ESSCIRC 2022- IEEE 48th European Solid State Circuits Conference (ESSCIRC) | 978-1-6654-8494-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ESSCIRC55480.2022.9911359

A 28nm 8-bit Floating-Point Tensor Core based CNN Training Processor with Dynamic Activation/Weight Sparsification

Shreyas Kolala Venkataramanaiah*, Jian Meng*, Han-Sok Suh*, Injune Yeo*, Jyotishman Saikia*,
Sai Kiran Cherupally*, Yichi Zhang†, Zhiru Zhang†, and Jae-sun Seo*
*Arizona State University, USA †Cornell University, USA

***Abstract*—We present an 8-bit floating-point (FP8) training 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 based on group Lasso, and (4) gradient skipping based on FP prediction error. We develop a custom ISA to flexibly support different CNN topologies and training parameters. The 28nm prototype chip demonstrates large improvements in 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*—Convolutional neural networks, deep neural network training, structured sparsity, hardware accelerator**

I. Introduction

Training deep convolutional neural networks (CNNs) requires a large amount of memory and iterative computation, which necessitates speedup and energy reduction for both cloud and edge devices. Several prior works reported CNN training processor designs, but some of them did not exploit sparsity during training [1], [2]. Bit-slice input/output sparsity is exploited in [3], but did not consider weight sparsity during training. In [4], the sparse channels are randomly selected to simplify the hardware design, but resulted in training accuracy loss. A global threshold is used in [5] to generate the element-wise sparse masks without sorting, but the non-structured sparse elements are not skippable.

In this work, we present a new sparse CNN training accelerator that exploits structured activation sparsity, structured weight sparsity, and gradient skipping to dynamically reduce the unimportant operations and achieve high speedup. We developed both hardware-efficient sparse training algorithms and the custom sparse training accelerator with programmable instructions. The 28nm prototype chip demonstrates large improvements in FLOPs reduction (7.3×), energy efficiency (16.4 TFLOPS/W), and overall training latency speedup (4.7×) across supervised and self-supervised training tasks.

II. Hardware-Efficient Sparse Training Algorithm

A. *Dynamic Structured Weight Sparsification*

We generate structured weight sparsity dynamically during CNN training via group Lasso [6], where the unimportant weight groups get penalized without any sorting. Suppose the weight groups in DNN layer l is $W_{l,g}$. The Lasso penalty term

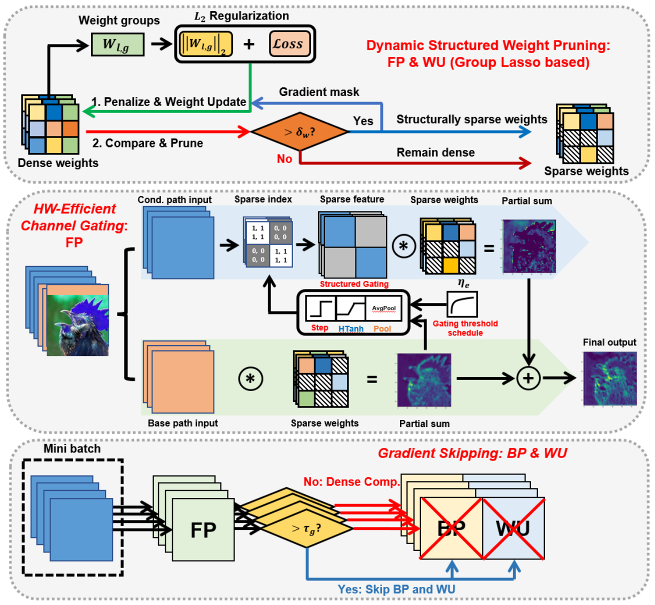


Fig. 1: Proposed efficient sparse training techniques.

in the loss function ($\hat{\mathcal{L}}$ in Eq. (1)) is the summation of group L_2 norm of $W_{l,g}$ over all groups (G_l) and layers (L).

$$\hat{\mathcal{L}} = \mathcal{L} + \lambda \sum_{l=1}^{L} \sum_{g=1}^{G_l} ||W_{l,g}||_2, \tag{1}$$

$$w^*_{i,g} = \underbrace{(1 - \eta \frac{\lambda}{||W_{l,g}||_2})}_{GL_{scale}} w_{i,g} - \underbrace{\eta \frac{\partial \mathcal{L}}{\partial \nabla w_{i,g}}}_{\text{Normal gradient}} \tag{2}$$

In Eq. (2), $\nabla w_{i,g}$ is the gradient of the weight element $w_{i,g} \in W_{l,g}$. The weight $w^*_{i,g}$ is updated by the normal gradient and also scaled down by the regularization scaler GL_{scale}. Since L_2 norm of the weight group is pre-computed, GL_{scale} can be easily implemented in hardware. Smaller weights are gently scaled down, while larger weights are penalized harder. As a result, every $|w_{i,g}| \in W_{l,g}$ decreases simultaneously, so the element-wise threshold can easily sparsify the entire group. The resultant structured sparse masks are applied to both FP and WU. Following the hardware design, we choose K_l (# of output channels) × C_l (# of input channels) = 8×8 as the sparse group size for efficient hardware mapping.

B. *Structured Activation Skipping*

We also exploit the activation sparsity by skipping the unsalient features during FP. In CGNet [7], activation sparsity was employed during CNN inference without any auxiliary salience predictors. The input/output channels are divided into base and conditional paths, and the gated base path's feature score determined the skipping of conditional path computation. However, the learnable threshold and the non-linear gating function are expensive for hardware implementation.

To elevate the hardware compatibility, we propose structured CGNet (SCG), including (a) low-precision threshold schedule, (b) non-linearity relaxation, and (c) structured feature map sparsification. The learning pattern of the gating threshold is generalized as a gradually increased low-precision scheduler, and the non-linear Sigmoid gating function [7] is simplified as a scaled and shifted HardTanh function:

$$\mathcal{G}(y^*_{base}) = \underset{\min=0,\max=1}{\mathrm{HardTanh}}\left(0.25 \times (y^*_{base} + 2) - 0.5\right) \quad (3)$$

We generate structured sparsity of feature maps by computing the *group salience* scores of the base path output features via 3-D average pooling and HardTanh (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.

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 [7]	Sigmoid	Learnable	70.21%	Element-wise	94.45%
SCGNet	**HardTanh**	**Scheduled**	68.71%	Structured	94.41%

C. Gradient Skipping

Inputs with high confidence during FP will have minimal weight update in the training process, thus can be skipped from the BP and WU phases [8]. 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.

Putting together the structured weight sparsity, structured activation skipping and gradient skipping, ResNet-18 FP8 training result for CIFAR-10 dataset is shown in Fig. 2.

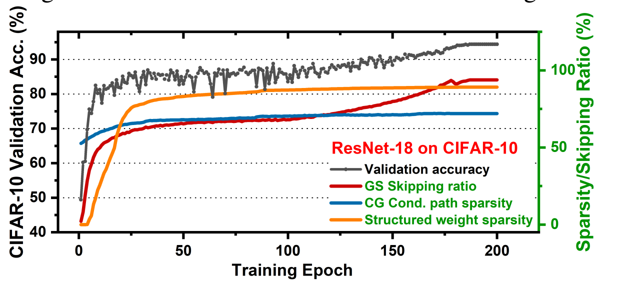
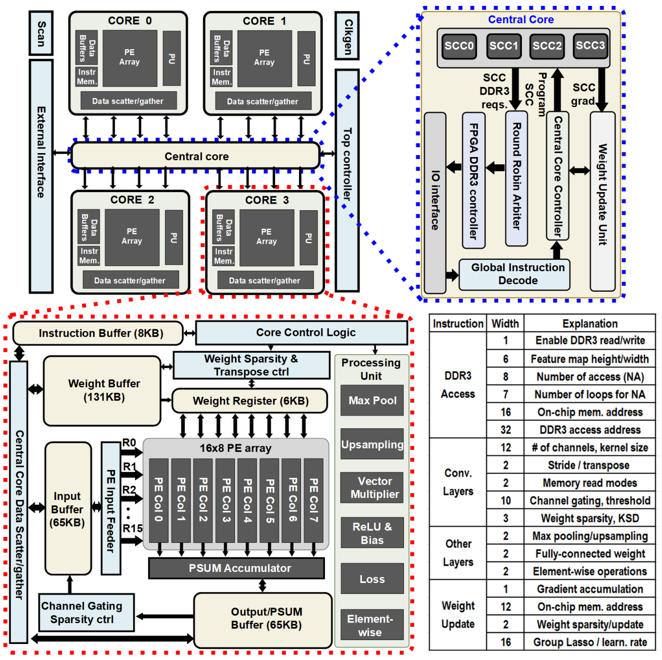


Fig. 2: Validation accuracy, sparsity, and skipping ratio of ResNet-18 FP8 training on CIFAR-10 dataset.

III. Chip Architecture and Operation

Fig. 3 shows the overall architecture of the sparse training processor, featuring four sparse compute cores (SpCC), a central core (CC), a global controller, and external I/O interface.



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. 3: Overall chip architecture and custom ISA.

A. Sparse Compute Cores (SpCC) and Central Core (CC)

Each SpCC is a programmable core that can be configured individually using the custom ISA (Fig. 3). The ISA is flexible to support all operations for CNN training, facilitates off-chip memory accesses during training, and controls all proposed sparsity/dataflow optimizations. The CC synchronizes all four SpCCs, accumulates gradients from SpCCs in WU phase, and processes off-chip DDR3 access requests using a round-robin arbiter. The SpCC includes (1) a 16×8 PE array for MAC operations, (2) SRAMs to store activations, weights, instructions, and sparsity masks, (3) vector processing units for non-MAC operations, (4) sparsity controllers to exploit dynamic structured sparsity of activations/weights, and (5) data scatter/gather units to store SRAM data in the required format.

During WU, the CC performs (1) weight gradient (WG) accumulation, (2) structured weight sparsity generation for new weights, and (3) weight update at the end of the batch based on stochastic gradient descent. After all four SpCCs complete the WG computations, a gradient accumulation instruction is dispatched to the CC controller. The CC controller loads the weight update memory with the previous WGs and reads all computed WGs from the SpCCs. The WG accumulator obtains all gradients and accumulates them using an FP16 adder tree.

B. PE Array

Each SpCC has a PE array that consists of eight PE columns with 16 PEs and a PE load balancer (Fig. 4). The PE column shares the weights obtained from the weight register, and the PE row shares the input activations during FP and BP. Each PE is a configurable dot-product engine with eight FP8 (1-5-2) multipliers and one FP16 (1-5-10) adder. The multiplier products are aligned by matching the exponents and shifting the mantissa. The aligned mantissa products are sent to the

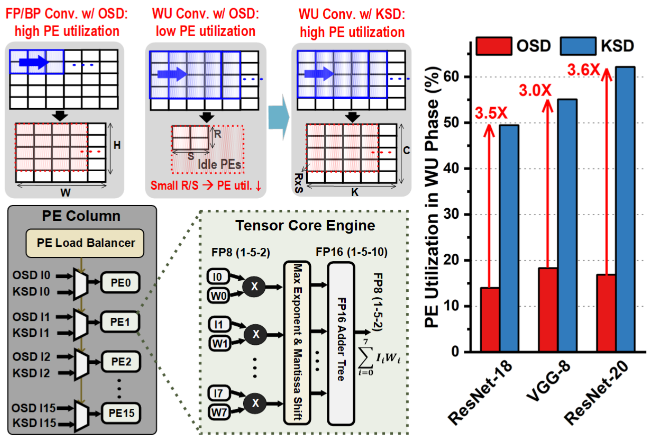

FP/BP Conv. w/ OSD: high PE utilization
WU Conv. w/ OSD: low PE utilization
WU Conv. w/ KSD: high PE utilization
Idle PEs
Small R/S → PE util. ↓
PE Column
PE Load Balancer
Tensor Core Engine
FP8 (1-5-2)
FP16 (1-5-10)
Max Exponent & Mantissa Shift
FP16 Adder Tree
OSD
KSD
PE Utilization in WU Phase (%)
3.5X
3.0X
3.6X
ResNet-18
VGG-8
ResNet-20

Fig. 4: PE array supporting OSD (FP/BP) and KSD (WU).

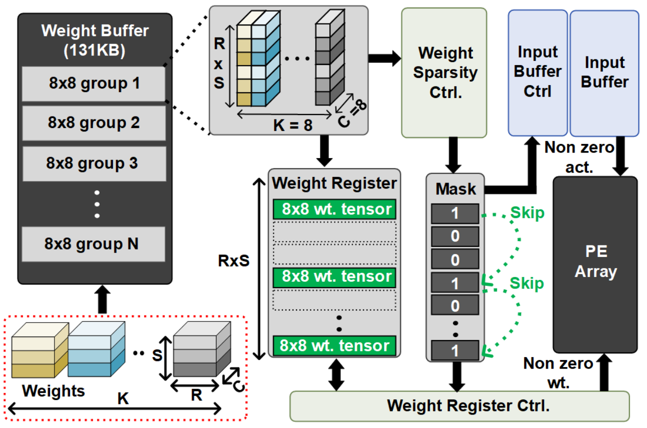


Fig. 5: Convolution with dynamic structured weight sparsity.

8-way FP16 adder tree, whose output is normalized and quantized back to FP8 precision (nearest neighbor rounding).

The PE array supports standard, transposed, and weight update convolutions required for training. During FP and BP, it follows output stationary dataflow (OSD) and computes 16 output pixels of 8 output channels in parallel with high PE utilization. However, OSD leads to low PE utilization for weight gradient computation during WU, 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 use the PE array. To overcome this, we propose a new 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 WU phase, improving the PE utilization (Fig. 4 (right)).

C. Structured Weight Sparsity

Fig. 5 shows the FP/BP operation with structured weight sparsity (WS) that is generated in. 8×8×1×1 ($K{\times}C{\times}R{\times}S$) groups. 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. Only the non-zero weight groups are executed in the PE array.

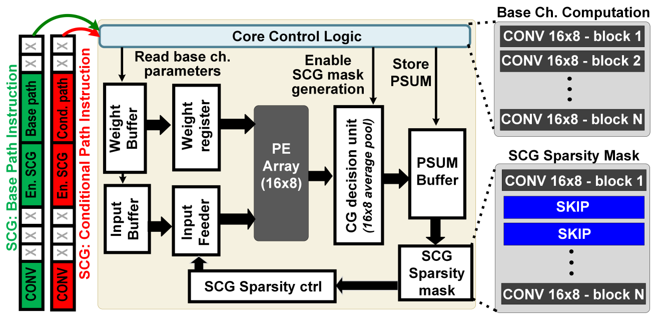


Fig. 6: Proposed SCG dataflow for dynamic activation sparsity.

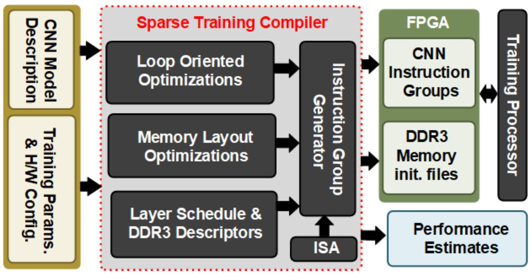

CNN Model Description
Training Params. & HW Config.
Sparse Training Compiler
Loop Oriented Optimizations
Memory Layout Optimizations
Layer Schedule & DDR3 Descriptors
Instruction Group Generator
ISA
FPGA
CNN Instruction Groups
DDR3 Memory init. files
Training Processor
Performance Estimates

Fig. 7: Compiler for the sparse training processor.

D. Structured Channel Gating (SCG) Dataflow

Fig. 6 describes 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 a SCG sparsity mask, a 16×8 average pooling module, and a comparator. 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 (CP) and enables structured output activation skipping. The CP outputs are accumulated with base channel outputs stored in the partial sum memory. Since 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 CP blocks, largely reducing latency/power.

E. Sparse Training Compiler

Fig. 7 illustrates the sparse training compiler (STC), which takes CNN models from TensorFlow or PyTorch frameworks as inputs, extracts the layer-wise details, and generates an instruction snippet deployed on the accelerator. STC also takes (1) training parameters such as SCG threshold, learning rate, group Lasso parameters, etc. and (2) hardware configuration details such as SpCC selection, enable/disable the sparsity memory layout, etc. as inputs. Based on these inputs and the given on-chip memory and PE array size of the accelerator, STC performs loop optimizations including unrolling the convolution loops and dividing the convolution into smaller tiles.

IV. Measurement Results

The prototype chip was fabricated in 28nm CMOS (Fig. 8(a)). Fig. 8(b) shows power/energy measurements of

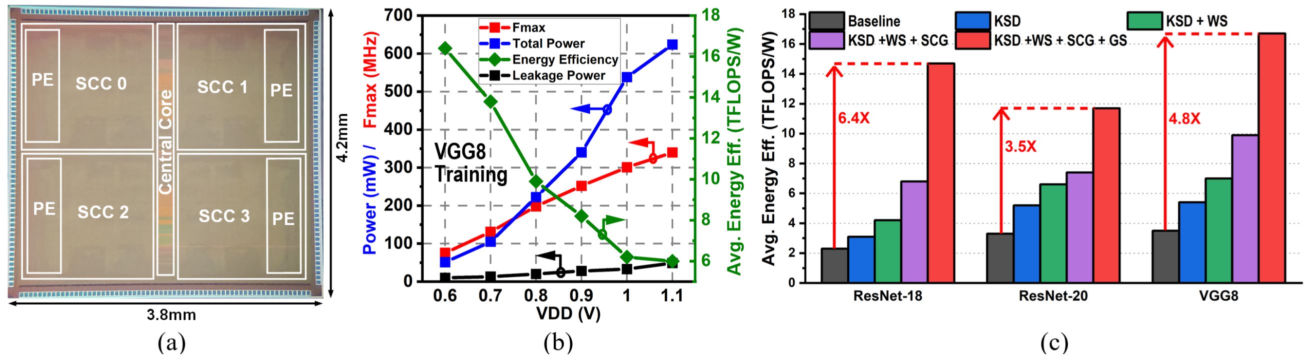
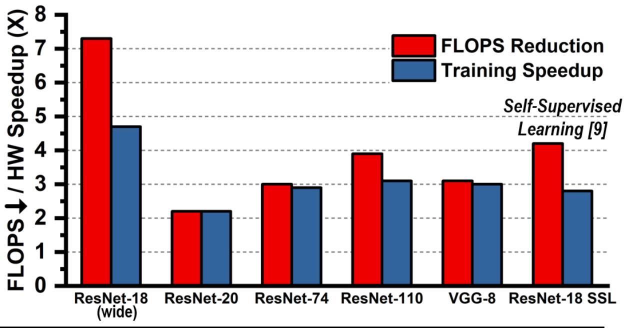


Fig. 8: (a) Chip micrograph. (b) Power/energy measurements with voltage and frequency scaling. (c) Average energy-efficiency improvement breakdown with the proposed techniques on ResNet-18/ResNet-20/VGG8 models.



# of Params	11.17M	0.27M	1.15M	1.73M	12.97M	11.17M
Baseline Acc.	94.01%	92.04%	93.24%	93.55%	94.01%	89.00%
Sparse Acc.	93.38%	91.41%	92.33%	91.73%	93.38%	88.32%

Fig. 9: FLOPS, speedup, and accuracy for training of CNNs.

the prototype chip with voltage and frequency scaling. Including the skipped operations, throughput of 3.76 TFLOPS is achieved at 1.1V, and average energy-efficiency of 16.4 TFLOPS/W is achieved at 0.6V for VGG8 training. The proposed architecture efficiently exploits the structured sparsity (CG, WS) and improves PE utilization (KSD) during training achieving ∼6.4× improvement in energy-efficiency (Fig. 8(c)) compared to the baseline of training dense CNNs without activation/weight sparsity, KSD and GS.

We trained various CNNs for both supervised and self-supervised learning [9] tasks with our chip programmed using custom ISA. As shown in Fig. 9, most of the FLOPs reduction (up to 7.3×) results in corresponding training speedup (up to 4.7×) with minimal accuracy degradation. Table II shows the comparison to prior works, where our work achieves higher average energy-efficiency for actual DNN training, including and excluding the skipped operations. Our training speedup is ∼2.7× higher than that of the state-of-the-art work [5].

V. Conclusion

In this work, we present an energy-efficient 8-bit floating-point programmable training processor with a custom ISA. The proposed KSD/WS/CG/GS schemes collectively achieved a high amount of hardware-efficient sparsity and computation skipping (up to 7.3×). The 28nm training processor was evaluated across various CNNs, and achieved high energy-efficiency and 4.7× speedup compared to training dense CNNs.

TABLE II: Comparison with prior works.

		[3]	[4]	[5]	This Work
Technology		28nm	65nm	28nm	**28nm**
Precision		Dyn. FXP	FP8/16	BFP8/16	**FP8/16**
Sparsity Support	I/O	Element (bit-slice)	Fine / coarse	Element / channel	**Structured (channel gating)**
	W	N/A	Fine / coarse	Element / kernel	**Structured (group Lasso)**
Supply (V)		0.58-1.04	0.78-1.1	0.58-1.1	**0.6-1.1**
Area (mm^2)		12.96	16.0	20.96	**16.4**
Freq. (MHz)		2-250	50-200	40-440	**75-340**
On-Chip SRAM (MB)		0.55	0.34	0.63	**1.25**
Throughput (TFLOPS)		3.6-8.13	0.61^1-18.0^2	0.9^1-58.7^2	**3.76^4**
Power (mW)		1.9-500	0.49-425	23-363	**51.1-623.7**
Energy-Efficiency (TFLOPS/W or TOPS/W)	Dense^1	3.3-7.5 (12b-8b)	3.1	4.3	**3.5 @VGG8 2.3 @ResNet-18**
	Ideal^2 (90% I, 90% W)	N/A	146.5	276.5	**N/A**
	Peak-Skipped^3	14.5 @ResNet-9 15.2 @ResNet-18	10.6 @AlexNet	N/A	**37.6 @VGG8 64.7 @ResNet-18**
	Avg.-Skipped^4	10.1 @ResNet-9 10.7 @ResNet-18	~6.8 @AlexNet	N/A	**16.4 @VGG8 14.4 @ResNet-18**
	Avg.-Executed^5	N/A	N/A	N/A	**5.4 @VGG8 3.1 @ResNet-18**
Training Speedup		N/A	N/A	1.76X	**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

VI. Acknowledgements

This work is supported in part by NSF and JUMP C-BRIC, a SRC program sponsored by DARPA.

References

[1] A. Agrawal *et al.*, “A 7nm 4-core AI chip with 25.6 TFLOPS hybrid FP8 training, 102.4 TOPS INT4 inference and workload-aware throttling,” in *IEEE ISSCC*, 2021.
[2] J. Park *et al.*, “A 40nm 4.81 TFLOPS/W 8b floating-point training processor for non-sparse neural networks using shared exponent bias and 24-Way fused multiply-add tree,” in *IEEE ISSCC*, 2021.
[3] D. Han *et al.*, “HNPU: An adaptive DNN training processor utilizing stochastic dynamic fixed-point and active bit-precision searching,” *IEEE Journal of Solid-State Circuits*, vol. 56, no. 9, pp. 2858–2869, 2021.
[4] S. Kim *et al.*, “A 146.52 TOPS/W deep-neural-network learning processor with stochastic coarse-fine pruning and adaptive input/output/weight skipping,” in *Symp. on VLSI Circuits*, 2020.
[5] Y. Wang *et al.*, “A 28nm 276.55 TFLOPS/W sparse deep-neural-network training processor with implicit redundancy speculation and batch normalization reformulation,” in *Symp. on VLSI Circuits*, 2021.
[6] W. Wen *et al.*, “Learning structured sparsity in deep neural networks,” in *NeurIPS*, 2016.
[7] W. Hua *et al.*, “Channel gating neural networks,” in *NeurIPS*, 2019.
[8] D. Shin *et al.*, “Prediction confidence based low complexity gradient computation for accelerating DNN training,” in *ACM/IEEE DAC*, 2020.
[9] T. Chen *et al.*, “A simple framework for contrastive learning of visual representations,” in *ICML*, 2020.