Abstract—Training Convolutional Neural Networks (CNNs) usually requires a large number of computational resources. In this paper, SparseTrain is proposed to accelerate CNN training by fully exploiting the sparsity. It mainly involves three levels of innovations: activation gradients pruning algorithm, sparse training dataflow, and accelerator architecture. By applying a stochastic pruning algorithm on each layer, the sparsity of back-propagation gradients can be increased dramatically without degrading training accuracy and convergence rate. Moreover, to utilize both natural sparsity (resulted from ReLU or Pooling layers) and artificial sparsity (brought by pruning algorithm), a sparse-aware architecture is proposed for training acceleration. This architecture supports forward and back-propagation of CNN by adopting 1-Dimensional convolution dataflow. We have built a cycle-accurate architecture simulator to evaluate the performance and efficiency based on the synthesized design with 14nm FinFET technologies. Evaluation results on AlexNet/ResNet show that SparseTrain could achieve about 2.7× speedup and 2.2× energy efficiency improvement on average compared with the original training process.

Index Terms—Convolutional Neural Networks, Training, Sparse, Pruning, Accelerator, Architecture

I. INTRODUCTION

Recent years, Convolutional Neural Networks (CNNs) have been widely used in computer vision tasks such as image classification [1], object detection [2], face recognition [3], and object tracking [4]. Modern CNN models could achieve state-of-the-art accuracy by extracting rich features and significantly outperform traditional methods [5]. However, running large scale CNN models often requires large memory space and consumes huge computational resources.

Many CNN accelerator architectures have been invented to improve the CNN inference throughput and energy efficiency in both industrial [6], [7] and academic [8]–[16] communities. Among them, exploiting network sparsity has been observed as one of the most efficient approaches. For example, by applying weight pruning, the model size of AlexNet and VGG-16 could be reduced by 35× and 49×, respectively [17], which provides a huge potential for accelerating CNN inference.

In fact, by using weight sparsity, a sparse-aware computing architecture — EIE [18] achieved 13× speedups and 3400× energy efficiency when compared to GPU implementation of the same DNN without compression. Fig. 1(a) and Fig. 1(b) show the original CNN inference process and the one with weight pruning and sparsity utilization, respectively. One step further, SCNN [9] achieved significant speedup in CNNs inference by utilizing both weight sparsity and natural sparsity of activations. Fig. 1(c) shows the basic idea of this procedure.

Most works focused on the inference of CNN. However, the sparsity of the training process was less studied. Compared with inference, CNN training demands much more computational resources. Usually CNN training introduces about 3× computational cost and consumes 10× to 100× memory space compared with the inference.

To utilize gradient sparsity in CNNs training, [19] [20] proposed a method that uses only three numerical levels {−1,0,1} for weight gradients to reduce communication in distributed training, which can be considered as a mixture of pruning and quantization. This scheme is shown in Fig. 1(d). However, it can hardly reduce the computation since the main training process (forward and back-propagation) is still calculated in a dense data format. There are also many training accelerators try to utilizing sparsity, which can be found in [21]. But neither of them exploit gradient sparsity.

Since activation gradients is an operand for both backward process and weight gradients generation process, the sparsity of activation gradients provides a significant reduction on the computation of training. To overcome the limitation of previous works and fully

This work is supported in part by the National Natural Science Foundation of China (61602022), State Key Laboratory of Software Development Environment (SKLSDE-2018ZX-07), CCF-Tencent IAGR20180101 and the 111 Talent Program B16001.
Forward stage starts from input layer until final layer, including convolutional (CONV), ReLU and MaxPool layer. Input activations \( I \) of \( n \)-th CONV layer are formulated as a 3-D tensor. The size of \( I \) is determined by the number of input channels \( C \), height \( H \) and width \( W \), and we denote the input activations of \( i \)-th channel as \( I_i \). Output activations \( O \) are also formulated as a 3-D tensor with \( F \) output channels. Weights are formulated as a 4-D tensor with the size of \( K \times K \times C \times F \), where each convolution kernel \( W_{i,j} \) is a 2-D tensor with the size of \( K \times K \). Vector \( b \) is the bias applied on \( O \) after convolution. If 2-D convolution is denoted by “*”, the CONV layer can be represented as

\[
O_i = \sum_{j=0}^{C} W_{i,j} * I_j + b_i, \quad i = 0, \cdots, F.
\]

ReLU and MaxPool are non-linear operation layers. ReLU layer applies point-wise function \( f(x) = \text{max}(0, x) \) on all activations. MaxPool layer select the maximum value from each window of activations as output. Hence, the activations usually become sparse after ReLU and MaxPool layers. And the non-zero patterns generated by ReLU and MaxPool layers are recorded as mask and will be adopted in backward stage.

**Backward** stage includes two steps:

- **Gradient To Activations (GTA):** it calculates the activation gradients (derivatives to certain layer’s activations). According to chain rule, the gradients are calculated from loss function back to input layer. The CONV layer in GTA step is represented as

\[
dI_j = \sum_{i=0}^{F} dO_i * W_{i,j}^+, \quad j = 0, \cdots, C,
\]

where \( dI_j \) indicates the input activation gradients (derivatives of input activations) for the \( j \)-th channel, \( dO_i \) represents the output activation gradients for the \( i \)-th channel, \( W_{i,j} \) is the \( i \)-th filter of \( j \)-th channel, \( W_{i,j}^+ \) is sequentially reversed of \( W_{i,j} \), or in other words, \( W_{i,j} \) rotated by 180 degrees. ReLU and MaxPool layers in GTA step directly adopt the pre-stored mask in forward stage.

- **Gradient To Weights (GTW):** it calculates the weight gradients \( dW \) (derivatives of loss function to layers’ weights). These weight gradients are utilized to update weights by Stochastic Gradient Descent (SGD) method. Weight gradients are calculated by

\[
dW_{i,j} = dO_i * I_j, \quad i = 0, \cdots, F, \quad j = 0, \cdots, C,
\]

where \( dW_{i,j} \) is the weight gradients of \( j \)-th channel in \( i \)-th filter.

**Weight Update** stage is to update the weights of each layer with the calculated weight gradients by using SGD method. For modern CNNs, batch training is a popular way to update the weights by sending a batch of inputs (e.g. 32 images) into the network. The weight gradients are computed by averaging the batch of gradients. Finally, the weights are updated according to a pre-set learning rate \( \alpha \). Generally, weight update stage is not a performance bottleneck for CNN training. Thus, **only the Forward, GTA and GTW procedures are taken into considerations for acceleration**.

### III. Algorithm

#### A. Stochastic Pruning

The stochastic gradients pruning algorithm. \( \tau \) is the threshold that determines whether a value should be pruned. \( p \) is the probability of setting a value to 0 or \( \pm \tau \).

From the experiments, we found that there are many gradients data whose absolute value is very small. Intuitively, a single small gradient value has little effect on the updating of weight, thus it can be pruned (set to zero) directly. However, if many values are pruned, the distribution of gradients is changed remarkably, which causes accuracy loss. Thus, we adopt a stochastic pruning algorithm proposed by (23) to solve this problem.

This algorithm treats the gradients that to be pruned (denoted as \( g \)) as a \( n \)-dimensional vector, and prunes the vector component whose absolute value is smaller than the threshold \( \tau \).
the stochastic pruning method. By setting values to zero or ±τ stochastically, the expectation of each component remains unchanged, which improves the convergence and accuracy. Detailed analysis can be found in [23].

Algorithm 1: Overall Pruning Scheme

\[
\begin{align*}
1 & \quad F \triangleq \text{FIFO with depth } N_F, \\
2 & \quad \text{for } i = 1; i \leq N; i \leftarrow i + 1 \text{ do} \\
3 & \quad \quad g \triangleq G_i; \quad \hat{g} \triangleq \hat{G}_i; \\
4 & \quad \quad n = \text{length of } g; \quad A = 0; \\
5 & \quad \quad \text{for } j = 1; j \leq n; j \leftarrow j + 1 \text{ do} \\
6 & \quad \quad \quad A \leftarrow A + |g_i|; \\
7 & \quad \quad \quad \text{if } i > N_F \text{ then} \\
8 & \quad \quad \quad \quad \tau = \text{mean}(F); \\
9 & \quad \quad \quad \quad \quad \text{if } |g_i| < \tau \text{ then} \\
10 & \quad \quad \quad \quad \quad \quad \text{Generate a random number } r \in [0, 1]; \\
11 & \quad \quad \quad \quad \quad \quad \quad \text{if } |g_i| > \tau r \text{ then} \\
12 & \quad \quad \quad \quad \quad \quad \quad \quad \hat{g}_i = (g_i > 0) ? \hat{\tau} : (-\hat{\tau}); \\
13 & \quad \quad \quad \quad \quad \quad \quad \text{else} \\
14 & \quad \quad \quad \quad \quad \quad \quad \quad \hat{g}_i = 0; \\
15 & \quad \quad \quad \quad \quad \text{else} \\
16 & \quad \quad \quad \quad \quad \quad \quad \hat{g}_i = g_i; \\
17 & \quad \quad \quad \quad \tau = \Phi^{-1}\left(\frac{1-p}{2}\right) \frac{1}{n} \sqrt{\frac{\sigma^2}{2}} A; \\
18 & \quad \quad \quad \text{F.push}(\tau); \\
\end{align*}
\]

B. Threshold Determination and Prediction

Clearly, it’s unfeasible to find the threshold by sorting, due to its huge memory and time consumption. Thus, we propose a hardware friendly threshold selection scheme with less overhead. This selection method contains 2 steps: Determination and Prediction, where the Determination step refers to [23].

Threshold Determination. Modern CNN models have two typical structures, as shown in Fig. [4] For the CONV-ReLU structure, where a CONV layer is followed by a ReLU layer, output activation gradients \( \mathbf{dO} \) are sparse, but subject to a irregular distribution. On the other hand, the input activation gradients \( \mathbf{dI} \), which going to be propagated to the previous layer, is full of non-zero values. Statistics show that the distribution of \( \mathbf{dI} \) is symmetric with 0 and its density decreases with the increment of absolute value. For the CONV-BN-ReLU structure, \( \mathbf{dO} \) subjects to the same distribution of \( \mathbf{dI} \). Simply, we assume these gradients all subject to a normal distribution with mean 0 and variance \( \sigma^2 \) according to [23].

\[
\sigma = \frac{1}{n} \sqrt{\frac{2}{\pi} \sum_{i=1}^{n} |g_i|}, \quad g_i \in g.
\]

Then we can compute the threshold \( \tau \) using the cumulative distribution function \( \Phi \) of the standard normal distribution, target pruning rate \( p \) and \( \hat{\sigma} \):

\[
\tau = \Phi^{-1}\left(\frac{1-p}{2}\right) \frac{1}{\hat{\sigma}} A.
\]

Threshold Prediction. The stochastic pruning scheme mentioned above needs to access all gradients two times: for computing \( A \) and for gradients pruning. What is worse, gradients need to be stored in memory temporarily before pruned, which brings overhead of memory access. The best way is to prune gradients before they are sent back to memory. To accomplish this, we improve the algorithm by predicting the threshold before calculating. Denoting the number of batches as \( N \), the prediction method keeps a FIFO with a length \( N_F \) for each CONV layer, where \( N_F \) is a hyper-parameter that satisfies \( N_F << N \).

Fig. [5] shows the pruning algorithm with threshold prediction for each CONV layer. Activation gradients of each batch will be pruned under the predicted threshold \( \tau' \) as soon as they are calculated, where \( \tau' \) is the average of all the thresholds stored in the FIFO. At the end of each batch, The determined threshold of this batch is calculated and pushed into the FIFO. Gradients will not be pruned before the FIFO is filled up. Details of the whole gradients pruning algorithm is shown in Algorithm [1].

The arithmetic complexity of our algorithm is \( O(n) \), while complexity of sorting is at least \( O(n \log n) \). Besides, all the gradients will be accessed just one time in our algorithm, so that almost no extra storage is required, which saves time and energy consumption.

IV. Datalflow

To gain more benefits from the above algorithm, it’s essential to design an accelerator that can utilize both activation sparsity and gradient sparsity. Prior works have shown that the utilized datalflow usually affects the architecture’s performance significantly [8]. Thus, we first introduce the datalflow used in the accelerator.

We propose a sparse training datalflow by dividing all computation into 1-Dimensional convolutions. This datalflow supports all kinds of sparsity in training and provides opportunities for exploiting all types of data reuse. This section will introduce the sparsity in training and the datalflow in detail.

A. Data Sparsity in Training

The sparsity of involved six data types during training have been summarized in Table [11] Input activations \( \mathbf{I} \) for each CONV layer are usually sparse because the previous ReLU layer sets negative activation values into zeros. Weights \( \mathbf{W} \) are also dense for all steps of training. The output activation gradients \( \mathbf{dO} \) are usually sparse. But for networks with BN layers, \( \mathbf{dO} \) becomes dense after passing through BN layers. However, this issue can be resolved by our gradients pruning algorithm. Thus, we can regard output activation gradients of all CONV layers as sparse data.
There is an additional optimization opportunity in the GTA step. The gradients \(dI\) are usually sent to a ReLU layer after generated from a CONV layer, leading to certain values be set to zero forcefully. Actually, we could predict the positions of these zeros and skip the corresponding calculations, according to the mask generated in the Forward step. Besides, both operands (input activations \(I\) and gradients to output activation \(dO\)) in the GTW step are sparse. Theoretically, it could significantly reduce the computation cost in obtaining the weight gradients \(dW\).

B. Sparse Training Dataflow

Aiming to utilize all the sparsity in the training process, we divide one 2-D convolution into a series of 1-D convolutions and treat the 1-D convolutions as basic operations for scheduling and running. In this subsection, we will show the way to disassemble three basic steps of CNN training into 1-D convolutions in detail.

**Forward step.** As shown in Fig. (6a) for one particular 2-D convolution, a small kernel moves through the input activations and perform multiplications and accumulations on each position. More detailed, one output row is the addition of \(K\) 1-D convolution results, where \(K\) is the kernel size. For a 1-D convolution in the Forward step, one operand is a certain row of the kernel, which is a short dense vector. Another operand is a row of the input activations, which is a long sparse vector. This basic operation is denoted as Sparse Row Convolution (SRC).

**GTW step.** The weight gradients computation in the GTW step is shown in Fig. (6c). The GTW step is significantly different from the Forward and the GTA step in two ways. The first is that operands of extracted 1-D convolutions are two sparse long vectors. The second lies in the range of results: traditional convolutions need to slide one vector from the start of the other vector to its end, and calculating multiplications and accumulations in each sliding position. However, in the GTW step, only the results in several positions are needed, and it is not necessary to fully calculate all the convolution results. The resulting vector is usually short (as the kernel size \(K\)) and we can store it in a small scratchpad during the whole convolution. Thus, this 1-D convolution is named Output Store Row Convolution (OSRC) and is considered as the basic operations of the GTW step.

Gradients to bias in CONV layers are also required in the GTW step. The calculation of bias gradients for each channel is just the summation of the output activation gradients from the corresponding channel. It is simple to calculate them by accumulating gradients during the GTA step.

### V. Architecture

Aiming to evaluate the dataflow sparsity in SparseTrain, we design an architecture of which overview is shown in Fig. (7a). The architecture consists of Buffer, Controller, and PE groups. Each PE group contains 3 PEs and a Post Processing Unit (PPU). PE is designed to calculate 1-D convolution and PPU is adopted for point-wise operations.

**PE Architecture:** As demonstrated in Section [IV] there are three kinds of basic operations in our dataflow: SRC, MSRC and OSRC. PE module is designed to support all of these operations and the architecture of PE module is shown in Fig. (7c).

Our PE is designed to perform a complete 1-D convolution, instead of just one multiplication. Each time an input value is loaded from Port-1, PE multiplies it by \(K\) values in Reg-1, providing \(K\) product results stored in Reg-2.

When performing SRC operations, A PE first loads weight vector from Port-2 and saves them in Reg-1. Then activation values are loaded from Port-1 and multiplied by the weights in Reg-1. Results are accumulated to Reg-2. This process repeats until the activation vector gets to the end.

For MSRC operations, the offset vector of input activations \(I\) is loaded from Port-3 and saved to Reg-2. Values in the offset vector is used to indicate the results that should be calculated. In other words,
TABLE II: Evaluation results for the gradients pruning algorithm, where \( \text{acc}\% \) means the training accuracy and \( \rho_{\text{nnz}} \) means the density of non-zeros.

| Model          | Dataset       | Baseline \( \text{acc}\% \) | \( p = 70\% \) \( \rho_{\text{nnz}} \) | \( p = 80\% \) \( \rho_{\text{nnz}} \) | \( p = 90\% \) \( \rho_{\text{nnz}} \) | \( p = 99\% \) \( \rho_{\text{nnz}} \) |
|----------------|---------------|-----------------------------|-------------------------------|-----------------------------|-------------------------------|-----------------------------|
| AlexNet        | CIFAR-10      | 90.14 1                      | 90.05 1                       | 89.50 1                    | 89.26 1                      |
| ResNet-18      | CIFAR-10      | 92.30 1                      | 92.05 1                       | 91.60 1                    | 91.30 1                      |
| ResNet-34      | CIFAR-10      | 90.04 1                      | 90.01 1                       | 89.60 1                    | 89.30 1                      |
| ResNet-152     | CIFAR-10      | 95.70 1                      | 95.65 1                       | 95.30 1                    | 95.20 1                      |
| AlexNet        | CIFAR-100     | 67.61 0.10                   | 67.51 0.10                   | 67.31 0.10                 | 67.21 0.10                   |
| ResNet-18      | CIFAR-100     | 76.47 1                      | 76.37 1                       | 76.27 1                    | 76.17 1                      |
| ResNet-34      | CIFAR-100     | 77.51 1                      | 77.41 1                       | 77.30 1                    | 77.20 1                      |
| ResNet-152     | CIFAR-100     | 79.25 1                      | 80.51 1                       | 81.76 1                    | 83.00 1                      |
| AlexNet        | ImageNet      | 56.38 0.07                   | 57.10 0.05                   | 56.84 0.04                 | 55.38 0.04                   |
| ResNet-18      | ImageNet      | 68.73 1                      | 69.02 0.41                   | 68.58 0.40                 | 68.38 0.38                   |
| ResNet-34      | ImageNet      | 72.93 1                      | 72.92 0.39                   | 72.86 0.38                 | 72.74 0.37                   | 72.42 0.34                   |

VI. EVALUATION

In this section, several experiments are conducted to demonstrate that the proposed approach could reduce the training complexity significantly with a negligible model accuracy loss. In the following experiments, AlexNet [1] and ResNet [22] series are evaluated on three datasets including CIFAR-10, CIFAR-100 and ImageNet. All the models are trained for 300 epochs on CIFAR-10, 100 and 180 epochs on ImageNet, due to our limited computing resources. PyTorch framework is adopted for the algorithm verification.

To verify the performance of our dataflow and architecture, a custom cycle-accurate C++ simulator is implemented based on SystemC model for the proposed architecture, and a simple compiler is designed with Python to convert CNN models in PyTorch into our internal instructions for driving our simulator. We also have the RTL implementation for PE, PPU and controller, which is synthesized by Synopsys Design Compiler with Global Foundries 14nm FinFET technology for area estimation and then simulated by Synopsys PrimeTime for power estimation. The area and energy consumption of buffer (SRAM) is estimated by PACTI [24]. To demonstrate the advantages of our proposed sparse training dataflow, we compare it with the architecture from Eyeriss [8]. Since Eyeriss is designed for CNN inference rather than training, we modify the architecture of Eyeriss to support the dense training process. We adopt 168 PEs in both the proposed architecture and the baseline architecture. For convenience, 386KB SRAM is utilized as the global buffer for intermediate data, which is sufficient for storing data used in each iteration. A larger buffer is beneficial to improving data-reuse and energy efficiency, but it is beyond the considerations of this work.

A. Sparsity and Accuracy

From Table [II] there is no accuracy lost for most situations and even 1% accuracy improvement for ResNet-50 on CIFAR-100. The only significant accuracy loss exists in AlexNet on ImageNet, when using a very aggressive pruning policy like \( p = 99.5\% \). That proves the accuracy loss caused by our layer-wise gradients pruning algorithm is almost negligible.

The gradient density illustrated in Table [II] shows that our method could reduce the gradient density by \( 3 \times \sim 10 \times \). In addition, the deeper networks could obtain a relatively lower gradient density with our sparsification, which means that our method works better for larger networks.

B. Convergence

The training loss of AlexNet/ResNet-18 on CIFAR-10 and ImageNet is demonstrated in [23]. In general, ResNet-18 is very robust for gradients pruning. For AlexNet, the gradients pruning could still be robust on CIFAR-10 but will degrade convergency speed.
a little on ImageNet with aggressive pruning policy. These results prove that with basically hyper-parameter $p$, our pruning algorithm basically has the same convergence property compared with the original training scheme.

C. Latency and Energy

Fig. 8 shows the training latency reduction brought by sparsity exploration. The proposed SparseTrain scheme achieves $4.5 \times$ speedup at most for AlexNet on CIFAR-10. On average, it achieves about $2.7 \times$ speedup compared with the baseline.

Fig. 9 shows the average energy consumption per data sample. Overall, SparseTrain has $1.5 \times$ to $2.8 \times$ (on average $2.2 \times$) energy efficiency improvement than baseline. For the baseline, $62\% \sim 71\%$ of the energy consumption comes from SRAM accesses. SparseTrain reduces the global buffer accesses by utilizing sparse dataflow and reduces the energy cost by $30\% \sim 59\%$. The energy consumption of combinational logic in SparseTrain could be reduced by $53\% \sim 88\%$, which is more significant than SRAM and register accesses. This also contributes much to the total energy saved.

VII. CONCLUSION

As the model size and datasets scale larger, CNN training becomes more and more expensive. In response, we propose a novel CNN training scheme called SparseTrain for acceleration by fully exploiting sparsity in the training process. By performing stochastic gradients pruning on each layer, SparseTrain achieves high sparsity with negligible accuracy loss. Additionally, with a 1-D CONV based sparse training dataflow and a sparse-aware architecture, SparseTrain can improve CNN training speed and efficiency significantly by exploiting different types of sparsity. With the threshold prediction method, gradients pruning can be performed on our architecture with almost no overhead. Experiments show that the gradients pruning algorithm could achieve $3 \times \sim 10 \times$ sparsity improved with negligible accuracy loss. Compared with the baseline, SparseTrain achieves about $2.7 \times$ speedup and $2.2 \times$ energy efficiency improvement on average.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in Proc. NIPS, 2012, pp. 1097–1105.
[2] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” in Proc. NIPS, 2015, pp. 91–99.
[3] Y. Sun, D. Liang, X. Wang, and X. Tang, “DeepID3: Face recognition with very deep neural networks,” arXiv preprint:1502.00873, 2015.
[4] L. Bertinetto, J. Valmadre et al., “Fully-convolutional siamese networks for object tracking,” in Proc. ECCV, 2016, pp. 850–865.
[5] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” Nature, 2015.
[6] N. P. Jouppi, C. Young, N. Patil et al., “In-datacenter performance analysis of a tensor processing unit,” in Proc. ISCA, 2017, pp. 1–12.
[7] N. Jouppi, C. Young, N. Patil et al., “Motivation for and evaluation of the first tensor processing unit,” in Proc. MICRO, 2018, pp. 10–19.
[8] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in Proc. ISCA, 2016, pp. 367–379.
[9] A. Parashar, M. Rhu et al., “SCNN: An accelerator for compressed-sparse convolutional neural networks,” in Proc. ISCA, 2017, pp. 27–40.
[10] M. Shafaii, S. Gupta, Y. Kim, and T. Rosing, “FloatPIM: In-memory acceleration of deep neural network training with high precision,” in Proc. ISCA, 2019, pp. 802–815.
[11] X. He, W. Lu, G. Yan, and X. Zhang, “Joint design of training and hardware towards efficient and accuracy-scalable neural network inference,” JETCAS, vol. 9, no. 4, pp. 810–821, 2018.
[12] Y. Li, J. Park, M. Alian et al., “A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks,” in Proc. MICRO, 2018, pp. 175–188.
[13] A. Ren, T. Zhang, S. Ye et al., “ADMM-NN: An algorithm-hardware co-design framework of dnn systems using alternating direction methods of multipliers,” in Proc. ASPLOS, 2019, pp. 925–938.
[14] Y. Wen, C. Xu, C. Wu et al., “Coordinating filters for faster deep neural networks,” in Proc. ICCV, 2017, pp. 658–666.
[15] W. Wen, Y. He et al., “Learning intrinsic sparse structures within long short-term memory,” arXiv preprint:1709.05027, 2017.
[16] B. Chen, T. Medini, J. Farwell et al., “SLIDE: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems,” arXiv preprint:1903.03129, 2019.
[17] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” arXiv preprint:1510.00149, 2015.
[18] S. Han, X. Liu, H. Mao et al., “EIE: efficient inference engine on compressed deep neural network,” ACM SIGARCH Computer Architecture News, vol. 44, no. 3, pp. 243–254, 2016.
[19] W. Wen, C. Xu, F. Yan et al., “TernGrad: Ternary gradients to reduce communication in distributed deep learning,” in Proc. NIPS, 2017, pp. 1509–1519.
[20] Z. He and D. Fan, “Simultaneously optimizing weight and quantizer activation gradients,” in Proc. CVPR, 2019, pp. 11 438–11 446.
[21] “DNN training accelerator comparison,” http://www.ci-lab.net/research/dnn-training-accelerator-comparison/ 2019.
[22] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in Proc. CVPR, 2016, pp. 770–778.
[23] X. Ye, J. Yang, P. Dai et al., “Accelerating CNN training by pruning activation gradients,” arXiv preprint:1908.00173, 2019.
[24] A. Shafaii, Y. Wang, X. Liu, and M. Pedram, “FinCACTI: Architectural analysis and modeling of caches with deeply-scaled FinFET devices,” in Proc. ISVLSI, 2014, pp. 290–295.