feature maps, read from DDR3, are stored by the data scatter unit. We add the two feature maps using a 32×1 element-wise adder array and store the result in the output memory. Once all the processing is complete, the DDR3 instruction writes the output memory data back to the DRAM.

The WU instruction is a central core instruction that collects all the computed weight gradients from SpCC, scales them with the specified learning rate, and computes new weights. Additionally, the sparsity instruction field can be used to enable or disable sparsity during this process. We can also specify the group Lasso threshold and the learning rate in the instruction fields. Each instruction takes several cycles to complete based on the type of operation, sparsity, and layer size.

D. Sparse Training Compiler

Fig. 11 shows the sparse training compiler (STC). STC takes two inputs: frozen CNN model graph in protobuf (.pb) format from TensorFlow/PyTorch and user-provided hardware

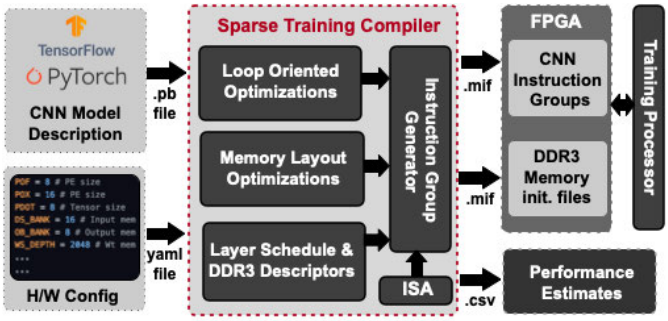


Fig. 11. Compiler for the sparse training processor.

configurations. The hardware configurations consist of three categories: 1) training parameters, such as SCG threshold, learning rate, group Lasso parameters, the number of SCG base channels, and so on; 2) hardware configuration details, such as SpCC selection, the number of memory banks, and the start address of a layer in the off-chip memory; and 3) on-chip memory and PE array size. To begin, the STC parses the entire graph to extract CNN model details and parameters from the frozen graph. With the extracted details and the user-provided hardware configurations, the STC performs optimizations, such as quantizing the parameters and optimizing memory layouts for on/off-chip memories. Additionally, the STC performs loop optimizations by unrolling convolution loops and dividing the convolution into smaller tiles. If the CNN model contains convolutions with channel gating, the STC divides the convolutions into the base channel and the conditional channel groups and enables activation sparsity for the condition channel group. During the backward pass, the STC parses the graph in reverse order and enables transposed convolutions and upsampling layers. Once the forward pass, backward pass, and WU pass are traversed in the graph, the STC schedules the operations/layers to minimize memory access and generates instructions using the ISA discussed in Section III-C. These instructions are then used to program the training processor for different CNN models.

To achieve the best training performance, the training parameters such as gradients, activations, and weights follow a predefined data layout in on-chip memory. The STC performs memory layout transformations on the raw inputs obtained from the neural network frameworks or the training dataset. This step includes compressing the data layouts to improve memory bandwidth utilization. The transformed data is stored in the FPGA DDR3 memory using the generated memory initialization files. The STC performs the memory allocations for each CNN layer, generates DDR3 descriptors, and embeds them in the instructions. The STC generates instructions based on loop/memory optimizations and divides them into small groups to facilitate the limited instruction memory of the accelerator. Each instruction group can carry one or multiple CNN layers' information depending on the layer size. These instruction groups are loaded one by one to the accelerator using FPGA.

With its tile-based convolution support, hardware compiler with custom ISA support, and off-chip memory communication capabilities, the training processor is well-equipped to

TABLE III
FLOP REDUCTION AND ACCURACY TRADEOFF ACROSS DIFFERENT SPARSITY VALUES/SCHEMES FOR RESNET-18 TRAINING ON CIFAR-10 DATASET

Method	Avg. Weight Sparsity (%)	Avg. CG Sparsity (%)	Gating Groups	Avg. Skipping Ratio (%)	FLOPs Reduction	CIFAR-10 Acc. (%)
FP32 Baseline	-	-	-	-	1.0 ×	94.78
FP8 Baseline	-	-	-	-	1.0 ×	94.57
This work (Structured)	47.08	56.62	2	54.31	3.01 ×	94.61
This work (Structured)	45.41	55.51	4	54.42	3.31 ×	94.50
This work (Structured)	79.61	68.71	4	69.65	7.32 ×	94.33

TABLE IV
ALGORITHM PERFORMANCE COMPARISON OF THE PROPOSED METHOD ON CIFAR-10 DATASET WITH RESNET-110 MODEL

Method	Model	Precision: W-A-G	FLOPs Reduction	Fine-tune	CIFAR-10 Accuracy (%)
SMD-SD [39]	ResNet-110	32-32-32	2.00×	✗	91.51
E^2-Train [40]	ResNet-110	8-8-16	6.75×	✓	91.74
SWAT [24]	ResNet-18	32-32-32	3.31×	✗	94.59
This work	ResNet-18	**8-8-8**	**7.32**×	✗	94.33
This work	ResNet-110	**8-8-8**	**5.43**×	✗	91.71

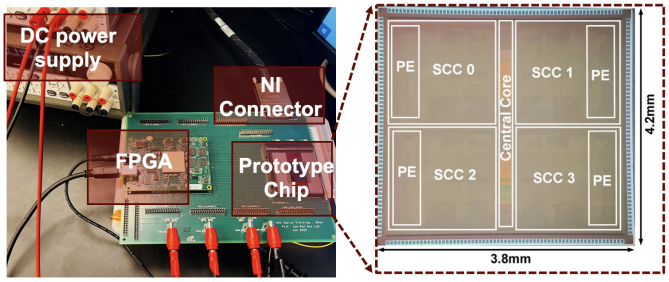


Fig. 12. Chip micrograph and the test setup.

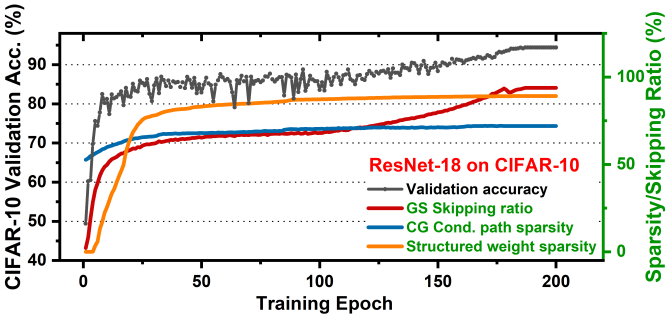


Fig. 13. Validation accuracy, sparsity, and skipping ratio of ResNet-18 FP8 training on CIFAR-10 dataset.

handle the three training phases of widely deployed CNN models, including those with residual connections, kernel sizes up to 7 × 7 (normal and transpose), and stride of 1 and 2. However, depthwise convolutions are not supported in the current design, which remains as future work. The processor architecture is scalable and can support larger models by increasing the size of the PE units and on-chip memory. For example, when we double the overall hardware capacity (compute and memory), we estimate that the training latency will improve by approximately 2× without requiring any hardware or dataflow changes. However, for much larger scaling factors ($\gg$2×), the performance can saturate due to 1) lower PE utilization and 2) high on-chip bandwidth requirements to feed the PE units. To achieve higher PE efficiency, the workload will need to be efficiently distributed among a large number of PE units to achieve optimal performance.

IV. RESULTS

The prototype chip was fabricated in 28-nm CMOS, and Fig. 12 shows the prototype chip micrograph and the chip testing setup. We employed XEM 7360 Opal Kelly FPGA to load the weights, training image, and compiler-generated instructions to the chip through an external IO interface.

A. Sparse Training Algorithm Performance Evaluation

We evaluate the proposed efficient training algorithm on the CIFAR-10 dataset with various DNN model architectures, including both VGG and deep ResNet architectures. We trained the selected models with both supervised and SSL [36] methods with our chip programed using custom ISA.

Table III summarizes the efficiency-accuracy tradeoff of the proposed efficient training algorithm for different sparsity on the CIFAR-10 dataset with ResNet-18 model architecture. Compared to the FP8 baseline, the proposed sparse training algorithm achieves 3.31× reduction in the number of FLOPs with 0.07% accuracy degradation and 7.32× reduction in FLOPs with 0.24% accuracy loss. Fig. 13 demonstrates the CIFAR-10 training progress of FP8 ResNet-18 with the proposed efficient training algorithm.

Compared to the recent SoTA efficient training algorithms [24], [39], [40], the proposed method exploits both weights and activation sparsity during low-precision (FP8) training, while dynamically skipping the BP with the awareness of the dedicated training accelerator. As shown in Table IV, the comprehensive exploration of sparse training leads to superior performance. Without using the pretrained

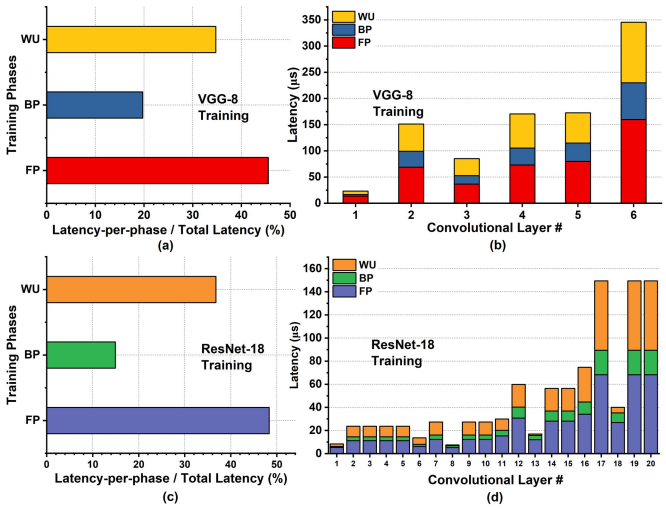


Fig. 14. Total/layer-wise sparse training latency breakdown of VGG-8 (a)/(b) and ResNet-18 (c)/(d) CNNs at 340 MHz.

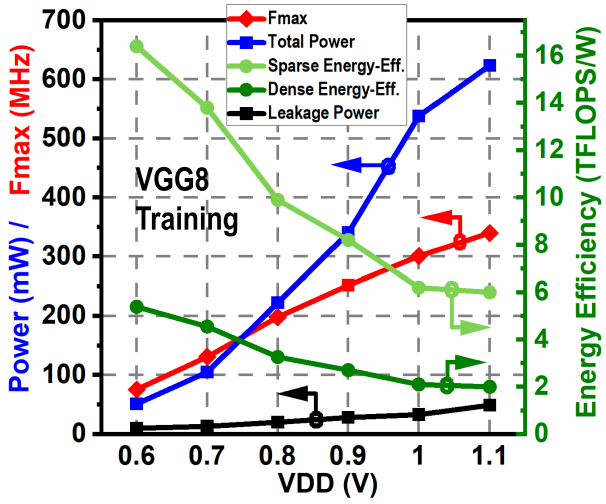


Fig. 15. Power, maximum frequency, and sparse/dense energy efficiency measurements with voltage and frequency scaling.

model, the proposed algorithm achieves up to 7.32×/5.43× computation reduction for training ResNet-18/ResNet-110 models.

B. Chip Measurement Results

Fig. 14 shows the total/layer-wise sparse training latency breakdown for VGG-8 and ResNet-18 CNNs at 340 MHz. GS enables us to efficiently skip the operations of BP/WU for low-confidence inputs thereby reducing the BP/WU training latency compared to FP.

Fig. 15 shows power, maximum frequency, and energy measurements of the prototype chip with voltage and frequency scaling. Including the skipped operations, a throughput of 3.76 TFLOPS is achieved at 1.1 V, and a peak sparse energy efficiency of 16.4 TFLOPS/W is achieved at 0.6 V for VGG8 training. Excluding the skipped operations and only enabling the KSD, we achieve 1.24 TFLOPS throughput at 1.1 V and a peak dense energy efficiency of 5.4 TFLOPS/W at 0.6 V

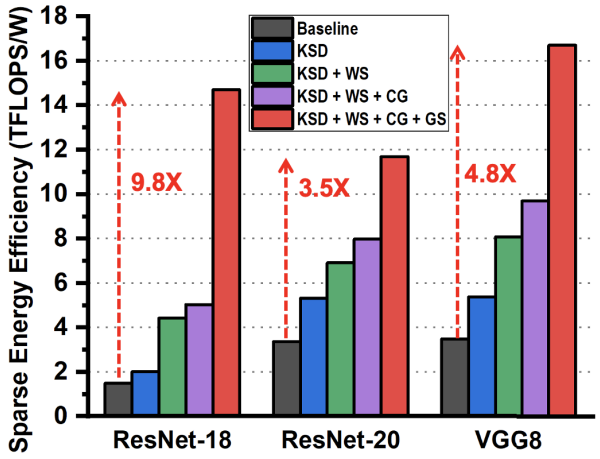
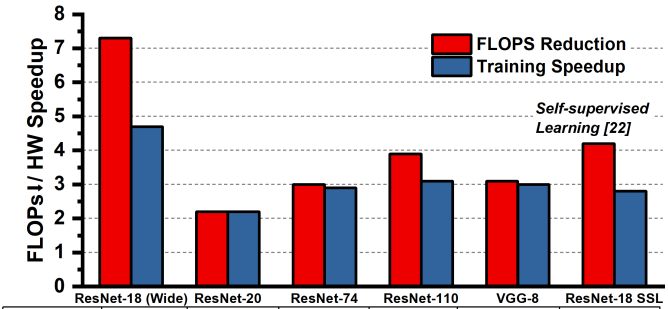


Fig. 16. Sparse energy efficiency improvement breakdown with the proposed techniques.



	ResNet-18 (Wide)	ResNet-20	ResNet-74	ResNet-110	VGG-8	ResNet-18 SSL
# of Params	11.17M	0.27M	1.15M	1.73M	12.97M	11.17M
Baseline Acc.	94.78	92.04	93.24	93.55	94.01	89.00
Sparse Acc.	94.33	91.41	92.33	91.73	93.38	88.32

Fig. 17. Reduction in FLOPs, hardware speedup, and accuracy on the CIFAR-10 dataset with various model architectures.

for VGG8 training. Fig. 15 shows both the sparse energy efficiency and dense energy efficiency across different supply voltages.

The proposed architecture efficiently exploits the structured weight/activation sparsity (WS/SCG) and improves PE utilization (KSD) during the overall training process, achieving large improvement in the sparse energy efficiency across different CNNs as shown in Fig. 16 (up to 9.8×), compared to the baseline of training dense CNNs without any of the proposed schemes.

We trained various CNNs for both supervised and SSL [36] tasks with our chip programed using custom ISA. As shown in Fig. 17, most of the FLOP reduction (up to 7.3×) results in corresponding training speedup (up to 4.7×) with minimal accuracy degradation. Training speedup corresponds to the improvement in chip latency when sparsity is enabled. The training speedup depicted here only represents training processor latency and does not include any off-chip memory access. We achieve high training speedup with minimal accuracy loss.

Table V shows the comparison to prior works, where our work achieves higher energy efficiency for actual CNN training, including and excluding the skipped operations. Han et al. [6] provide ∼3× more throughput than what we achieve however, their approach uses variable fixed-point

TABLE V
COMPARISON WITH PRIOR WORKS

		Park *et al.*, JSSC'20 [3]	Han *et al.*, JSSC'21 [6]	Kim *et al.*, Symp. VLSI'20 [7]	Wang *et al.*, Symp. VLSI'21 [8]	Lee *et al.*, ISSCC'19 [9]	Tu *et al.*, JSSC'20 [13]	Kim *et al.*, TCAS-I'22 [14]	**This work**
Technology		40nm	28nm	65nm	28nm	65nm	28nm	65nm	**28nm**
Precision		FP8 - SEB	Dyn. FXP	Fine/ Coarse	BFP8/16	FP8/16	Int2/4/8	FP8/16	**FP8/16**
Sparsity Support	**I/O**	N/A	Element (bit-slice)	Fine/Coarse	Element/Channel	Fine/Coarse	Runtime sparsity	Fine/Coarse	**Structured (channel gating)**
	W	N/A	N/A	Fine/Coarse	Element/Channel	N/A	N/A	Fine/Coarse	**Structured (group lasso)**
Supply (V)		0.75 – 1.1	0.58 - 1.04	0.78 - 1.1	0.58 - 1.1	0.78 - 1.1	0.75 - 1.1	0.78 - 1.1	**0.6 - 1.1**
Area (mm2)		6.25	12.96	16	20.96	16	5.64	16	**16.4**
Freq. (MHz)		20 - 180	2 - 250	50 - 200	40 - 440	50 - 200	120 - 268	200	**75 - 340**
On-Chip SRAM (MB)		0.293	0.55	0.34	0.63	0.372	N/A	0.34	**1.25**
Throughput (TFLOPS or TOPS)		0.567	10.2	0.61[1] - 18.0[2]	0.9[1] - 58.7[2]	151	0.13 (INT8)	0.6[1] - 56.4[2]	**0.56[1] - 3.7[3]**
Power (mW)		13.1 - 230	1.9 - 500	49 - 425	23 - 363	0.6	6.75-36	45 - 419	**51.1 - 623.7**
Energy Efficiency (TFLOPS/W or TOPS/W)	**Ideal[2] (90% I, 90% W)**	N/A	N/A	146.5	276.5	25.3	N/A	216.22	**N/A**
	Skipped[3] (sparse)	N/A	10.1 @ResNet-9 10.7 @ResNet-18	10.6 @AlexNet	N/A	N/A	3.83 @VGG8	1.71 @AlexNet	**16.4 @VGG8 11.7 @ResNet-20**
	Executed[4] (dense)	1.64 @ResNet18	N/A	~2.9 @AlexNet	4.3	3.4	N/A	1.48 @AlexNet	**5.4 @VGG8 5.3 @ResNet-20**
Training Speedup		N/A	N/A	N/A	1.76	1.82X	N/A	N/A	**4.7X**

[1] sparsity = 0%, [2] input/output/weight sparsity = 90% (not relevant for any specific DNN training), [3] peak energy-efficiency including skipped operations during sparse DNN training
[4] average energy-efficiency including skipped operations throughout sparse DNN training, [4] average energy-efficiency for actual executed operations throughout sparse DNN training

precision, which typically results in lower accuracy compared to floating-point representations. Our training speedup at FP8 precision is $\sim$2.7$\times$ higher than that of the state-of-the-art work [8].

V. CONCLUSION

In this work, we present an energy-efficient FP8 training processor with a custom ISA. Hardware-compatible, low-precision, global threshold-based channel gating is proposed to generate structured sparsity with negligible ($<$2%) accuracy degradation. We integrated the proposed hardware-efficient channel gating/group-Lasso algorithms into the training processor and enabled dual-zero skipping. A highly parallel fused multiply–add tree with dual data flow support (OSD, KSD) is implemented and yields high PE utilization during all three phases of the training. We employed a gradient skipping scheme that eliminates the need for BP and WU based on the FP prediction error. We evaluated our processor training various DNN benchmarks including supervised and SSL. The 28-nm prototype chip demonstrates a peak energy efficiency of 16.4 TFLOPS/W at 0.6 V. The proposed training optimization schemes collectively achieve up to 7.3$\times$ FLOPs reduction, up to 6.4$\times$ improvement in energy efficiency, and up to 4.7$\times$ improvement in training latency compared to the dense models.

REFERENCES

[1] J. Lee and H.-J. Yoo, "An overview of energy-efficient hardware accelerators for on-device deep-neural-network training," *IEEE Open J. Solid-State Circuits Soc.*, vol. 1, pp. 115–128, 2021.

[2] D. Han, S. Kang, S. Kim, J. Lee, and H.-J. Yoo, "Energy-efficient DNN training processors on micro-AI systems," *IEEE Open J. Solid-State Circuits Soc.*, vol. 2, pp. 259–275, 2022.

[3] J. Park, J. Lee, and D. Jeon, "A 65-nm neuromorphic image classification processor with energy-efficient training through direct spike-only feedback," *IEEE J. Solid-State Circuits*, vol. 55, no. 1, pp. 108–119, Jan. 2020.

[4] 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*, Feb. 2021, pp. 1–3.

[5] S. Yin and J.-S. Seo, "A 2.6 TOPS/W 16-bit fixed-point convolutional neural network learning processor in 65-nm CMOS," *IEEE Solid-State Circuits Lett.*, vol. 3, pp. 13–16, 2020.

[6] D. Han et al., "HNPU: An adaptive DNN training processor utilizing stochastic dynamic fixed-point and active bit-precision searching," *IEEE J. Solid-State Circuits*, vol. 56, no. 9, pp. 2858–2869, Sep. 2021.

[7] 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*, 2020, pp. 1–2.

[8] Y. Wang et al., "A 28 nm 276.55TFLOPS/W sparse deep-neural-network training processor with implicit redundancy speculation and batch normalization reformulation," in *Proc. Symp. VLSI Circuits*, Jun. 2021, pp. 1–2.

[9] 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.

[10] S. Kang et al., "GANPU: An energy-efficient multi-DNN training processor for GANs with speculative dual-sparsity exploitation," *IEEE J. Solid-State Circuits*, vol. 56, no. 9, pp. 2845–2857, Sep. 2021.

[11] S. Kim, J. Lee, S. Kang, J. Lee, W. Jo, and H.-J. Yoo, "PNPU: An energy-efficient deep-neural-network learning processor with stochastic coarse-fine level weight pruning and adaptive input/output/weight zero skipping," *IEEE Solid-State Circuits Lett.*, vol. 4, pp. 22–25, 2021.

[12] Z.-S. Fu, Y.-C. Lee, A. Park, and C.-H. Yang, "A 40-nm 646.6TOPS/W sparsity-scaling DNN processor for on-device training," in *Proc. IEEE Symp. VLSI Technol. Circuits (VLSI Technol. Circuits)*, Jun. 2022, pp. 40–41.

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

[14] S. Kim, J. Lee, S. Kang, D. Han, W. Jo, and H.-J. Yoo, "TSUNAMI: Triple sparsity-aware ultra energy-efficient neural network training accelerator with multi-modal iterative pruning," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 69, no. 4, pp. 1494–1506, Apr. 2022.

[15] J. Heo, J. Kim, S. Lim, W. Han, and J.-Y. Kim, "T-PIM: An energy-efficient processing-in-memory accelerator for end-to-end on-device training," *IEEE J. Solid-State Circuits*, vol. 58, no. 3, pp. 600–613, Mar. 2023.

[16] K. Prabhu et al., "CHIMERA: A 0.92-TOPS, 2.2-TOPS/W edge AI accelerator with 2-MByte on-chip foundry resistive RAM for efficient training and inference," *IEEE J. Solid-State Circuits*, vol. 57, no. 4, pp. 1013–1026, Apr. 2022.
[17] 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.
[18] J. Oh et al., "A 3.0 TFLOPS 0.62 V scalable processor core for high compute utilization AI training and inference," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2020, pp. 1–2.
[19] 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*, Feb. 2021, pp. 144–146.
[20] S. Venkataramani et al., "RaPiD: AI accelerator for ultra-low precision training and inference," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 153–166.
[21] S. K. Venkataramanaiah et al., "A 28 nm 8-bit floating-point tensor core based CNN training processor with dynamic activation/weight sparsification," in *Proc. IEEE 48th Eur. Solid State Circuits Conf. (ESSCIRC)*, Sep. 2022, pp. 89–92.
[22] S. Ruder, "An overview of gradient descent optimization algorithms," 2016, *arXiv:1609.04747*.
[23] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29, 2016, pp. 1–9.
[24] M. A. Raihan and T. Aamodt, "Sparse weight activation training," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 15625–15638.
[25] N. Lee, T. Ajanthan, and P. Torr, "SNIP: Single-shot network pruning based on connection sensitivity," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2018, pp. 1–15.
[26] J.-S. Seo et al., "Digital versus analog artificial intelligence accelerators: Advances, trends, and emerging designs," *IEEE Solid StateCircuits Mag.*, vol. 14, no. 3, pp. 65–79, Summer 2022.
[27] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. Peter Graf, "Pruning filters for efficient ConvNets," 2016, *arXiv:1608.08710*.
[28] S. Liu et al., "Sparse training via boosting pruning plasticity with neuroregeneration," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 34, 2021, pp. 9908–9922.
[29] X. Gao, Y. Zhao, L. Dudziak, R. Mullins, and C.-Z. Xu, "Dynamic channel pruning: Feature boosting and suppression," 2018, *arXiv:1810.05331*.
[30] F. Li, G. Li, X. He, and J. Cheng, "Dynamic dual gating neural networks," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2021, pp. 5330–5339.
[31] B. E. Bejnordi, T. Blankevoort, and M. Welling, "Batch-shaping for learning conditional channel gated networks," 2019, *arXiv:1907.06627*.
[32] Z. Su, L. Fang, W. Kang, D. Hu, M. Pietikainen, and L. Liu, "Dynamic group convolution for accelerating convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2020, pp. 138–155.
[33] W. Hua, Y. Zhou, C. M. De Sa, Z. Zhang, and G. E. Suh, "Channel gating neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 1–11.
[34] D. Shin, G. Kim, J. Jo, and J. Park, "Prediction confidence based low complexity gradient computation for accelerating DNN training," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.
[35] 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. (NIPS)*, vol. 31, 2018, pp. 1–10.
[36] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2020, pp. 1597–1607.
[37] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 9729–9738.
[38] J.-B. Grill et al., "Bootstrap your own latent—A new approach to self-supervised learning," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2020, pp. 21271–21284.
[39] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2016, pp. 646–661.
[40] Y. Wang et al., "E^2-train: Training state-of-the-art CNNs with over 80% energy savings," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 32, 2019, pp. 1–13.

Shreyas Kolala Venkataramanaiah (Member, IEEE) received the B.Tech. degree in electronics and communication from the Siddaganga Institute of Technology, Tumakuru, India, in 2016, and the M.S. and Ph.D. degrees in electrical engineering from the Arizona State University, Tempe, AZ, USA, in 2018 and 2021, respectively.

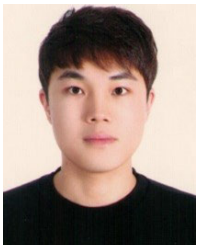
He is currently working as a Power Modeling Engineer, Apple Inc., Cupertino, CA, USA. His research interests include energy-efficient hardware design for machine learning, high-performance computing on FPGA/ASIC, and deep-learning compilers.

Jian Meng (Student Member, IEEE) received the B.S. degree from Portland State University, Portland, OR, USA, in 2019. He is currently pursuing the Ph.D. degree with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA.

His current research focuses on deep neural network compression optimization, self-supervised learning, hardware–software co-design with neuromorphic hardware acceleration, neuromorphic algorithm design for event-based vision and spiking neural networks, and energy-efficient object rendering.

Han-Sok Suh (Student Member, IEEE) received the B.S. degree (summa cum laude) from Inha University, Incheon, South Korea, in 2017, and the M.S. degree in computer engineering from the Viterbi School of Engineering, University of Southern California (USC), Los Angeles, CA, USA, in 2019. He is currently pursuing the Ph.D. degree with Arizona State University, Tempe, AZ, USA, working on computer architecture design on FPGA.

His area of research spans from software–hardware co-design to energy-efficient computing for machine learning applications.



Injune Yeo (Member, IEEE) received the B.S. degree in semiconductor science from Dongguk University, Seoul, South Korea, in 2011, and the M.S. degree in mechatronics engineering and the Ph.D. degree in electrical engineering from the Gwangju Institute of Science and Technology, Gwangju, South Korea, in 2014 and 2020, respectively.

From 2020 to 2022, he was a Post-Doctoral Scholar with the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA. Currently, he is an Assistant Professor with the School of Electrical Engineering, Chosun University, Gwangju, South Korea. His current research interests include an analog-to-digital converter, PUF, and in-memory computing.

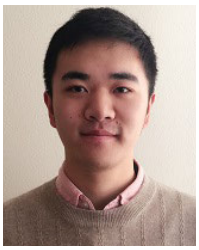
Jyotishman Saikia (Student Member, IEEE) received the B.Tech. degree in electronics and telecommunication engineering from KIIT University, Bhubaneswar, India, in 2016, and the M.S. degree in computer engineering from Arizona State University (ASU), Tempe, AZ, USA, in 2019, where he is currently pursuing the Ph.D. degree in electrical engineering.

His research interests include the design of memory systems, low-power design, and in-memory computation-based implementation of deep neural network algorithms.

Sai Kiran Cherupally (Member, IEEE) received the B.Tech. degree in electronics and communications engineering from Jawaharlal Nehru Technological University, Hyderabad, India, in 2015, the M.S. degree in electrical and computer engineering from Portland State University, Portland, OR, USA, in 2017, and the Ph.D. degree from the School of Electrical, Computer and Energy Engineering, Arizona State University, AZ, USA, in 2022.

He is currently working as a Research and Development Systems Engineer at Kilby Labs, Texas Instruments, Dallas, TX, USA. His research interests include machine-learning-assisted hardware security system design and energy-efficient hardware design.

Dr. Cherupally was a recipient of the Best Master's Student Award from Portland State University in 2017.



Yichi Zhang (Student Member, IEEE) is currently pursuing the Ph.D. degree with the Computer Systems Laboratory, Cornell University, Ithaca, NY, USA, advised by Prof. Zhiru Zhang.

His research interests align with the area of efficient ML model-hardware co-design. His work spans neural network quantization and binarization algorithms, ML accelerators, and building learning systems at hyperscale.

Zhiru Zhang (Fellow, IEEE) is an Associate Professor with the School of ECE, Cornell University, Ithaca, NY, USA. Before joining Cornell, he was a Co-Founder of AutoESL, a high-level synthesis startup later acquired by Xilinx (now AMD). His current research investigates new algorithms, design methodologies, and automation tools for heterogeneous computing.

Dr. Zhang's research has been recognized with a Facebook Research Award, Google Faculty Research Award, the DAC Under-40 Innovators Award, the Rising Professional Achievement Award from the UCLA Henry Samueli School of Engineering and Applied Science, a DARPA Young Faculty Award, and the IEEE CEDA Ernest S. Kuh Early Career Award, an NSF CAREER Award, the Ross Freeman Award for Technical Innovation from Xilinx, and multiple best paper awards and nominations.

Jae-Sun Seo (Senior Member, IEEE) received the B.S. degree in electrical engineering from Seoul National University, Seoul, South Korea, in 2001, and the M.S. and Ph.D. degrees in electrical engineering from the University of Michigan, Ann Arbor, MI, USA, in 2006 and 2010, respectively.

From 2010 to 2013, he was with IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, where he worked on cognitive computing chips under DARPA SyNAPSE Project and energy-efficient integrated circuits for high-performance processors. In 2014, he joined the School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA, where he is now an Associate Professor. He has held Visiting Researcher positions at Intel Circuits Research Laboratory in 2015 and Meta Reality Labs from 2022 to 2023. His current research interests include efficient hardware design of machine learning and neuromorphic algorithms and integrated power management.

Dr. Seo was a recipient of the Samsung Scholarship in 2004 and 2009, the IBM Outstanding Technical Achievement Award in 2012, the NSF CAREER Award in 2017, the Intel Outstanding Researcher Award in 2021, and the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS Best Paper Award in 2022. He has been a Technical Program Committee Member for ISSCC, DATE, DAC, ICCAD, and MLSys. He serves as an Associate Editor for IEEE OPEN JOURNAL OF THE SOLID-STATE CIRCUITS SOCIETY and IEEE TRANSACTIONS ON BIOMEDICAL CIRCUITS AND SYSTEMS.