Efficient *N:M* Sparse DNN Training Using Algorithm, Architecture, and Dataflow Co-Design

Chao Fang, Graduate Student Member, IEEE, Wei Sun, Aojun Zhou, and Zhongfeng Wang, Fellow, IEEE

Abstract-Sparse training is one of the promising techniques to reduce the computational cost of DNNs while retaining high accuracy. In particular, N:M fine-grained structured sparsity, where only N out of consecutive M elements can be nonzero, has attracted attention due to its hardware-friendly pattern and capability of achieving a high sparse ratio. However, the potential to accelerate N:M sparse DNN training has not been fully exploited, and there is a lack of efficient hardware supporting N:M sparse training. To tackle these challenges, this paper presents a computation-efficient training scheme for N:M sparse DNNs using algorithm, architecture, and dataflow co-design. At the algorithm level, a bidirectional weight pruning method, dubbed BDWP, is proposed to leverage the N:M sparsity of weights during both forward and backward passes of DNN training, which can significantly reduce the computational cost while maintaining model accuracy. At the architecture level, a sparse accelerator for DNN training, namely SAT, is developed to neatly support both the regular dense operations and the computationefficient N:M sparse operations. At the dataflow level, multiple optimization methods ranging from interleave mapping, pregeneration of N:M sparse weights, and offline scheduling, are proposed to boost the computational efficiency of SAT. Finally, the effectiveness of our training scheme is evaluated on a Xilinx VCU1525 FPGA card using various DNN models (ResNet9, ViT. VGG19. ResNet18. and ResNet50) and datasets (CIFAR-10. CIFAR-100, Tiny ImageNet, and ImageNet). Experimental results show the SAT accelerator with the BDWP sparse training method under 2:8 sparse ratio achieves an average speedup of $1.75 \times$ over that with the dense training, accompanied by a negligible accuracy loss of 0.56% on average. Furthermore, our proposed training scheme significantly improves the training throughput by $2.97 \sim 25.22 \times$ and the energy efficiency by $1.36 \sim 3.58 \times$ over prior FPGA-based accelerators.

I. INTRODUCTION

D EEP neural networks (DNNs) have been widely used in many applications, such as computer vision, speech recognition, autonomous driving, and robotics. However, their impressive accuracy comes at the cost of expensive computational requirements: the state-of-the-art DNNs could contain trillions of parameters [1] and consume thousands of peta-level floating-point operations (FLOPs) [2] for the training process. To relieve this burden, it is crucial to seek a computationefficient scheme for DNN training.

C. Fang and Z. Wang are with the School of Electronic Science and Engineering, Nanjing University, Nanjing 210008, China (e-mail: fantasy-see@smail.nju.edu.cn; zfwang@nju.edu.cn).

W. Sun is with Electronic System Group, Eindhoven University of Technology, Netherlands (e-mail: w.sun@tue.nl).

A. Zhou is with CUHK-Sensetime Joint Lab, CUHK, Hong Kong, China (e-mail: aojunzhou@link.cuhk.edu.hk).

Sparse training [3]-[12] is one of the promising techniques to reduce the training computational cost while retaining impressive accuracy of DNNs. It dynamically prunes elements such as weights, activations, gradients, and then eliminates computations associated with the pruned elements during training iterations. Prior works mostly focus on exploiting structured [6], [7] or unstructured [8]–[11] sparsity patterns for sparse DNN training. Structured sparsity [13]-[15], which involves pruning entire kernels or channels of weights at a coarse-grained level, has limitations in reducing the number of FLOPs of DNN training (less than 40% sparse ratio in [6]). On the other hand, unstructured sparsity [16]-[18] involves pruning elements in any position without constraints, leading to a high sparse ratio (over 80% sparse ratio in [9]) and a significant reduction in the number of FLOPs for training. However, its irregular pattern makes it challenging to be fully utilized on hardware for effective training speedup [19]–[21].

In recent years, there has been a growing interest in leveraging fine-grained structured sparsity [21]-[25] for efficient DNN acceleration. N:M sparsity [21] has attracted significant attention among various fine-grained structured sparsity for its practical sparsity ratio and hardware-friendly pattern, which allows only N out of consecutive M elements in a group to be nonzero. The static 2:4 sparse pattern was initially introduced by NVIDIA Ampere GPUs [20] for efficient DNN inference. However, researchers have gone beyond the 2:4 static pattern and explored more aggressive N:M sparse patterns, such as 2:8 or 2:16, which have demonstrated significant inference acceleration [26]–[28] with competitive accuracy to the dense counterparts [29]-[31]. In addition to its use in efficient DNN inference, N:M sparsity has also shown potential in reducing the computational cost of DNN training [3], [32]. However, accelerating N:M sparse DNN training is a challenging task that involves several issues to be addressed.

- The potential to accelerate *N:M* sparse DNN training has not been fully exploited. Previous works solely accelerate DNN training by introducing *N:M* sparsity in either forward pass [32] or backward pass [3]. A unified approach that takes advantage of *N:M* sparsity in both passes could lead to further acceleration of DNN training.
- Current hardware platforms are unable to fully leverage the sparsity benefits of *N:M* sparsity to accelerate DNN training. Firstly, the Ampere GPUs [20] only support 2:4 sparse operations, which restricts the acceleration capability of various *N:M* patterns with higher sparsity ratios for efficient DNN training. Secondly, while *N:M* sparse data can be packed in advance for DNN infer-

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. This work has been accepted by the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD).

This work was supported in part by the National Key R&D Program of China under Grant 2022YFB4400604 and in part by the National Natural Science Foundation of China under Grant 62174084. (*Corresponding author: Zhongfeng Wang.*)

ence deployment, the N:M elements need to be updated in training iterations. However, the lack of dedicated hardware implementation for this iterative update process results in substantial computational overhead, hampering the speedup of N:M sparse training.

• Dataflow mapping optimizations are required to further improve hardware utilization and achieve significant acceleration. Specifically, in *N:M* sparse DNN training, various types of computational intensive operations in both forward and backward passes, require tailored dataflow optimizations for both dense and *N:M* sparse operations to better utilize hardware resources and accelerate the training process.

To address the aforementioned issues, this paper presents a computation-efficient N:M sparse training scheme for DNNs, featuring three aspects: algorithm, architecture, and dataflow. 1) The bidirectional weight pruning algorithm for N:M sparse training, namely BDWP, leverages the N:M sparse pattern on weights in both forward and backward passes, and significantly reduces the number of training operations. 2) The efficient hardware architecture for N:M sparse DNN training, dubbed as SAT, supports both regular dense matrix multiplication (MatMul) and N:M sparse MatMul with improved training throughput and energy efficiency. Additionally, SAT also supports online N:M sparse reduction of data, further enhancing its computational efficiency. 3) The dataflow optimization techniques, including interleave mapping, pre-generation of N:M sparse weights, and offline scheduling, further increase hardware utilization and improve the throughput of SAT.

The main contributions can be summarized as follows:

- Algorithm: We propose a bidirectional weight pruning method for *N:M* sparse training, namely BDWP, leveraging *N:M* sparsity of weights during both forward and backward passes of DNN training. Compared to dense training, our 2:8 BDWP training reduces the number of training operations by 48% with a negligible accuracy loss of 0.56% on average.
- Architecture: We propose a sparse accelerator for training DNNs, namely SAT, to support computation-efficient *N:M* sparse operations besides the regular dense operations. It achieves 2.97~25.22× higher throughput and 1.36~3.58× greater energy efficiency than prior training accelerators [33]–[39] evaluated on FPGA.
- **Dataflow**: We propose several dataflow optimization methods, including interleave mapping, pre-generation of *N:M* sparse weights, and offline scheduling, to boost the computational efficiency of SAT.
- Scheme: We present an efficient scheme for DNN training that incorporates the BDWP algorithm and the SAT architecture. It improves the training speed by $1.75 \times$ on average compared to the conventional dense training scheme deployed on SAT.

The remainder of this paper is organized as follows. Sec. II introduces training steps of DNNs and reviews sparse training techniques and FPGA-based training accelerators. Sec. III, Sec. IV, and Sec. V describe BDWP algorithm, SAT architecture, and optimization methods for SAT dataflow, respectively.

Sec. VI presents experimental results to illustrate the effectiveness of our proposed computation-efficient scheme for *N:M* sparse DNN training. The paper is concluded in Sec. VII.



Fig. 1. Example of the training process for a convolutional layer: (a) it consists of FF in the forward pass, BP, and WU in the backward pass. An input tensor can be transformed into a matrix if required by the (b) *im2col* process. (c), (d), and (e) illustrate the MatMul operations during FF, BP, and WU, respectively, using a single batch data.

II. BACKGROUND AND RELATED WORKS

A. Training Steps of DNNs

As shown in Fig. 1 (a), the training steps of a DNN layer consist of two stages: the forward and the backward passes. In the forward pass, the network layer feeds forward (FF) the activations as input and produces the outputs based on the trainable weights. The intermediate outputs computed during FF must be kept in memory for the backward pass. In the backward pass, the gradients of activations and weights are computed by backward propagation (BP) and weight update (WU), respectively. The activation gradients are calculated based on the weights and the output gradients, while the weight gradients are computed according to the stored activations and the output gradients. MatMul can be utilized to unify the computational flow of computationally intensive layers like convolutional layers of ResNet18 and linear layers of ViT in large-batch DNN training, boosting computational efficiency. Taking a convolutional layer as an example, Fig. 1 (b) demonstrates the *im2col* process [40], which converts a tensor into a matrix for subsequent DNN training operations. Fig. 1 (c) to (e) further illustrates how FF, BP, and WU for a single layer of DNN are transformed into MatMul through im2col process, respectively. Furthermore, we leverage PyTorch profiler [41] to dissect the execution time for training three typical DNNs with a batch size of 512 using an RTX 2080 Ti card. As shown in Fig. 2, those operations can be unified into MatMuls and constitute a considerable portion, up to 84%, of the training time per batch on average, significantly impacting the training process. By optimizing these MatMuls, significant improvements in training speed can be attained.



Fig. 2. Execution time profile results on training three DNN models with a batch size of 512 using an RTX 2080 Ti card. The dominance of MatMul operations in the training process highlights the significance of accelerating these operations to improve training efficiency.

B. Sparse Training for DNNs

Sparse training [3]-[12] is one of the promising techniques to reduce the training computational cost while retaining impressive accuracy of DNNs. A majority of prior arts mainly focus on ReLU-based [42], [43], structured [6], [7] or unstructured sparsity [8], [9] for sparse training. ReLU-based sparsity has no impact on model accuracy but suffers a low sparse ratio (approximately 50%) and can be difficult to be exploited on hardware. Structured sparsity has limitations in reducing computational complexity (less than 40% sparse ratio in [6]), while unstructured sparsity, despite saving a great number of operations (over 80% sparse ratio in [9]), is difficult to accelerate on hardware [20]. To achieve a better tradeoff between structured and unstructured sparsity, pre-defined sparsity [22], [23], has been exploited as the pioneering technique, where the sparse pattern is predetermined based on prior knowledge or heuristics. Recently, N:M fine-grained structured sparsity [21], demanding that at least N values must be zero for each group of M values, has attracted the attention of researchers [3], [26]–[32], [44] due to its practical sparsity ratio as well as its hardware-friendly pattern. As for N:M fine-grained sparse training, SR-STE [32] and SDGP [3] can boost training efficiency by forcing N:M sparsity on weights in the forward pass and output gradients in the backward pass, respectively. The sparsity introduced in one training direction hinders the further improvement of training efficiency. In this work, our BDWP leverages the N:M sparsity of weights in both forward and backward passes of DNN training, significantly reducing the number of operations and retaining competitive training accuracy compared to SR-STE and SDGP.

C. FPGA-based DNN Training Accelerators

Nowadays, DNN training is mainly accelerated on powerhungry GPU devices [3], [6] leading to low energy efficiency [37]. Additionally, many ASIC designs dedicated to DNN training, such as [45], achieve good energy efficiency but require lengthy design cycles. In contrast, building dedicated accelerators on FPGA enable agile deployment with satisfactory energy efficiency. Various DNN training accelerators [33], [34], [36]–[39], [46]–[49] have been developed on FPGA platforms. [33], [34], [36], [37] exploited optimization for standard DNN training process. However, with the increasing number of operations in developing DNNs, it is difficult to achieve satisfying speedup by simply optimizing dataflow or hardware design for the standard training process. Furthermore, [39], [49] exploit DNN sparse training acceleration on FPGA but suffer low computational efficiency due to leveraging irregular unstructured sparse patterns. In addition, [46]–[49] employed aggressive reduced numerical precision, such as FP9 or INT8, to decrease the computational cost of DNN training which is orthogonal to sparse DNN training. Compared to prior works, we aim to significantly improve DNN training efficiency by developing an FPGA-based accelerator, namely SAT, that enables both dense and computation-efficient N:M sparse operations within the N:M sparse DNN training process.



Fig. 3. Comparison of (a) conventional training, uni-directional *N:M* sparse training, including (b) SR-STE and (c) SDGP, and our proposed bidirectional *N:M* sparse training, i.e., (d) BDWP for a DNN layer. Without compromising convergent accuracy, the DNN training process using BDWP can significantly speed up due to aggressive *N:M* pruning in both forward and backward passes.

III. N:M SPARSE TRAINING ALGORITHM

N:M sparsity pattern can be leveraged to accelerate the DNN training process by significantly reducing the number of operations, which has been introduced through SR-STE [32] in the forward pass and SDGP [3] in the backward pass. In this section, we first learn the sensitivity of the training loss by introducing *N:M* sparse patterns during DNN training and then propose our BDWP based on our findings, which unifies *N:M* patterns in both forward and backward passes to further elevate DNN training efficiency.

A. Exploiting N:M Sparse Training Potential

As shown in Fig. 3, compared with the conventional training, SR-STE prunes weights in the forward pass, while SDGP prunes output gradients in the backward pass to reduce the number of required operations of DNNs. To evaluate the effectiveness of these *N:M* pruning techniques, we employ the from-scratch training loss as a metric and compare the errors of the pruned models with those of the densely trained models. Fig. 4 presents the loss curves when training from scratch ResNet9, ViT, and ResNet18 models on CIFAR-10, CIFAR-100, and Tiny ImageNet datasets, respectively. For ResNet9 on the simple CIFAR-10 dataset, both SR-STE and SDGP exhibit good performance compared to the dense baseline. However, for larger models or more complicated datasets



Fig. 4. Comparison of training loss using multiple N:M pruning methods for (a) ResNet9 on CIFAR-10, (b) ViT on CIFAR-100 dataset and (c) ResNet18 on Tiny ImageNet dataset.

like ViT on CIFAR-100 and ResNet18 on Tiny ImageNet, pruning activations in the backward pass using SDGP with a sparse ratio of about 75% results in a loss curve that deviates significantly from the dense training scheme. To address this issue, we explore an alternative approach by pruning weights in the backward pass, denoted as SDWP in Fig. 4. Notably, SDWP demonstrates better convergence compared to SDGP at the same N:M sparse ratio. Therefore, a novel bidirectional N:M sparse training approach called BDWP is proposed by integrating both unidirectional weight pruning techniques, i.e., SR-STE and SDWP. The training loss curve of BDWP in Fig. 4 closely aligns with SR-STE and SDWP at the same N:M sparse ratio, showing BDWP achieving negligible impact on training convergence with a significant reduction of training operations.

Algorithm 1 Training an L layer network using BDWP

Input: A mini-batch of input activations and labels (a_t^0, y_t) , current weights w_t , sparse ratio N and M at iteration t. **Output:** Updated weights w_{t+1} .

Forward Pass

- 1: for l = 1 to L do
- $$\begin{split} & \widetilde{w}_{\text{FF}}^{l} \leftarrow \text{BDWP}_{\text{FF}}(w_{t}^{l}, N, M). \\ & a_{t}^{l} \leftarrow \text{FF}(a_{t}^{l-1}, \widetilde{w}_{\text{FF}}^{l}). \end{split}$$
 2.
- 3:
- 4: end for
- 5: Compute the gradient of the output layer g_{aL} . **Backward Pass**
- 6: for l = L downto 1 do
- $\widetilde{w}_{BP}^{l} \leftarrow BDWP_{BP}(w_t^l, N, M).$ 7:
 $$\begin{split} g_{a_t^{l-1}} &\leftarrow \mathbf{BP}(g_{a_t^l}, \widetilde{w}_{\mathbf{BP}}^l). \\ g_{w_t^{l-1}} &\leftarrow \mathbf{WU}(a_t^{l-1}, g_{a_t^l}). \end{split}$$
 8:
- 9.
- 10: end for
- 11: Optimize w_{t+1} with momentum SGD.

B. Bidirectional Weight Pruning

Our N:M sparse training method, BDWP, is illustrated in Fig. 3 (d) and detailed in Algorithm 1.



Fig. 5. Applying BDWP in the convolutional layer and the linear layer in both forward and backward passes of DNN training. For the convolutional layer, BDWP is applied to each group across (a) input channels in the forward pass and (b) output channels in the backward pass, respectively. For the linear layer, BDWP is applied to each group across (c) input features in the forward pass and (d) output features in the backward pass, respectively.

Notation. Given a mini-batch of training samples a_t^0 and labels y_t , we aim to optimize weights w_t to w_{t+1} at iteration t for an L layer DNN. Activations and weights of the l-th layer are denoted as a_t^l and w_t^l , respectively. In addition, g_{a^l} and $g_{w_{1}^{l}}$ denote gradients with respect to activations and weights of the *l*-th layer, respectively. Weights of the corresponding sparse network are denoted with \widetilde{w} , and $\widetilde{w}_{\rm FF}$ and $\widetilde{w}_{\rm BP}$ denote the pruned N:M sparse weights of BDWP using in the forward pass and backward pass, respectively.

Training Flow. Algorithm 1 describes the process of BDWP when training an L layer network at the iteration t. BDWP_{FF} and $BDWP_{BP}$ denote the N:M element group generation process in the forward pass and backward pass, respectively. Both of them take the dense weights w as input and generate N:M sparse weights \widetilde{w} for further computation. In FF, as shown in Line 3, the activations perform operations with the N:M sparse $\widetilde{w}_{\rm FF}$, which have been slimmed for $\frac{M}{N}$ times. In BP, as shown in Line 8, the activation gradients are obtained by performing operations with output gradients and $\widetilde{w}_{\rm BP}$. The other steps of the training process stay the same as the standard training flow. For from-scratch training, N:M sparse patterns are leveraged from the first to the final training steps, and following SR-STE and SDGP, updated in every training step. Other training hyperparameters remain the same as for dense training.

N:M Element Group Generation. Fig. 5 presents how to apply BDWP in forward and backward passes to the convolutional layer and the linear layer, respectively, both of which dominate most of the required DNN training operations. BDWP preserves values with the N most significant magnitude in each group of M elements. N:M is assumed as 2:4 here, and B denotes batch size. Additionally, the subscripts *i* and *o* refer to the input activations and output gradients, respectively. When training a convolutional layer, the input activations are represented by a tensor with B batches, each having a height of H, width of W, and C_i channels, and the weights are denoted as a tensor with a height of K, width of K, C_i input channels, and C_o output channels.

BDWP removes pruned elements in each group across the input channels (C_i) in the forward pass, and across the output channels (C_o) in the backward pass, respectively. As for training a linear layer, the input activations are represented by a tensor with B batches, each having a width of D and F_i features, and the weights are represented by a transformation matrix from F_i to F_o . BDWP is applied to each group across input features (F_i) in the forward pass, respectively. The preserved elements in each group would be packed into the compact format as in [21] to reduce memory consumption.

Opportunities on Hardware Implementation. Given hardware accelerators supporting N:M sparsity are rather limited [20], [27], there are opportunities to develop new architectures to accelerate N:M sparse training with high performance and efficiency. For instance, generating N:M sparse weights during each iteration presents an opportunity for an on-chip hardware module capable of producing N:M sparse data. In addition, to accommodate the varying sizes of MatMuls required at different stages of the training process, a flexible interconnect capable of adapting to the changing demands is imperative. Finally, a unified and efficient computing unit is critical to enable the accelerator to handle both N:M sparse and dense operations with high computational efficiency.

IV. HARDWARE ARCHITECTURE

The efficient hardware architecture is crucial for achieving significant acceleration in DNN training through computational optimization resulting from N:M sparsity pattern. This section presents an efficient N:M sparse accelerator for DNN training, namely SAT, fully leveraging the computationefficient operations from N:M sparse training algorithms. We first briefly introduce the overall architecture of SAT and then elaborate on the designs of the major computing engines.



Fig. 6. The overall microarchitecture of SAT is composed of three computing engines, namely STCE, WUVE, and SORE, respectively.

A. Overall Architecture

As shown in Fig. 6, SAT consists of three major computing engines: 1) an N:M sparse tensor computing engine (STCE),

2) a weight update vector engine (WUVE), and 3) a sparse online reduction engine (SORE). STCE significantly boosts the computational efficiency of DNN training by efficiently unifying the MatMuls across FF, BP, and WU, and flexibly supporting both N:M sparse and dense operations in its processing elements. WUVE is a dedicated optimizer capable of updating weights through a mixed-precision scheme following NVIDIA Adaptive Mixed Precision (AMP) [50], which can significantly reduce off-chip memory access. SORE undertakes online N:M sparse reduction operations by taking dense weights with a group size of M as input and producing N:Msparse weights along with corresponding indexes as output. To improve the overall hardware performance, double-buffering is employed across all on-chip buffers to overlap the data transfer and computation.



Fig. 7. (a) The microarchitecture of unified *N:M* sparse processing element (USPE). Example of 2:4 USPE performing (b) 1:4 sparse, (c) 2:4 sparse, and (d) 2:2 dense dot-product operations.

B. Unified N:M Sparse Processing Element

STCE employs a systolic array composed of 32×32 unified *N:M* sparse processing elements (USPEs), which can be configured dynamically at runtime to perform *N:M* sparse-dense or dense-dense products. As depicted in Fig. 7 (a), USPE comprises a task counter, an FP16 multiplier, an FP16-to-FP32 switcher, and an FP32 adder, in addition to four register files. These register files serve as temporary storage for input data received from the west and north, input valid indexes received from the north, and output accumulated partial results to the south. During each cycle, the USPE is capable of multiplying two FP16 data and then adding its result to the input partial sum. Both multiplier and adder in the USPE are pipelined by 3 stages to improve computational efficiency.

USPE is flexible enough to support diverse types of dotproduct operations during DNN training. Fig. 7 (b)-(d) illustrate how the USPE performs dot-product operations across various N:M sparse and dense configurations. In order to accommodate varying N:M sparsity, the USPE utilizes a valueserial computing approach, which facilitates the folding of the dot-product operation for an N:M group into N cycles. For instance, a 2:4 USPE can execute a 1:4 sparse dot-product within a cycle, and a 2:4 sparse dot-product within two cycles. Additionally, we decompose dense MatMul into multiple 2:2 dense dot-products, which are then assigned to USPEs. Each USPE can perform a 2:2 dense dot-product within two cycles.



Fig. 8. Example of STCE performing a 2:4 sparse MatMul in the WS dataflow and a dense MatMul in the OS dataflow.

C. Flexible Systolic Interconnect

STCE improves on previous works that employed weightstationary (WS) [51]–[53] or output-stationary (OS) [27], [33], [37] systolic architectures by leveraging a flexible systolic interconnect that is capable of dynamically switching between WS and OS dataflows on the fly, providing increased mapping space for MatMul operations and enabling efficient MatMuls across various computing patterns in FF, BP, and WU stages.

Fig. 8 presents how STCE equipped with the flexible systolic interconnect performs a 2:4 sparse MatMul in WS dataflow and a dense MatMul in OS dataflow. In Fig. 8 (a), we present a case of 2:4 sparse MatMul, where the sparse matrix is compactly packed by preserving only the two most significant values in each 2:4 group, along with their corresponding indexes. When performing this operation in the WS dataflow, STCE first preloads the compact 2:4 weight groups to each USPE. The computation starts once the preload is complete. To accomplish the dot-product operation of a 2:4 sparse group, each USPE in STCE consumes two cycles, after which it transfers its data from the west to the east and data from the north to the south. Fig. 8 (c) illustrates the data transfer process every two times 2:4 sparse group computation tasks. Due to the removal of pruned weight elements, STCE performs fewer operations in comparison with the dense task, thereby improving computational efficiency. Fig. 8 (b) is a case of dense MatMul. When a dense MatMul is performed in OS dataflow, STCE with 2:4 USPEs streams the two input dense matrices from the west and north directions, respectively. Every two cycles, a USPE performs 2:2 dense dot-product operations, and the computed data from the west is transferred to the east, while the data from the north is passed to the south. The data transfer process is shown in Fig. 8 (d), which illustrates the exchange of data every two times 2:2

dense group computation tasks. The high utilization of USPEs during the computation stage enables STCE to achieve high computational efficiency for dense operations as well. Finally, STCE sequentially pops its accumulation results to the south upon the computation is finished.

D. Hardware Costs of STCE Enabling N:M Sparse Operations

To support N:M sparse operations, STCE requires additional logic to support sparse decoding. It also requires more registers to support the storage overheads caused by the significant increase in input bandwidth in N:M sparse MatMul compared to dense MatMul. As shown in Fig. 8, for 2:4 STCE, each USPE requires 4 registers to store data from the west in the sparse mode, while in the dense mode, only two registers need to be enabled to complete the dense operations. In this case, the two disabled registers are the additional hardware overhead compared to a dense systolic array at the same scale. When enabling higher sparse ratios such as 2:8 and 2:16, the 2:4 STCE cannot directly implement 2:8 or 2:16 sparse operations, and needs to be reconfigured on FPGA. At higher N:M sparse ratios, the register overhead per USPE in STCE will continue to grow, which may lead to disproportionate hardware costs, and the improvement in training accuracy may not be able to offset the continuous increase in hardware costs. Therefore, the selection of N:M sparsity is a trade-off between model accuracy and hardware cost.

E. Weight Update Vector Engine

As depicted in Fig. 6, WUVE serves as a dedicated optimizer that employs momentum stochastic gradient descent (SGD) to update weights using the mixed precision scheme of NVIDIA AMP [50]. To update master parameters for the next training iteration, WUVE takes weight gradients in FP16 format and other master parameters in FP32 format as input. It specifically elevates the numerical precision of weight gradients from FP16 to FP32 to minimize quantization errors in the WU step. Moreover, WUVE provides 32 parallel lanes to improve computational efficiency, and each lane consists of three FP32 multipliers, two FP32 adders, one FP16-to-FP32 switcher, and one FP32-to-FP16 switcher.

F. N:M Sparse Online Reduction Engine

Dedicated accelerators [20], [27], designed for N:M sparse inference acceleration, utilize N:M sparse weights that are generated offline prior to inference. However, during N:Msparse training, weights, activations, and gradients are varied in every iteration, leading to dynamic updates of the preserved elements and their corresponding indexes in an N:M element group, as illustrated in Fig. 9. Therefore, to handle the dynamic updates of N:M sparse elements during training, the dedicated module for N:M sparse online reduction, namely SORE, is designed to enable the efficient generation of compact groups of N:M sparse elements with their associated indexes.

There are 32 parallel lanes in SORE, and each lane consists of a top-K sorter and a data provider. The top-K sorter sequentially receives dense data in a group with a size of M,



Fig. 9. Example of the processing process of a 2:4 SORE. It generates 2:4 sparse groups with corresponding indexes in parallel using four cycles, starting with a four-element group as input.

and after M cycles, the K data sorted in the top-K sorter with their indexes in the group are passed to the data provider. The data provider, which is configurable to support all N not larger than K, sequentially outputs the top-K elements in a group. Fig. 9 takes a 2:4 SORE as an example for illustration. A 4-element group is sequentially provided to the top-K sorter as input, and a 2:4 sparse group with corresponding indexes is generated in parallel to the data provider after 4 cycles.

V. DATAFLOW OPTIMIZATION

SAT is capable of effectively supporting *N*:*M* sparse DNN training through its innovative design of STCE, which includes unified processing elements and a flexible interconnect, as well as SORE, which efficiently generates *N*:*M* sparse data groups. To further optimize the computational efficiency of SAT, we introduce several dataflow optimization methods to improve the utilization of STCE and the efficiency of SORE.

A. Interleave Mapping of USPE

To reduce the critical path of STCE and improve the operating frequency of SAT, both the multiplier and adder in USPE are deeply pipelined. However, when STCE is configured as OS dataflow computing mode, there is an accumulation loop in USPE as shown in Fig. 10 (a). This causes a computation stall when a dot-product operation is mapped to USPE, since the partial sums generated during dot-product operations are held in the pipelines until they reach the output stage. As a result, the next dot-product operation has to wait until the previous one has cleared the pipeline, resulting in a three-cycle latency as shown in Fig. 10 (b).

By contrast, we propose an interleave mapping method for USPE that allows for the simultaneous processing of computationally independent operations during the accumulation loop. This helps to minimize the stall and improve the computational efficiency of USPE. As shown in Fig. 10 (c), three parallel dot-product operations are interleaved and fed into USPE, effectively filling up the pipelines. By leveraging the proposed interleaving mapping method, USPE can achieve $3 \times$ throughput improvement when employing OS dataflow.



Fig. 10. (a) The loop in USPE when employing OS dataflow slows down the computational efficiency of USPE using (b) the conventional systolic mapping strategy. By contrast, (c) the proposed interleave mapping strategy can improve 3x throughput by improving the utilization of USPE.

B. Pre-generation of N:M Sparse Elements

Pre-generation technique for *N:M* sparse weights is proposed to boost the efficiency of *N:M* sparse training with NVIDIA AMP [50]. AMP is a training scheme that optimizes the precision of the arithmetic operations used during training in DNNs. Fig. 11 (a) presents the computational steps to train a convolutional layer using AMP. It can perform the forward pass with half-precision (FP16) arithmetic and accumulate and convert the gradients back to the original precision (FP32) before updating the weights.



Fig. 11. Weight dataflow schedule using (a) conventional AMP training, (b) sparse AMP training of BDWP with N:M sparse generation in FF and BP, and (c) sparse AMP training of BDWP with N:M sparse pre-generation in WU. Compared with (b), (c) saves external memory space, memory access bandwidth, and execution time at a high N:M sparse ratio.

BDWP is taken as an example to integrate N:M sparse training into AMP. As shown in Fig. 11 (b), N:M sparse data can be generated after the weight tiles are loaded in FF and BP, and the computational cost is significantly reduced by skipping zero-value operations. However, this generation process can slow down computational efficiency, as the MatMul must wait for the generation of N:M sparse weights. Fig. 11 (c) illustrates the working flow of our proposed pre-generation technique for N:M sparse weights, which can improve the computational efficiency and storage requirement of N:M sparse training. In WU stage, the FP32 weight updates are calculated in WUVE and then directly sent to SORE for N:M sparse compression to obtain FP16 sparse weights. This process is fine-grained pipelined, thus achieving the overlap of computation and storage. In contrast, in Fig. 11 (b), dense FP16 weights must be loaded from external memory and sent to SORE to obtain sparse weights before they can be sent to STCE for MatMul

computation, which cannot achieve the overlap and affects the overall computation efficiency. In addition, compared to Fig. 11 (b), Fig. 11 (c) can save the bandwidth requirement by storing and loading the *N:M* sparse weights. It requires to store for the *N:M* weights $\tilde{w}_{\rm FF}$ and $\tilde{w}_{\rm BP}$ from its input and output channels. When its sparse ratio is higher than 50%, the storage cost is also significantly lower than the conventional AMP training.



Fig. 12. Enabling N:M sparse DNN training on SAT with offline dataflow scheduling, which first transforms the DNN model into the MatMul format, and then generates per-layer configuration words for the three training stages.

C. Offline Dataflow Scheduling

Fig. 12 illustrates the implementation of *N:M* sparse DNN training on SAT through flexible offline dataflow scheduling. It involves transforming the DNN model into the MatMul format, and generating per-layer configuration words for the three training stages. The reconfiguration word generator (RWG) takes the MatMul format of DNNs in three training stages as input and generates per-layer configuration words based on the selected *N:M* sparse training method and the *N:M* sparse ratio. During training acceleration, SAT's controller fetches each layer's reconfiguration words gradually at the FF, BP, and WU stages and produces corresponding control signals for the other components.

RWG is the key component for improving the training throughput of SAT. Fig. 12 presents how RWG produces reconfiguration words for ResNet18 when using three N:M sparse training methods: SR-STE, SDGP, and BDWP enabled with 2:8 sparse ratio. RWG first assigns N:M sparse mode for FF, BP, and WU these three training stages based on the user-configured N:M sparsity (e.g., 2:8) and the selected sparse training method. For example, for SR-STE, RWG will determine the FF stage of the network layer as 2:8 sparse training, and the BP and WU stages as dense training. For BDWP, which introduces N:M sparsity in both directions, RWG will determine the FF and BP stages as 2:8 sparse training, and WU as dense training. Next, RWG allocates the method of generating N:M sparse data. The pre-generation of N:M sparse elements in the WU stage is prioritized due to its significant advantages. If pre-generated features are not available, the corresponding layers need to generate N:M sparse elements during the FF and BP computing stages. For example, SR-STE and BDWP, which allow the pre-generation of sparse weights,

prompt the RWG to schedule SORE within the WU stage. Conversely, SDGP, which prunes input gradients during the BP stage, requires RWG to schedule SORE within the BP stage. Finally, RWG predicts the computational utilization of SAT based on the scale of transformed MatMul of each network layer in advance, so that it can arrange the superior dataflow and the output data layout rules for each layer. As shown in Fig. 12, for BDWP, within the 4th layer, RWG calculates the hardware utilization of OS and WS in the FF, BP, and WU phases, and based on predicted results, the OS, OS, and WS dataflows are allocated to the three phases, respectively. By assigning the above three phases, RWG effectively improves the computational utilization of each network layer, thereby significantly improving the training throughput of SAT.

TABLE I FROM-SCRATCH TRAINING SETUP OF EVALUATED DNNS

Model	Dataset	Optimizer	$\mathbf{E}\mathbf{P}^{\dagger}$	\mathbf{BS}^{\dagger}	LR^{\dagger}	WD
ResNet9	CIFAR-10	Momentum SGD	150	512	0.5	5e-4
ViT	CIFAR-100	Momentum SGD	150	512	0.1	5e-4
VGG19	CIFAR-100	Momentum SGD	150	512	0.1	5e-4
ResNet18	Tiny ImageNet	Momentum SGD	88	512	0.05	5e-3
ResNet50	ImageNet	Momentum SGD	120	256	0.1	5e-5

[†] EP: epochs; BS: batch size; LR: initial learning rate; WD: weight decay.

VI. EXPERIMENTAL RESULTS

A. Experimental Setup

Benchmarks. To evaluate the algorithmic and hardware performance of our proposed *N:M* sparse training scheme, we choose five typical DNN models on four popular datasets for from-scratch training. The details are shown in Table I. The ResNet9, ResNet18, ResNet50 [54] and VGG19 [55] are the conventional convolutional neural networks (CNNs), and the vision Transformer (ViT) [56] is a novel DNN architecture utilizing the attention mechanism.

Software Implementation. We train the DNN benchmarks with the hyper-parameter settings shown in Table I using PyTorch v1.10 with mixed-precision training support [50]. Note that, in each training iteration, *N:M* sparsity is applied to all convolutional layers except for the first layer of evaluated CNNs, and all linear layers in the Transformer blocks of ViT. Excluding the first convolutional layer from *N:M* sparsity aligns with SDGP's experimental setup [3] and quantization-related practices [57] due to the first layer's sensitivity to accuracy impacts arising from its limited input channels. *N:M* sparse patterns are introduced since the first training step and kept updated for every iteration until the end of training.

Hardware Implementation. SAT is implemented in SystemVerilog with the help of hardware components from the PULP platform [58] and the BaseJump standard template library [59]. We evaluate hardware performance using a Xilinx Virtex UltraScale+ VCU1525 card with an XCVU9P FPGA, with Xilinx Vivado 2018.2 at a clock frequency of 200 MHz. We measure power consumption using the Xilinx Power Estimator tool. Moreover, we validate speed performance using a cycle-accurate performance model that is cross-validated with RTL simulation results following methods in [33], [60]. The memory accesses to external memory are also considered.

							ResNet9 on CIFAR-10			VGG19 on CIFAR-100		ViT on CIFAR-100		ResNet18 on Tiny ImageNet		ResNet50 on ImageNet		eNet
Method	Pat.	Spa FW	irsity in &BW	$\begin{vmatrix} \text{Train.} \\ \text{FLOPS} \\ (\times 10^{16}) \end{vmatrix}$	Infer. FLOPS $(\times 10^9)$	Top-1 Acc. (%)	$\begin{vmatrix} \text{Train.} \\ \text{FLOPS} \\ (\times 10^{15}) \end{vmatrix}$	Infer. FLOPS $(\times 10^8)$	Top-1 Acc. (%)	Train. FLOPS (×10 ¹⁶)	Infer. FLOPS $(\times 10^8)$	Top-1 Acc. (%)	Train. FLOPS (×10 ¹⁶)	Infer. FLOPS $(\times 10^9)$	Top-1 Acc. (%)	$\begin{vmatrix} \text{Train.} \\ \text{FLOPS} \\ (\times 10^{18}) \end{vmatrix}$	Infer. FLOPS $(\times 10^9)$	Top-1 Acc. (%)
Baseline	-	×	x	2.62	1.16	95.27	9.00	4.00	72.23	1.45	6.43	60.81	4.82	1.83	65.46	1.91	4.14	76.72
[32]	2:4	~	x	2.19	0.59	95.18	7.52	2.02	73.04	1.22	3.36	60.37	4.07	0.98	65.43	1.61	2.16	76.52
[3]	2:4	×	~	2.19	1.16	95.11	7.52	4.00	72.01	1.22	6.43	57.67	4.07	1.83	64.99	1.61	4.14	N/A
BDWP	2:4	~	~	1.75	0.59	95.10	6.03	2.02	72.21	0.99	3.36	60.85	3.33	0.98	65.40	1.30	2.16	76.80
[32]	2:8	~	x	1.97	0.30	95.18	6.78	1.03	72.78	1.10	1.83	59.55	3.70	0.55	65.04	1.45	1.17	75.88
[3]	2:8	×	~	1.97	1.16	95.18	6.78	4.00	71.25	1.10	6.43	46.10	3.70	1.83	62.40	1.45	4.14	N/A
BDWP	2:8	~	~	1.32	0.30	95.18	4.55	1.03	72.32	0.76	1.83	59.60	2.58	0.55	65.14	1.00	1.17	75.44
[32]	2:16	~	x	1.86	0.15	95.15	6.40	0.53	72.18	1.04	1.06	55.72	3.52	0.34	63.75	1.38	0.67	74.75
[3]	2:16	×	~	1.86	1.16	94.91	6.40	4.00	69.95	1.04	6.43	38.37	3.52	1.83	48.00	1.38	4.14	N/A
BDWP	2:16	~	~	1.10	0.15	95.16	3.80	0.53	72.04	0.64	1.06	55.70	2.21	0.34	63.94	0.84	0.67	74.24

 TABLE II

 Accuracy Comparison Using Various N:M Sparse Training Schemes

B. Algorithmic Performance

We compare the Top-1 accuracy of BDWP against the other state-of-the-art *N:M* sparse training methods, including SDGP [3] and SR-STE [32]. To evaluate the robustness of these sparse training methods, we inherit the hyperparameter settings of the baseline to BDWP, SDGP, and SR-STE. Notably, all experiments are repeated three times for reliability, except for ResNet50 on ImageNet, which has a single run due to limited available computational resources.

Table II shows how the different N:M sparse training methods affect the convergent model accuracy. 'Pat.' is short for N:M sparsity pattern, while 'FW' and 'BW' refer to introduced N:M sparse patterns from the forward and backward passes, respectively. In most cases, BDWP results in the best performance with the lowest training FLOPS compared to SDGP and SR-STE, indicating BDWP is an effective method to reduce computational costs without sacrificing accuracy. As shown in Table II, with a 2:8 sparse ratio, BDWP achieves an average theoretical computational reduction of 1.93× across evaluated training tasks compared to the dense training scheme. The significant reduction of computational operations is coupled with negligible impact on model convergence accuracy, showing an average loss of only 0.56%. Moreover, the required number of operations for inference significantly reduces by $3.54 \times$ on average.

Fig. 13 illustrates the impact of various N:M sparse ratios on model convergent accuracy when employing BDWP sparse training. Experimental results show that the larger the N:M sparse ratio, the less likely the model is to overfit the dataset. For example, ResNet9 on CIFAR-10 is less prone to overfitting, so it can tolerate a higher N:M sparse ratio without losing too much accuracy. However, for models that are less prone to overfitting on selected datasets, such as ViT, ResNet18, and ResNet50, a higher N:M sparse ratio, up to 87.5%, may result in a slight decrease in accuracy. This is because these models are less tolerant of the loss of representation capability due to sparsity. Additionally, the impact of M on model accuracy varies depending on the level of sparsity. For instance, at lower sparsity levels, like 50%, the choice of M may not have a significant impact on accuracy at the same N:M sparsity ratio. On the other hand, for higher sparsity ratios, up to 87.5%, a larger *M* can provide more flexible pattern choices, leading to better accuracy performance. For example, when comparing the ViT, ResNet18, and ResNet50 models, it has



Fig. 13. Impact of various N:M sparse ratios on model convergent accuracy when employing BDWP sparse training.

been observed that the convergence accuracy of a 2:16 sparse model with a sparsity ratio of 87.5% is higher than that of a 1:8 sparse model. Therefore, the choice of M should be carefully considered when designing N:M sparse models to achieve optimal accuracy and sparsity trade-offs.

C. Hardware Resource Consumption

To precisely understand the hardware overhead of STCE, we conduct an experiment using a 4×4 dense systolic array as a baseline, along with 4×4 STCEs under various *N:M* sparse configurations. Additionally, for a fair comparison, we implement the other baseline systolic arrays with the same throughput of *N:M* STCEs. To support sparse *N:M* operations, STCE requires additional LUT overhead for supporting sparse indexes, additional FF overhead for storing *N:M* data groups,



Fig. 14. Hardware resource comparison between multiple dense systolic arrays and STCE of various N:M sparse ratios.

and corresponding power overhead compared to the dense baseline. Experimental results are shown in Fig. 14. Compared to the 4×4 dense baseline, 2:4, 2:8, and 2:16 STCEs increase the LUT overhead by $1.1\times$, $1.2\times$, and $1.3\times$, respectively, while the FF overhead increases significantly by $1.7\times$, $2.2\times$, and $3.3\times$. However, these additional hardware costs are highly worthwhile when comparing STCE with the dense systolic arrays operating at the same throughput scale. As shown in Fig. 14, 2:8 STCE has significantly lower hardware overheads than 4×16 dense systolic array, with $3.4\times$ lower LUT, $2.0\times$ lower FF, $4.0\times$ lower DSP, and $3.1\times$ lower power consumptions, which shows that STCE is a promising architecture for performing sparse *N:M* operations.

 TABLE III

 SAT RESOURCE BREAKDOWN ON XILINX VCU1525

Component	Logic	Registers	Memory blocks	DSP blocks
STCE	389K	589K	0	1024
WUVE	40K	20K	0	192
SORE	3K	5K	0	0
Input Buffer (W2E)	0	0	128	0
Input Buffer (N2S)	0	0	38	0
Output Buffer (N2S)	0	0	38	0
Optimizer Buffer	0	0	64	0
Others	257K	358K	443	12
Total	689K (58%)	972K (41%)	711 (23%)	1228 (18%)

Based on the comprehensive trade-off from the algorithm and hardware perspectives, we select a 2:8 sparsity pattern in the following hardware implementation of SAT. Table III presents the resource consumption, where the 'others' row includes DDR4 controller, PCIe DMA, the interconnect from DDR to SAT, and other auxiliary components. Due to the adoption of 2:8 sparse patterns, the number of memory banks for W2E buffer needs to be expanded to four times that of N2S buffer, resulting in the use of 128 banks. Additionally, N2S buffer requires additional storage space to store sparse indexes, and therefore a total of 38 banks are used for both N2S input and output buffers. Furthermore, the optimizer buffer needs to store weight update parameters for 64 banks.

As shown in Table III, STCE dominates DSP consumption of SAT since it is the computing core for computational intensive MatMuls that transformed from convolutional or linear layers. STCE also takes the majority of the register consumption since there are multiple pipelines in USPEs to shorten the critical path and improve computational throughput. Furthermore, SORE, the sparse online reduction engine, is an area-efficient hardware solution for enabling N:M sparse online reduction capability in SAT, as it consumes less than 1% of the resources utilized by STCE.

D. Training Efficiency

To evaluate the training efficiency of the proposed training scheme composed of BDWP and SAT, Time-To-Accuracy (TTA) metric [3], [61] is used in comparison with other training methods, including conventional dense training, SDGP, and SR-STE. SAT enables 2:8 sparse acceleration for SDGP, SR-STE, and BDWP. As depicted in the upper part of Fig. 15, SAT with 2:8 BDWP training achieves an average of 46% reduction in single-batch training times compared to dense training, which corresponds to a significant $1.82 \times$ speedup per batch. Furthermore, the introduction of sparsity during training can impact the speed of model convergence, affecting the overall acceleration of sparse training. To ensure a fair comparison, the lower part of Fig. 15 shows the model convergence curves during 2:8 BDWP sparse training compared to dense training on SAT, where training time is normalized by the required time for a training epoch of BDWP. It reveals an average practical speedup of $1.75 \times$, which highlights the combined contribution of the sparse training algorithm and the hardware accelerator, with the algorithm reducing the computation and the hardware accelerating the process.

To provide a clear picture of the runtime breakdown, Fig. 16 presents the running time per batch of BDWP sparse training for each N:M sparse convolutional layer in ResNet18 on Tiny ImageNet with a batch size of 512. Note that we purposely did not overlap the memory access and computing cores during our analysis. In actual deployment, the running time of memory access and computing can be significantly reduced through the use of double buffering techniques. In Fig. 16, STCE's running time for FF and BP, which enable 2:8 sparse computing, is significantly lower compared to that for WU, by approximately a quarter of that required for dense computing. Additionally, WUVE and SORE exhibit a low activation frequency and short latency in task completion, respectively, consuming only a negligible fraction of the total running time. Overall, the SAT accelerator shows promise in enabling highly computation-efficient DNN training, with SORE's low resource consumption and running time overhead, coupled with significantly reduced number of operations through N:M sparse patterns in FF and BP.

E. Comparison with CPU and GPU

To evaluate the potential of SAT for efficient DNN training, we compare it against CPU and GPU platforms. The CPU baseline is the Intel Core i9-9900X processor, equipped with 19.25 MB L3 cache, 10 physical cores, 20 threads running at 3.50 GHz, and a thermal design power (TDP) of 165 W. Meanwhile, the GPU baselines are the NVIDIA RTX 2080 Ti card and Jetson Nano. The former achieves a peak throughput of 76 TFLOPS equipped with 4352 CUDA cores running at 1.35 GHz and a TDP of 250 W, and the latter is



Fig. 15. Required training time of SR-STE, SDGP, and our BDWP on SAT for ResNet9, ViT, VGG19, ResNet18, and ResNet50, respectively.



Fig. 16. Layer-wise running time per batch of 2:8 sparse training with BDWP in ResNet18 on Tiny ImageNet with a batch size of 512.

a competitive candidate for energy-efficient edge computing scenarios with a peak throughput of 472 GFLOPS. As shown in Table IV, the peak throughput of SAT can achieve 409.6 GOPs for dense operations, which closely aligns with Jetson Nano's performance, and achieve 1638.4 GOPs for 2:8 sparse operations.

Energy efficiency across CPU, GPUs, and SAT is evaluated to highlight the potential in *N:M* sparse training. The Intel performance counter monitor utility [62] is used to measure the actual CPU power consumption. We use *nvidia-smi* on RTX 2080 Ti and *jtop* on Jetson Nano to measure the GPU runtime power. For fair comparison against SAT, we use PyTorch v1.10 to perform convolutional layers that have been arranged in MatMul form in ResNet18 on both CPU and GPU with a batch size of 512. As presented in Fig. IV, SAT achieves $8.42 \times$ energy efficiency improvement compared to the CPU baseline. Moreover, compared to Jetson Nano and RTX 2080 Ti, SAT improves energy efficiency by $1.72 \times$ and $1.53 \times$, respectively.

To make a fair comparison with the RTX 2080 Ti, we scale

TABLE IV PERFORMANCE COMPARISON OF SAT VERSUS CPU AND GPU

	CPU	GPU	GPU	FPGA
Platform	Intel i9-9900X	NVIDIA Jetson Nano	NVIDIA RTX 2080 Ti	Xilinx XCVU9P
Frequency	3.50 GHz	921 MHz	1.35 GHz	200 MHz
Model		ResNet18	(Batch Size $= 51$	2)
Precision	FP32	FP32+FP16	FP32+FP16	FP32+FP16
Bandwidth (GB/s)	57.6	25.6	616	25.6
Latency (s)	12.91	61.28	1.72	11.98
Power (W)	165.00	7.54	238.36	22.38 (avg.) 20.73 (dense) 24.15 (2:8 sparse)
Peak Throughput (GFLOPS)	2240	472	76000	409.6 (dense) 1638.4 (2:8 sparse)
Runtime Throughput (GFLOPS)	423.69	94.66	3372.52	484.21 (avg.) 280.31 (dense) 702.54 (2:8 sparse)
Energy Efficiency (GFLOPS/W)	2.57	12.56	14.15	21.64 (avg.) 13.52 (dense) 29.09 (2:8 sparse)

SAT by changing the number of USPEs in STCE and the off-chip bandwidth, while keeping other factors unchanged. The experimental results are shown in Fig. 17, where the X-axis represents the systolic array size of STCE in SAT. It can be seen that the number of USPEs and the off-chip bandwidth have a significant impact on the runtime throughput of training. When the off-chip bandwidth is 409.6 GB/s, as shown in Fig. 17 (c), which is less than the 616 GB/s of the RTX 2080 Ti GPU as shown in Table IV, the runtime throughput of SAT executing 2:8 BDWP reaches 3.9 TOPS, which is greater than the runtime throughput of the RTX 2080 Ti GPU in training ResNet18 (only 3.4 TOPS). In addition, the dense peak performance of 2:8 SAT under this configuration is 6.6 TOPS, and the sparse peak performance of 2:8 SAT is 26.2 TOPS, which is also significantly less than the 76 TOPS of the RTX 2080 Ti GPU. This shows that SAT after scaling has a higher computational utilization for training a ResNet18.

Accelerator	Platform	Network	Precision	DSP Util.	Freq. (MHz)	Power (W)	Throughput (GOPS)	Comp. Effi. (GOPS/DSP)	Energy Effi. (GOPS/W)
SAT (this work)	XCVU9P	ResNet-18	FP16+FP32	1228	200	22.38	484.21	0.39	21.64
TODAES'22 [34]	ZCU102	VGG-16	FP32	1508	100	7.71	46.99	0.03	6.09
FPGA'20 [35]	Stratix 10	AlexNet	FP32	1796	253	N/A	$\sim \! 24.00$	0.01	N/A
FPT'17 [36]	ZU19EG	LeNet-10	FP32	1500	200	14.24	86.12	0.06	6.05
ICCAD'20 [33]	Stratix 10 MX	VGG-like	FP16	1046	185	$\sim \! 20.00$	$\sim \! 158.54$	0.15	~ 9.00
OJCAS'23 [39]	ZCU104	AlexNet	BFP16	1285	200	6.44	102.43	0.08	15.90
AICAS'21 [38]	XC7Z100	FC	INT16	64	150	2.50	19.20	0.30	7.68
FPL'19 [37]	Stratix 10 GX	VGG-like	INT16	1699	240	20.60	163.00	0.09	7.90
FPL'19 [49]	XCVU9P	AlexNet	FP9	1106	200	75.00	375.61	0.34	5.00
ISVLSI'21 [46]	VC709	VGG-like	INT8	2324	200	16.27	771.00	0.33	47.38
JOS'20 [47]	XCVU9P	VGG-like	INT8	4202	200	13.50	1417.00	0.34	104.96
TNNLS'22 [48]	VC709	VGG-16	PINT8	1728	200	8.44	610.98	0.35	72.37

 TABLE V

 Comparison of Prior FPGA-based Training Accelerators



Fig. 17. Runtime training throughput of ResNet18 with different available off-chip memory bandwidths when scaling the number of USPEs in STCE.

F. Comparison with FPGA-based Training Accelerators

Table V presents a comparison between SAT and existing state-of-the-art FPGA-based training accelerators for DNNs. Our SAT outperforms other architectures equipped with FP16 or higher numerical formats, exhibiting superior performance in terms of throughput, computational efficiency, and energy efficiency. Specifically, SAT improves the training throughput by $2.97 \sim 25.22 \times$, computational efficiency by $1.3 \sim 39 \times$, and energy efficiency by $1.36 \sim 3.58 \times$ when compared to [33]-[39]. The superior results of SAT can be attributed to its efficient hardware implementation of N:M sparsity acceleration. First, by exploiting the parallelism offered by large batch sizes, SAT can significantly increase the throughput of the training process, which is a significant improvement over prior FPGA-based accelerators [33], [37] that use small batch sizes for DNN training. Second, a more efficient dataflow design is adopted to efficiently cover the data loading and computation process, leading to high throughput and energy efficiency of SAT. Third, we exploit 2:8 sparsity in the forward and backward training processes, which significantly reduces the number of training operations by 48% on average, leading to improvements in throughput and energy efficiency. By leveraging the potential of N:M sparsity acceleration on hardware, SAT presents a novel approach to efficient DNN training that is orthogonal [12] to prior works that employ reduced numerical precision [46]–[49]. Our results demonstrate that SAT is a promising FPGA-based accelerator for DNN training that significantly outperforms existing state-of-the-art solutions, highlighting the potential of N:M sparse acceleration for efficient DNN training.

G. Discussion

The effectiveness of N:M sparse DNN training has been demonstrated both at the algorithm and hardware levels. From an algorithm perspective, BDWP achieves significant computational reduction without sacrificing model accuracy compared to other state-of-the-art N:M sparse training methods. Moreover, BDWP can be easily integrated with AMP training pipeline. From a hardware perspective, SAT is efficient in supporting N:M sparse DNN training. SORE incurs less than 1% hardware overhead, and STCE supports flexible dataflows, as well as both regular dense and computation-efficient N:M sparse operations, significantly improving DNN training efficiency. Deployment of BDWP on SAT reduces training time by 43%, resulting in $1.75 \times$ training acceleration compared to dense training. Additionally, SAT outperforms CPU and GPU in energy efficiency and also shows significant improvements over prior state-of-the-art FPGA-based training accelerators. These results demonstrate the proposed N:M sparse training scheme is particularly promising for achieving efficient and rapid training for increasingly large DNN models.

VII. CONCLUSION

In this paper, we present an efficient *N:M* sparse DNN training scheme on FPGA exploiting optimizations of algorithm, architecture, and dataflow aspects. At the algorithm level, a novel bidirectional weight pruning method, dubbed BDWP, is first proposed to significantly reduce the number of operations while maintaining model accuracy. At the architecture level, a sparse accelerator for DNN training, namely SAT, is further developed to support computation-efficient *N:M* sparse operations besides the regular dense operations efficiently. At the dataflow level, multiple optimization techniques further increase hardware utilization, improving the throughput of SAT. Experimental results show our *N:M* sparse training scheme can significantly improve the training throughput by $2.97 \sim 25.22 \times$ and the energy efficiency by $1.36 \sim 3.58 \times$ compared to state-of-the-art FPGA training accelerators. As the computations involved in DNN training are rapidly increasing, this work should be helpful for developing efficient sparse DNN training.

ACKNOWLEDGMENT

The authors would like to sincerely thank their reviewers for the valuable feedback.

REFERENCES

- W. Fedus *et al.*, "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity," *Journal of Machine Learning Research (JMLR)*, vol. 23, no. 120, pp. 1–39, 2022.
- [2] T. Brown et al., "Language Models Are Few-shot Learners," Advances in Neural Information Processing Systems (NeurIPS), vol. 33, pp. 1877– 1901, 2020.
- [3] B. McDanel et al., "Accelerating DNN Training with Structured Data Gradient Pruning," in 2022 26th International Conference on Pattern Recognition (ICPR). IEEE, 2022, pp. 2293–2299.
- [4] F. Zhang et al., "XST: A Crossbar Column-wise Sparse Training for Efficient Continual Learning," in 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2022, pp. 48–51.
- [5] Z. Song et al., "Approximate Random Dropout for DNN Training Acceleration in GPGPU," in 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2019, pp. 108–113.
- [6] X. Yuan et al., "Growing Efficient Deep Networks by Structured Continuous Sparsification," in 9th International Conference on Learning Representations (ICLR), 2021.
- [7] T. Zhang et al., "StructADMM: Achieving Ultrahigh Efficiency in Structured Pruning for DNNs," *IEEE Transactions on Neural Networks* and Learning Systems (TNNLS), vol. 33, no. 5, pp. 2259–2273, 2021.
- [8] X. Ye et al., "Accelerating CNN Training by Pruning Activation Gradients," in European Conference on Computer Vision (ECCV). Springer, 2020, pp. 322–338.
- [9] U. Evci et al., "Rigging the Lottery: Making All Tickets Winners," in International Conference on Machine Learning (ICML). PMLR, 2020, pp. 2943–2952.
- [10] Y. Wu et al., "Enabling On-Device CNN Training by Self-Supervised Instance Filtering and Error Map Pruning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 11, pp. 3445–3457, 2020.
- [11] S. Hassantabar et al., "SCANN: Synthesis of Compact and Accurate Neural Networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 9, pp. 3012–3025, 2021.
- [12] J.-H. Park et al., "Quantized Sparse Training: A Unified Trainable Framework for Joint Pruning and Quantization in DNNs," ACM Transactions on Embedded Computing Systems (TECS), vol. 21, no. 5, pp. 1–22, 2022.
- [13] G. Yuan et al., "TinyADC: Peripheral Circuit-aware Weight Pruning Framework for Mixed-signal DNN Accelerators," in 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2021, pp. 926–931.
- [14] Y. Liang *et al.*, "OMNI: A Framework for Integrating Hardware and Software Optimizations for Sparse CNNs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 40, no. 8, pp. 1648–1661, 2020.
- [15] F. Tu et al., "SDP: Co-Designing Algorithm, Dataflow, and Architecture for In-SRAM Sparse NN Acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 1, pp. 109–121, 2022.
- [16] F. Li et al., "FSA: A Fine-Grained Systolic Accelerator for Sparse CNNs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 11, pp. 3589–3600, 2020.
- [17] B. Liu et al., "Search-Free Inference Acceleration for Sparse Convolutional Neural Networks," *IEEE Transactions on Computer-Aided Design* of Integrated Circuits and Systems (TCAD), vol. 41, no. 7, pp. 2156– 2169, 2021.

- [18] Y. Jung et al., "Energy-Efficient CNN Personalized Training by Adaptive Data Reformation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 1, pp. 332–336, 2022.
- [19] H. Wang et al., "A Low-latency Sparse-Winograd Accelerator for Convolutional Neural Networks," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE, 2019, pp. 1448–1452.
- [20] NVIDIA, "NVIDIA A100 Tensor Core GPU Architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/ nvidia-ampere-architecture-whitepaper.pdf, 2020.
- [21] A. Mishra et al., "Accelerating Sparse Deep Neural Networks," arXiv preprint arXiv:2104.08378, 2021.
- [22] S. Kundu *et al.*, "Pre-defined Sparsity for Low-Complexity Convolutional Neural Networks," *IEEE Transactions on Computers (TC)*, vol. 69, no. 7, pp. 1045–1058, 2020.
- [23] W. Niu et al., "PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-based Weight Pruning," in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020, pp. 907–922.
- [24] E. Ozen et al., "Unleashing the Potential of Sparse DNNs Through Synergistic Hardware-Sparsity Co-Design," *IEEE Transactions on* Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2022.
- [25] K. Huang et al., "Structured Term Pruning for Computational Efficient Neural Networks Inference," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 1, pp. 190–203, 2022.
- [26] X. Xie *et al.*, "Efficient Layer-Wise N:M Sparse CNN Accelerator with Flexible SPEC: Sparse Processing Element Clusters," *Micromachines*, vol. 14, no. 3, p. 528, 2023.
- [27] C. Fang et al., "An Algorithm-Hardware Co-optimized Framework for Accelerating N:M Sparse Transformers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI)*, vol. 30, no. 11, pp. 1573–1586, 2022.
- [28] Z. Chen et al., "Dynamic N:M Fine-Grained Structured Sparse Attention Mechanism," in Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP), 2023, pp. 369–379.
- [29] M. Lin et al., "1xN Pattern for Pruning Convolutional Neural Networks," *IEEE Transactions on Pattern Analysis and Machine Intelli*gence (TPAMI), 2022.
- [30] E. Frantar et al., "SPDY: Accurate Pruning with Speedup Guarantees," in International Conference on Machine Learning (ICML). PMLR, 2022, pp. 6726–6743.
- [31] W. Sun et al., "DominoSearch: Find layer-wise fine-grained N:M sparse schemes from dense neural networks," Advances in Neural Information Processing Systems (NeurIPS), vol. 34, pp. 20721–20732, 2021.
- [32] A. Zhou et al., "Learning N:M Fine-grained Structured Sparse Neural Networks from Scratch," in International Conference on Learning Representations (ICLR), 2021.
- [33] S. K. Venkataramanaiah et al., "FPGA-based Low-batch Training Accelerator for Modern CNNs Featuring High Bandwidth Memory," in Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD), 2020, pp. 1–8.
- [34] Y. Tang et al., "EF-Train: Enable Efficient On-device CNN Training on FPGA Through Data Reshaping for Online Adaptation or Personalization," ACM Transactions on Design Automation of Electronic Systems (TODAES), 2022.
- [35] K. He et al., "FeCaffe: FPGA-enabled Caffe with OpenCL for Deep Learning Training and Inference on Intel Stratix 10," in Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2020, pp. 314–314.
- [36] Z. Liu et al., "An FPGA-based Processor for Training Convolutional Neural Networks," in 2017 International Conference on Field Programmable Technology (ICFPT). IEEE, 2017, pp. 207–210.
- [37] S. K. Venkataramanaiah et al., "Automatic Compiler Based FPGA Accelerator for CNN Training," in 2019 29th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2019, pp. 166–172.
- [38] X. Chen et al., "EILE: Efficient Incremental Learning on the Edge," in 2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS). IEEE, 2021, pp. 1–4.
- [39] T.-H. Tsai et al., "An On-Chip Fully Connected Neural Network Training Hardware Accelerator Based on Brain Float Point and Sparsity Awareness," *IEEE Open Journal of Circuits and Systems (OJCAS)*, 2023.

- [40] C. S. Rohwedder *et al.*, "Pooling Acceleration in the DaVinci Architecture Using Im2col and Col2im Instructions," in 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2021, pp. 46–55.
- [41] PyTorch, "PyTorch Profiler," https://pytorch.org/tutorials/recipes/recipes/ profiler_recipe.html, 2023.
- [42] P. Dai et al., "SparseTrain: Exploiting Dataflow Sparsity for Efficient Convolutional Neural Networks Training," in 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 2020, pp. 1–6.
- [43] J. Lu et al., "THETA: A High-Efficiency Training Accelerator for DNNs With Triple-Side Sparsity Exploration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI)*, vol. 30, no. 8, pp. 1034–1046, 2022.
- [44] C. Fang et al., "CEST: Computation-Efficient N:M Sparse Training for Deep Neural Networks," in 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2023, pp. 1–2.
- [45] Y. Wang *et al.*, "Trainer: An Energy-Efficient Edge-Device Training Processor Supporting Dynamic Weight Pruning," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 57, no. 10, pp. 3164–3178, 2022.
- [46] H. Shao et al., "An FPGA-Based Reconfigurable Accelerator for Low-Bit DNN Training," in 2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). IEEE, 2021, pp. 254–259.
- [47] C. Luo *et al.*, "Towards Efficient Deep Neural Network Training by FPGA-based Batch-level Parallelism," *Journal of Semiconductors (JOS)*, vol. 41, no. 2, p. 022403, 2020.
- [48] J. Lu et al., "ETA: An Efficient Training Accelerator for DNNs Based on Hardware-algorithm Co-optimization," *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)*, 2022.
- [49] H. Nakahara et al., "FPGA-based Training Accelerator Utilizing Sparseness of Convolutional Neural Network," in 2019 29th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2019, pp. 180–186.
- [50] NVIDIA, "Automatic Mixed Precision," https://nvidia.github.io/apex/ amp.html, 2018.
- [51] N. P. Jouppi et al., "In-Datacenter Performance Analysis of A Tensor Processing Unit," in Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), 2017, pp. 1–12.
- [52] D. Wu et al., "uSystolic: Byte-Crawling Unary Systolic Array," in 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2022, pp. 12–24.
- [53] H. Kung et al., "Maestro: A Memory-on-Logic Architecture for Coordinated Parallel Use of Many Systolic Arrays," in 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP), vol. 2160. IEEE, 2019, pp. 42–50.
- [54] K. He et al., "Deep Residual Learning for Image Recognition," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778.
- [55] K. Simonyan et al., "Very Deep Convolutional Networks for Large-Scale Image Recognition," in 3rd International Conference on Learning Representations (ICLR), 2015.
- [56] A. Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," in *International Conference on Learn*ing Representations (ICLR), 2021.
- [57] Y. Choukroun et al., "Low-bit Quantization of Neural Networks for Efficient Inference," in 2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW). IEEE, 2019, pp. 3009–3018.
- [58] D. Rossi et al., "PULP: A Parallel Ultra Low Power Platform for Next Generation IoT Applications," in 2015 IEEE Hot Chips 27 Symposium (HCS). IEEE Computer Society, 2015, pp. 1–39.
- [59] M. B. Taylor, "Basejump STL: Systemverilog Needs a Standard Template Library for Hardware Design," in 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). IEEE, 2018, pp. 1–6.
- [60] H. Fan et al., "Adaptable Butterfly Accelerator for Attention-based NNs via Hardware and Algorithm Co-design," in 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2022, pp. 599–615.
- [61] S. Q. Zhang et al., "Fast: DNN Training Under Variable Precision Block Floating Point with Stochastic Rounding," in 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2022, pp. 846–860.
- [62] Intel, "Intel Performance Counter Monitor," https://github.com/intel/ pcm, 2022.