An Algorithm–Hardware Co-Optimized Framework for Accelerating *N:M* Sparse Transformers

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

***Abstract*— The Transformer has been an indispensable staple in deep learning. However, for real-life applications, it is very challenging to deploy efficient Transformers due to the immense parameters and operations of models. To relieve this burden, exploiting sparsity is an effective approach to accelerate Transformers. Newly emerging Ampere graphics processing units (GPUs) leverage a 2:4 sparsity pattern to achieve model acceleration, while it can hardly meet the diverse algorithm and hardware constraints when deploying models. By contrast, we propose an algorithm–hardware co-optimized framework to flexibly and efficiently accelerate Transformers by utilizing general *N:M* sparsity patterns. First, from an algorithm perspective, we propose a sparsity inheritance mechanism along with inherited dynamic pruning (IDP) to obtain a series of *N:M* sparse candidate Transformers rapidly. A model compression scheme is further proposed to significantly reduce the storage requirement for deployment. Second, from a hardware perspective, we present a flexible and efficient hardware architecture, namely, STA, to achieve significant speedup when deploying *N:M* sparse Transformers. STA features not only a computing engine unifying both sparse–dense and dense–dense matrix multiplications with high computational efficiency but also a scalable softmax module eliminating the latency from intermediate off-chip data communication. Experimental results show that, compared to other methods, *N:M* sparse Transformers, generated using IDP, achieves an average of 6.7% improvement on accuracy with high training efficiency. Moreover, STA can achieve 14.47× and 11.33× speedups compared to Intel i9-9900X and NVIDIA RTX 2080 Ti, respectively, and perform 2.00 ∼19.47× faster inference than the state-of-the-art field-programmable gate array (FPGA)-based accelerators for Transformers.**

***Index Terms*— Algorithm–hardware codesign, hardware accelerator, model compression, pruning, Transformer.**

I. Introduction

TRANSFORMER-BASED networks are a formidable force in deep learning [1]. Tremendous impact on many fields, such as neural machine translation (NMT) [2], language understanding [3], and image processing [4], has been made since the innovation of Transformers. Nevertheless, the impressive performance of Transformers comes with heavy computing and memory costs, which become a significant barrier to the efficient deployment of Transformer-based applications. Notably, BERT, a representative Transformer-based model [3], requires 440-MB memory and over 176 G floating-point operations. Such severe requirements on memory and computation make it critical to find an efficient solution for deploying Transformers.

Sparsity is an important feature that can be utilized to improve the efficiency of DNNs deployment in dedicated accelerators. In the pioneering works, OPTIMUS [5] and EdgeBERT [6], the latest ASIC accelerators, leverage unstructured sparsity to realize efficient deployment for Transformers. Nevertheless, it is hard to predict the unstructured sparsity in advance, and therefore, the acceleration performance can be greatly dragged. Recent studies [7] demonstrate deep neural networks leveraging *N:M* fine-grained structured sparsity, where *N* out of *M* parameters are zeros for every continuous *M* parameters and can achieve comparable performance over those leveraging unstructured sparsity [8]. However, it is significantly restricted to accelerate *N:M* sparse networks on current hardware platforms. As shown in Fig. 1(a), the only existing solution is *ASP with Ampere graphics processing units (GPUs)* that focuses on the middle-level (2:4), i.e., 50%, sparse ratio. Based on our experiments, a heavy Transformer can be dramatically slimmed by weight pruning with the aggressive *N:M* pattern, e.g., 2:8 or 1:8, achieving a considerable reduction in the amount of both parameters and operations. The only choice of uniform 2:4 sparsity limits performance when deploying Transformer-based models, making it inflexible to meet different hardware constraints (e.g., latency and energy). Compared with the uniform 2:4 sparsity, the more flexible general *N:M* sparsity in real applications can satisfy various algorithm and hardware constraints under different deployment scenarios. However, there is currently a lack of an integrated framework to investigate the deployment of Transformer with general *N:M* sparse patterns. To bridge this gap, as presented in Fig. 1(b), we propose an algorithm–hardware co-optimized framework for accelerating *N:M* sparse Transformers, which addresses two significant issues: 1) how to produce a series of *N:M* sparse Transformers in an efficient way and 2) how to design a flexible and efficient dedicated architecture for *N:M* sparse Transformers on diverse field-programmable gate array (FPGA) platforms.

Although advanced optimization algorithms, ASP [9] and SR-STE [7], can maintain the middle-level (2:4) sparsity via static and dynamic fine-tuning, we observe existing methods

Manuscript received 27 February 2022; revised 30 June 2022; accepted 2 August 2022. Date of publication 16 August 2022; date of current version 24 October 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 62174084 and Grant 62104097, in part by the High-Level Personnel Project of Jiangsu Province under Grant JSSCBS20210034, and in part by the Key Research Plan of Jiangsu Province of China under Grant BE2019003-4. *(Corresponding author: Zhongfeng Wang.)*

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

Aojun Zhou is with the CUHK-Sensetime Joint Laboratory, The Chinese University of Hong Kong (CUHK), Hong Kong, China (e-mail: aojunzhou@gmail.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2022.3197282.

Digital Object Identifier 10.1109/TVLSI.2022.3197282

degrade the performance significantly under high sparse ratio (e.g., $\geq 75\%$). In addition, the ASP and SR-STE schemes leverage single-shot magnitude-based pruning for a specified hyperparameter N and M. This traditional recipe results in a significant performance drop with a higher sparse ratio and restricts the deployment of flexible $N{:}M$ sparse models on the FPGA platform. To overcome the aforementioned problems, we propose a sparsity inheritance mechanism, which increases the sparsity progressively to enable efficient searching for $N{:}M$ sparse Transformers under various sparsity configurations (e.g., 2:8 and 1:8). We also propose a pruning method, namely, inherited dynamic pruning (IDP), which shrinks prepruning models progressively, and the convergent prepruning initialized models can aid in the convergence of the following subnetworks. Extensive experiments are conducted on Transformer-based models, showing that models generated by IDP with the sparsity inheritance mechanism have superior performance on various sparsity ratios than those using the ASP and SR-STE. Moreover, for efficient model deployment, we apply a simple but effective bitmap-based compression scheme, which dramatically reduces the storage requirements for $N{:}M$ sparse Transformers.

To enable flexible and efficient deployment on various FPGA devices, we design a highly configurable dedicated accelerator for $N{:}M$ sparse Transformers, namely, STA. STA fully explores the parallelism of Transformers in three aspects, including head parallelism, row parallelism, and column parallelism, which significantly improves computational efficiency. It features two computing cores, a diverse matrix multiplication (MatMul) engine, called DMME, and a scalable softmax module, both of which are highly configurable. Operations of $N{:}M$ sparse Transformers are dominated by two types of MatMuls. One is the sparse–dense MatMul with $N{:}M$ sparse network parameters, and the other is dense–dense MatMul free of parameters. DMME performs both sparse–dense and dense–dense MatMuls on-the-fly and achieves much higher computational efficiency over the prior work [10] under both modes. Especially for sparse–dense MatMul, DMME only performs operations related to those remaining nonzero parameters, which greatly improves computational efficiency. The scalable softmax module can perform the softmax function in Transformers. It keeps all the intermediate results fully local, eliminating latency from intermediate off-chip data communication. According to the given architectural settings, STA can be rapidly implemented on FPGAs to realize efficient deployment for specific $N{:}M$ sparse Transformers.

To summarize, the contributions of this article are given as follows.

1) To the best of our knowledge, this is the first work that presents an algorithm–hardware co-optimized framework to systematically study the efficiency of fine-grained $N{:}M$ sparse Transformers on FPGA. The proposed framework can adjust to diverse hardware constraints for flexible and efficient model deployment.
2) To generate a series of $N{:}M$ sparse Transformers simultaneously, we propose a sparsity inheritance mechanism along with the IDP algorithm, which can significantly achieve about 6.7% accuracy improvement of Transformers under high sparsity compared with state-of-the-art methods.
3) We present a simple but effective bitmap-based compression scheme for $N{:}M$ sparse Transformers compared to multiple sparse indexing formats, which dramatically reduces the storage requirements up to 5.33×.
4) We propose a dedicated hardware architecture, namely, STA, to realize flexible and efficient deployment of $N{:}M$ sparse Transformers. It features two novel hardware modules handling intensive operations of Transformers, including a diverse MatMul engine (DMME) that unifies dense and sparse MatMul operations in high computational efficiency and a scalable softmax module to avoid frequent off-chip memory accesses.
5) Extensive experiments have been conducted on four NLP tasks and four Transformer-based models to evaluate the effectiveness of the proposed framework, which achieves up to 19.47× speedup over Intel i9-9900X, NVIDIA RTX 2080 Ti, and prior FPGA-based accelerators for Transformers.

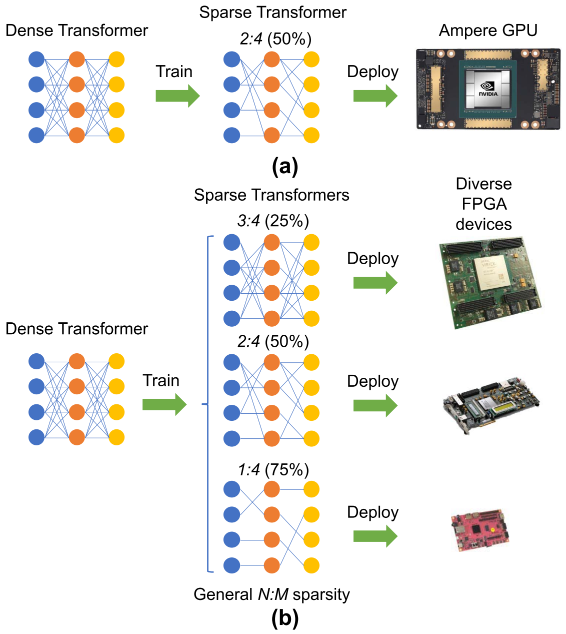


Fig. 1. Accelerating $N{:}M$ sparse Transformer-based models (a) using modern ampere GPUs and (b) using diverse FPGAs with our framework. Compared to (a), (b) can generate a series of $N{:}M$ sparse Transformers along with the dedicated accelerators for efficient model deployment. (a) Deploy 2:4 sparse Transformers on Ampere GPU. (b) Deploy general $N{:}M$ sparse Transformers on diverse FPGA devices.

The rest of this article is organized as follows. Section II presents an overview of Transformers and state-of-the-art works for accelerating Transformers with innovations in hardware architecture. Section III introduces the workflow of our proposed algorithm–hardware co-optimization framework. Sections IV and V elaborate on optimizations on pruning algorithm and hardware architecture, respectively. Comprehensive experimental results are presented in Section VI to show the significant potential of our proposed co-optimization framework in Transformer-based applications.

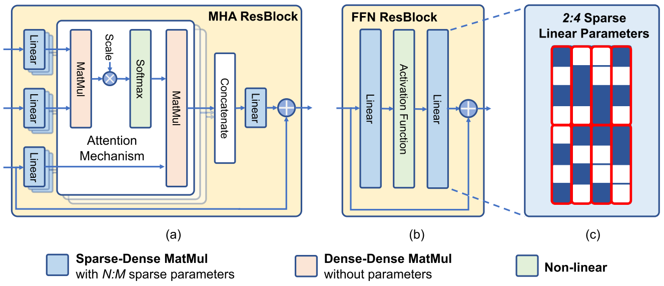


Fig. 2. Operations in (a) MHA ResBlock and (b) FFN ResBlock under *N:M* sparsity pattern. Both Resblocks are the key structures of the Transformer. (c) Illustration of 2:4 sparse parameters in a linear layer.

II. Background and Motivation

In this section, we provide an overview of key structures in Transformers and review related work on hardware accelerators for Transformers.

A. Transformer Overview

The key architectures of the Transformer [11] are characterized by a multihead attention (MHA) residual block (ResBlock) and a positionwise feedforward network (FFN) ResBlock. Fig. 2(a) and (b) illustrates the inner structures of MHA and FFN ResBlocks, respectively. The input and output of the FFN ResBlock are connected by a residual connector. Two linear transformation modules along with an activation function are inside the FFN ResBlock. The structure of MHA ResBlock is more complicated. The inputs of MHA ResBlock are split into multiple parallel heads with corresponding linear projection at first. Then, the results are as input fed into the attention mechanism in parallel, and finally, the results of attention heads are concatenated together and passed into a linear layer to obtain the output linear projection. Note that the attention mechanism is totally different from the linear layer, performing parameter-free MatMuls. Thus, the computing engine for Transformers is required to support both sparse and dense MatMuls even though sparsity is introduced to parameters. The residual connector of FFN ResBlock is organized the same as the FFN ResBlock.

B. Recent Advances for Transformer Acceleration

Extensive research has concentrated on the design of high-performance and energy-efficient DNN hardware accelerators [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28]. However, most of these works focus on CNN and RNN computations, and not as much scrutiny has been given to accelerating Transformer-based networks with self-attention mechanisms.

As a pioneer work, Lu *et al.* [10] proposed a dense systolic array accelerator along with the partitioning scheme for FPGA-based acceleration of Transformers. Moreover, FTRANS [29] exploited block-circulant matrix-based weight representation for Transformer acceleration. However, both of them fail to utilize the sparsity of parameters in Transformers, thereby limiting the speedup of model deployment. A^3 [30], SpAtten [31], and Sanger [32] merely focused on the speedup potential for the sparse attention mechanism, all of which can hardly satisfy the needs of agile and efficient deployment of Transformer models. OPTIMUS [5] and EdgeBERT [6] holistically accelerate Transformers with unstructured sparse MatMuls and save energy by skipping the computations related to those zero-value parameters. Nevertheless, the unstructured sparsity leads to irregular data access, making both designs suffer low computational efficiency. Peng *et al.* [33] exploited the coarse-grained block-based sparsity pattern for accelerating Transformers, while this sparsity pattern is so coarse that the models can hardly achieve a considerable sparsity ratio with acceptable accuracy.

In summary, it is hard for all these works to achieve satisfying speedup and efficiency of Transformer deployment due to the lack of attention to model sparsity, limited sparse potential exploration on the whole Transformer models, or restricted computational efficiency for sparse Transformers. To address the above issues, this work presents an algorithm–hardware co-optimization framework to realize flexible and efficient deployment of Transformers by leveraging general *N:M* sparsity patterns. For algorithm optimization, we focus on how to generate a series of *N:M* sparse Transformers in high quality and efficiency. For hardware optimization, we concentrate on designing a flexible and efficient dedicated architecture that can accelerate *N:M* sparse Transformers with high computational efficiency.

III. Overview of Co-Optimization

To achieve agile and efficient deployment of Transformers, we propose an algorithm–hardware co-optimized framework. The overview of our framework is presented in Fig. 3. According to the given specific requirements, our framework can quickly obtain the required *N:M* sparse Transformer model with high accuracy and provide corresponding Transformer accelerators on FPGA devices to realize efficient model deployment. In this section, we elaborate on the workflow of our algorithm–hardware co-optimized framework.

At the algorithm level, we focus on quickly obtaining any desired *N:M* sparse Transformer model and achieving effective compression of the *N:M* sparse Transformer. The algorithm optimization is divided into two stages. As shown in Fig. 3, the first stage is IDP based on the sparsity inheritance mechanism. Compared with single-shot training [7], [9], our method can utilize the knowledge of the previous *N:M* sparse model, which contributes to faster and better convergence. The second stage is model compression. Only the nonzero parameters in the *N:M* sparse Transformer would be stored, along with an additional binary mask that indicates the position of all recorded elements. The methods of pruning and model compression are presented in Section IV.

At the hardware level, we concentrate on efficient and flexible hardware architecture design that boosts the computational efficiency for *N:M* sparse Transformers. The hardware optimization features an efficient hardware architecture for *N:M* sparse Transformers along with an automatic hardware generator, which can meet requirements on various

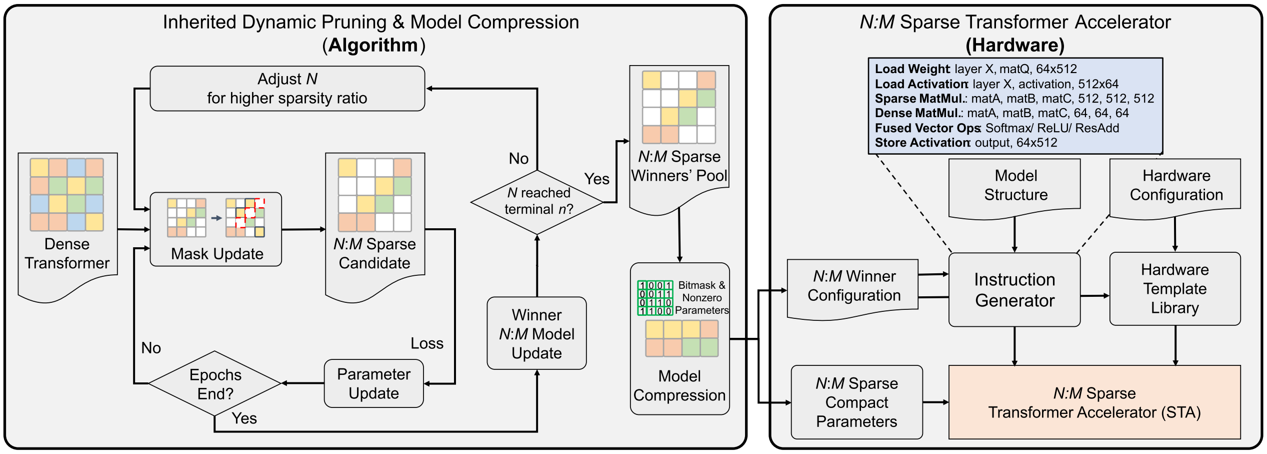

Inherited Dynamic Pruning & Model Compression
(Algorithm)
Adjust N
for higher sparsity ratio
Dense Transformer
Mask Update
N:M Sparse Candidate
No
Yes
N reached terminal n?
N:M Sparse Winners' Pool
Winner N:M Model Update
Loss
No
Epochs End?
Yes
Parameter Update
1001
0011
0110
1100
Bitmask & Nonzero Parameters
Model Compression
N:M Sparse Transformer Accelerator
(Hardware)
Load Weight: layer X, matQ, 64x512
Load Activation: layer X, activation, 512x64
Sparse MatMul.: matA, matB, matC, 512, 512, 512
Dense MatMul.: matA, matB, matC, 64, 64, 64
Fused Vector Ops: Softmax/ ReLU/ ResAdd
Store Activation: output, 64x512
Model Structure
Hardware Configuration
N:M Winner Configuration
Instruction Generator
Hardware Template Library
N:M Sparse Compact Parameters
N:M Sparse Transformer Accelerator (STA)

Fig. 3. Workflow of our proposed algorithm–hardware co-optimized framework. At the algorithm level, *N:M* sparse Transformers can be rapidly generated by IDP and significantly compressed for further deployment. At the hardware level, the dedicated accelerator, STA, is implemented on the FPGA platform to accelerate the deployed *N:M* sparse Transformer.

Transformer models, FPGA devices, and *N:M* sparsity. The automatic hardware generator is composed of an instruction generator and a hardware template library. According to the *N:M* configuration of the winner model and the network structure of the deployment model, the instruction generator can automatically produce instructions that guide STA to perform operations of the winner *N:M* sparse Transformer. As shown in Fig. 3, instructions are divided into three categories: load/store data, sparse/dense MatMul operators, and fused vector operators. The hardware template library can quickly generate a dedicated STA based on the predefined hardware configurations and the *N:M* configuration of the winner model. STA performs inference tasks for the Transformer model by accessing the compact sparse parameters and the pregenerated instructions. STA, whose hardware architecture is elaborated in Section V, can achieve significant improvement in computational efficiency by eliminating all zero-valued parameter operations.

In real-life deployment, the choice of *N:M* may change if there are multiple FPGA devices with different hardware resources and varying deployment constraints, including latency and model accuracy. However, considering all the above factors, once an *N:M* model is determined to be deployed, the model can meet the needs of practical applications. Therefore, the *N:M* would change before deploying models, while it would not change after the model deployment. Compared to the Ampere GPU dedicated for 2:4 sparse acceleration, specific *N:M* STA can be flexibly configured and automatically generated on the selected FPGA device with significant performance gains benefited from dedicated *N:M* sparse acceleration. As *N:M* changes, our framework would efficiently benefit from the algorithm–hardware co-optimization. At the algorithm level, the proposed IDP could provide a series of *N:M* models with varying computing complexity and model accuracy, among which we could select the most suitable one for further model deployment. At the hardware level, the proposed STA could be flexibly generated based on the selected *N:M* and other configurations, achieving significant acceleration of *N:M* Transformers.

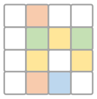
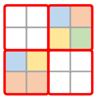
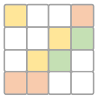
IV. Algorithm Optimization

In this section, we elaborated on the algorithm optimizations of our framework. First, we demonstrate advantages of *N:M* sparsity pattern in Section IV-A by comparing it with other popular sparsity patterns. Then, the pruning algorithm and the compression scheme of *N:M* sparse Transformers are presented in Sections IV-B and IV-C, respectively.

A. N:M Sparsity Pattern

A dense parameter matrix can be pruned with a sparsity ratio of 50% using three existing sparsity patterns, unstructured sparsity [5], block-based structured sparsity [33], and *N:M* group-based structured sparsity [7], respectively. Table I summarizes all these pruning patterns. Elements in any position of the parameter matrix can be pruned if the unstructured sparsity pattern is employed. The unstructured sparse model can achieve a considerable compression ratio (CR) while maintaining comparable accuracy to the dense model. However, there is a limited speedup of the unstructured sparse model on hardware [5], [6] due to the irregular pattern. For block-based pruning, the parameter matrix is first divided into multiple blocks, and then, some unimportant blocks was dropped to reduce storage and computing. The block-based sparse model having a regular pattern can achieve high computational efficiency on hardware. Nevertheless, the speedup of block-based sparse models [33] is inefficient since there is limited CR using the block-based pattern. As for *N:M* group-based structured sparsity, the parameter matrix is divided into multiple groups. Here, we consider consecutive columnwise elements in the matrix gathered as a group. Each group has M elements and contains N nonzero elements at most. The *N:M* sparsity can achieve high CR along with computational efficiency on hardware due to its fined-grained regular pattern. Hence, Transformers with *N:M* sparsity pattern, which remains a lot to be explored, have much more speedup potential than that with unstructured and block-based sparsity.

TABLE I
COMPARISON BETWEEN EXISTING THREE SPARSITY PATTERNS

	Unstructured	Block-based	**N:M Group-based**
Visualization			
Accuracy	**High** ✓	Medium	**High** ✓
Efficiency	Low	**High** ✓	**High** ✓
Speedup	Medium	**High** ✓	**High** ✓

B. Pruning Algorithm

Given a pretrained dense Transformer model, generally, a *N:M* sparse Transformer can be trained with the objective as

$$\min_{S(\mathcal{W},N,M)} \mathcal{L}(\mathcal{W};\mathcal{D}) \tag{1}$$

where $\mathcal{D}$ denotes the observed data, $\mathcal{L}$ represents the loss function, $\mathcal{W}$ indicates the parameters of the Transformer, and $S(\mathcal{W}, N, M)$ is the sparse Transformer with *N:M* sparsity pattern. N is the number of nonzero values. For the dense model $\mathcal{W}$, it can be equivalent to $S(\mathcal{W}, N = M, M)$.

Existing methods, NVIDIA ASP [9] and SR-STE [7], leverage the single-shot magnitude-based pruning and dynamic sparse training from dense models $\mathcal{W}$, respectively. The specific sparse models $S(\mathcal{W}, N, M)$ inherit from global dense models $S(\mathcal{W}, M, M)$ with pretrained weights and random initialization in ASP and SR-STE. This may lead to suboptimal problems, and we observe the ASP and SR-STE hurt the performance significantly on Transformer-based models with the higher sparse ratio (e.g., $\geq$75%). In addition, the ASP and SR-STE undesirably require intensive training computation if we have different hardware constraints with multiple sparsity levels (e.g., 1:8, 2:8, 3:8, and 4:8).

Therefore, we propose a general and simple algorithm for generating models with general *N:M* sparse patterns, namely, IDP, which can produce a series of sparse models with different *N:M* configurations. Algorithm 1 presents the detail of IDP. To handle the optimization difficulty of the sparse subnetworks inherited from large dense models, we introduce a novel cotraining scheme, which optimizes different multiple-level *N:M* sparse models simultaneously (e.g., 1:8, 2:8, 3:8, 4:8, and 5:8). During the training phase, we gradually reduce the nonzeros parameters N, which can guarantee the super models converge well. We can give the general inheritance mechanism of the IDP as follows:

$$S(\mathcal{W}, N_1, M) \leftarrow S(\mathcal{W}, N_2, M) \leftarrow \cdots \leftarrow S(\mathcal{W}, M, M) \tag{2}$$

where $S(\mathcal{W}, N_i, M)$ are *N:M* sparse models, and the $S(\mathcal{W}, M, M)$ represents the dense model, where $N_1 < N_2 < \cdots < M$, and $\leftarrow$ means the smaller model $S(\mathcal{W}, N - 1, M)$ prune from $S(\mathcal{W}, N, M)$, named inheritance mechanism. It requires merely a hyperparameter n denoted as the end of iterations of N. With the novel inheritance mechanism, our IDP training method can be summarized in four steps.

Step 1: Initialize $N = M - 1$, and set the dense pretrained model as the first winner model.

Algorithm 1 IDP

```
Input: Pre-trained dense weights $\mathcal{W}$, datasets $\mathcal{D}$, initial learning rate $\gamma_0$ and the end of iterations $n$.
for $N = M-1, M-2, .., n$ do
    Sparsity Inheritance:
        $S(\mathcal{W}, N, M) \leftarrow$ the winner of $S(\mathcal{W}, N+1, M)$.
    for each training iteration $t$ do
        Forward Pass:
            generate $\mathcal{B}_t$ by group-wise magnitude pruning.
        Backward Pass:
            $\mathcal{W}_{t+1} = \mathcal{W}_t - \gamma_t g(\mathcal{W}_t \odot \mathcal{B}_t) + \lambda((1-\mathcal{B}_t) \odot W_t)$.
    end for
end for
Output: A series of N:M sparse models with different computation complexity and corresponding masks: the winners of $S(\mathcal{W}, N = M-1, M)$, $S(\mathcal{W}, N = M-2, M)$,.., $S(\mathcal{W}, N = n, M)$.
```

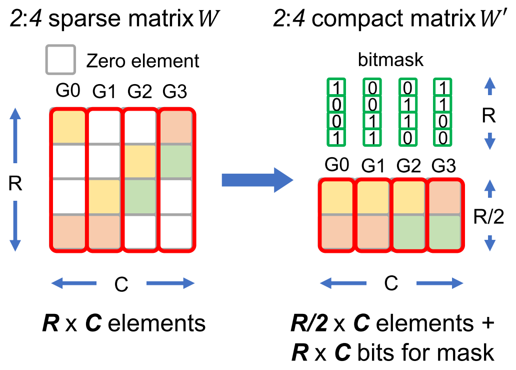


Fig. 4. Compact storage scheme example for *N:M* sparse parameter matrix.

Step 2: Sparsity inheritance applies the kept parameters of the winner of all the $S(\mathcal{W}, N + 1, M)$ candidates to initialize following sparse model $S(\mathcal{W}, N, M)$.

Step 3: Sparse training for *N:M* sparse candidates in several epochs. Parameters are adjusted in every epoch by updating the mask based on their magnitude. This step generates a new winner model, which is the convergent model at the last epoch.

Step 4: If $N = n$, the whole process is finished, or otherwise, $N \leftarrow N - 1$, and then, go to *Step 2*.

We expect to obtain $M - N + 1$ preserved winner models with different *N:M* sparsities for subsequent deployment. In addition, in the forward pass, we leverage the popular groupwise magnitude pruning [7], [9]. Parameter matrices are partitioned into multiple groups, every one of which contains M consecutive columnwise elements, as shown in Table I. We keep the N-largest parameters in these groups and generate corresponding masks $\mathcal{B} \in \{0, 1\}^d$. Specifically, if the ith parameter of $\mathcal{W}$ survived in the pruned subnetwork, we set $\mathcal{B}_i = 1$, or else, $\mathcal{B}_i = 0$. In the backward pass, recent studies [7], [34] demonstrate that the dynamic sparse training can benefit both model convergence and accuracy, and we follow their methods to calculate gradients.

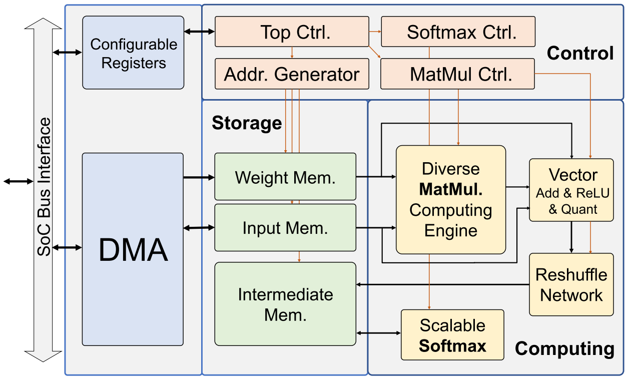


Fig. 5. Overall architecture of STA. It is composed of computing, storage, and control function blocks. These red arrows pass control signals, while those black arrows transfer data.

C. Packing N:M Sparse Parameters

An *N:M* sparse Transformer can be obtained after IDP, where each group of all parameter matrices only contains at most *N* nonzero elements. However, it occupies a large amount of memory since the parameter store scheme is the same as the dense Transformer. We apply the bitmap-based compression scheme to obtain a compact *N:M* sparse Transformer, which greatly achieves saving on storage for a deployment. Compared to COO, CSC, CSR [35], and step indexing [36], our scheme has better compression performance in the range of practical *N:M* sparsity. Fig. 4 presents the compression scheme of *N:M* sparse parameter matrix using 2:4 sparsity as an example. For a parameter matrix $W \in \mathbb{R}^{R\times C}$, after IDP, there are at most two nonzero elements in a group. The entire parameter matrix has $(R/2) \times C$ remaining nonzero elements. In our scheme, we merely preserve nonzero elements in each group and use a binary mask to indicate the elements' position. By using our scheme, a 2:4 parameter matrix $W \in \mathbb{R}^{R\,\times\,C}$ can be stored with $(R/2) \times C$ valid elements and $R \times C$ bits for the mask, instead of the $R \times C$ elements.

Considering a dense parameter matrix $W \in \mathbb{R}^{R\times C}$, in which all elements are quantized using q bits, W can be compressed to $\tilde{W} \in \mathbb{R}^{R\times\lceil(C/M)\rceil N}$, where there are $R\lceil(C/M)\rceil$ groups, and each group has N nonzero parameters at most. The storage requirement of W is qRC bits, and after pruning, we can only store the *N:M* sparse matrix $\tilde{W}$ in a compact way with only $qR\lceil(C/M)\rceil N$ bits and an additional binary mask with RC bits. Therefore, the CR can be represented as

$$\mathrm{CR} = \frac{qC}{q\lceil\frac{C}{M}\rceil N + C}. \tag{3}$$

V. Hardware Optimization

The flexible and efficient hardware architecture, namely, STA, is developed for *N:M* sparse Transformers in this section. We first present the overall architecture of STA and then elaborate on the designs of its core computing engines, including DMME and scalable softmax module.

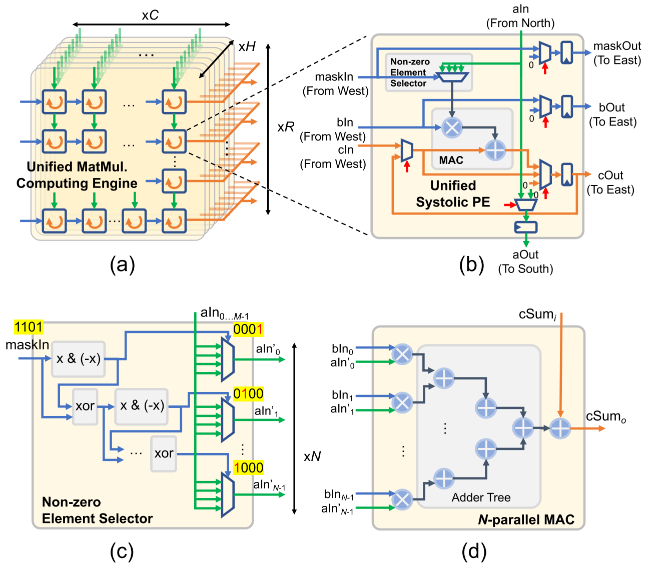


Fig. 6. Hierarchical architecture of DMME. (a) It consists of an H parallel unified MatMul computing engine. (b) Each engine contains $R \times C$ unified systolic PE capable of handling both sparse–dense and dense–dense dot products. The key components of PEs, NZES, and the N-MAC are shown in (c) and (d), respectively.

A. Overall Architecture

The overall architecture of STA is shown in Fig. 5, which consists of three major function blocks, including computing, storage, and control. The computing blocks consist of a diverse MatMul computing engine, namely, DMME, a scalable softmax module, a vector unit, and a data reshuffle network. Dominated operations of *N:M* sparse Transformers, i.e., sparse–dense or dense–dense MatMuls, are performed by DMME on the fly with the dynamic configuration under high computational efficiency. The scalable softmax module is responsible for the softmax operation in MHA ResBlocks, eliminating the off-chip transfer for intermediate data. The vector unit takes charge of operations with low computational density including bias addition, residual addition, and activation functions. The reshuffle network reorders the temporary results before writing back to the intermediate on-chip memory. As for on-chip storage, it can be partitioned into three parts, including the weight memory, the input memory, and the intermediate memory. The weight and input memory store model parameters and input data of Transformers from the off-chip memory, respectively. The results of a ResBlock are also written back to the input memory and pass to the off-chip memory. All the temporary results in a ResBlock will be stored in the intermediate memory with no communication to the external memory.

B. DMME

DMME unifies both sparse–dense and dense–dense MatMuls with high computational efficiency in *N:M* sparse Transformers. When it performs sparse–dense MatMuls, it merely loads nonzero weight parameters and selects corresponding

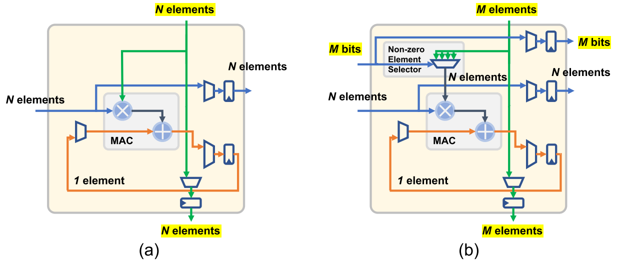


Fig. 7. Activated datapath of PEs under (a) dense–dense and (b) sparse–dense modes. (a) Dense–dense MatMul. (b) Sparse–dense MatMul.

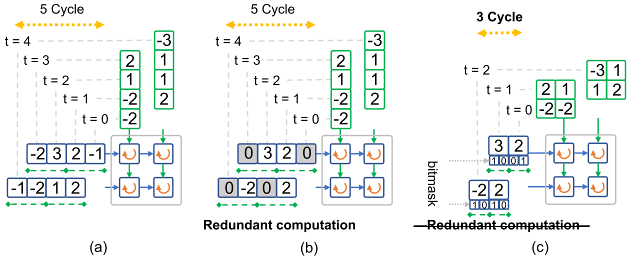


Fig. 8. Computing dataflows of DMME when it performs (a) dense–dense and (c) sparse–dense (on the classic array) MatMuls. Compared to (b) as a baseline, (c) eliminates all zero-valued redundant operations under sparse–dense mode, thus improving computational efficiency. (c) Sparse–dense MatMul on the unified computing engine

activations to compute, thereby improving computational efficiency.

The architecture of the DMME is illustrated in Fig. 6. It is a two-level hierarchy design with a full exploration of parallelism inside the MatMuls of *N:M* sparse Transformers. The exploited parallelism consists of head parallelism, row parallelism, and column parallelism, which are denoted as H, R, and C, respectively. The DMME is composed of H parallel $R \times C$ unified MatMul computing engine [see Fig. 6(a)], every one of which can efficiently realize both sparse–dense and dense–dense MatMuls in a time-division multiplexing manner. The capability of performing sparse–dense and dense–dense MatMuls comes from the inner unified systolic PE [see Fig. 6(b)]. It is composed of a nonzero element selector (NZES), an N-parallel MAC (N-MAC), multiple multiplexers, and registers. The NZES, only being activated in the sparse–dense MatMul mode, is to select the proper activation according to the input bitmask. The N-MAC accepts N 16-bit input data and parameters, realizes the inner product, and then accumulates the result with the local 32-bit output partial sum. The multiplexers and registers are used for datapath selection and temporary data storage, respectively. The design of NZES is presented in Fig. 6(c). It takes as input an M-bit mask, in which only N bits are set as 1 to indicate the position of nonzero elements, and then generates N one-hot encoding masks to select the corresponding N data for dot product computation. The translation to N one-hot encoding masks is performed by cascading the simple bit-arithmetic blocks and *XOR* gates. With the help of N one-hot encoding masks, data related to nonzero parameters are fed into the N-MAC along with these nonzero parameters in one group. It could be pointed out that the NZES can be further optimized by pruning the redundant indexing indicators and element candidates. The N-MAC, as shown in Fig. 6(d), is composed of N parallel multipliers, an adder tree, and a final accumulator. Both NZES and N-MAC are fully pipelined to maximize the throughput of DMME.

The activated datapaths under dense–dense and sparse–dense MatMuls are presented in Fig. 7(a) and (b), respectively. The dense–dense mode of PEs would be only activated when performing the self-attention operation of MHA. Both input operands are arranged in dense sequences in this mode. In this case, N elements, as an operand, in the input sequences are in parallel streamed into the PE from the west and the north, respectively. In a cycle, the PE performs a dot product with a size of N under dense–dense mode. The partial sum is stored in the local registers, and the input operands from the west and the north stored in the registers are passed into adjacent PEs on the east and south, respectively. For energy saving, the NZES is bypassed to avoid signal switching. Under the sparse–dense MatMul mode, as shown in Fig. 7(b), the input operands are different from that under the dense–dense mode. In a cycle, N nonzero parameters in a group along with the corresponding M-bit mask are streamed into the PE from the west, while M data in one group are fed from the north. The N valid data in pair with the input parameters are picked up by the NZES and then perform a dot product with these input parameters. When the computing task is done, under either dense–dense or sparse–dense modes, the PE turns into the shifting mode, accepts the results from its western PE to its local registers, and transfers its local result registers to the east.

C. Supporting Efficient Matrix Computations

STA is capable of supporting both sparse–dense and dense–dense MatMuls of *N:M* sparse Transformers in an efficient way. We demonstrate this significant capability of STA by exploiting four aspects: the computing dataflow of DMME, data access pattern of DMME, data mapping of input memory, and datapath from input memory to DMME.

Fig. 8 illustrates efficient computing dataflows of DMME under both dense–dense and sparse–dense modes. For simplicity, we assume that *N:M* is 1:2, and MatMul is performed by 2 × 4 input sequences and 4 × 2 parameter sequences. Here, we consider the computing engine in [10] as a baseline, which is orchestrated as a classic systolic array. In Fig. 8(a), DMME finishes the dense–dense MatMul in the given computing task using five cycles, which consumes the same cycles as the baseline. Hence, DMME achieves the same computational efficiency as the baseline when performing dense–dense MatMuls. As for sparse–dense MatMuls, Fig. 8(b) and (c) presents the computational manner of the baseline and DMME, respectively. The baseline takes five cycles to finish the task, while it cost merely three cycles by DMME since the redundant operations can be skipped with no waste on computing cycles. For sparse–dense MatMuls, DMME improves the computational efficiency by eliminating redundant computations, thereby significantly reducing latency and energy.

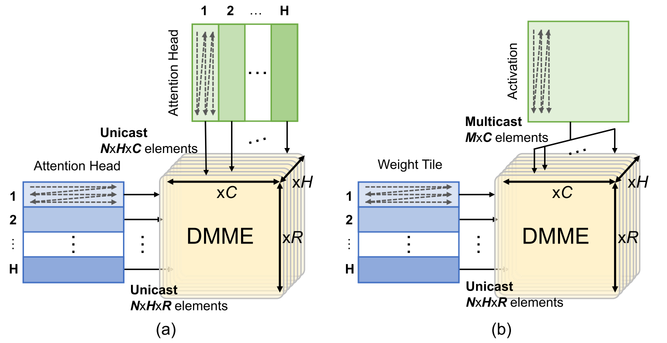


Fig. 9. Data access pattern of DMME to support efficient MatMuls under (a) dense–dense and (b) sparse–dense modes. (a) Dense–dense MatMul. (b) Sparse–dense MatMul.

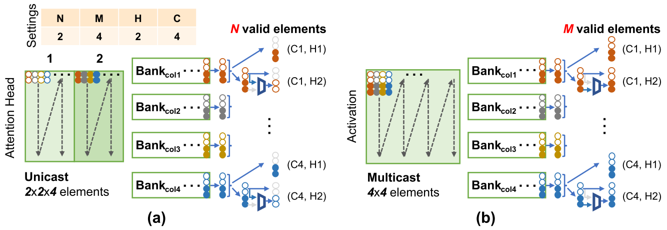


Fig. 10. Data mapping of input memory and datapath from input memory to DMME under (a) dense–dense and (b) sparse–dense modes. (a) Dense–dense MatMul. (b) Sparse–dense MatMul.

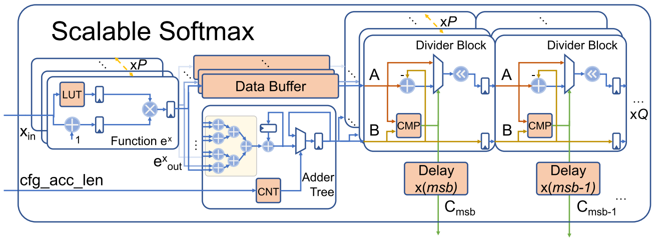


Fig. 11. Architecture of the scalable softmax operator.

Fig. 9(a) and (b) presents data access patterns of DMME when it performs dense–dense and sparse–dense MatMuls, respectively. For dense–dense MatMuls, attention heads as input are both separated into H tiles. In this case, DMME can be decomposed as H independent systolic arrays, every one of which fetches elements from the corresponding tiles to the top-most and leftmost PEs, respectively. For sparse–dense MatMuls, compressed weight parameters are divided into H tiles. Every cycle DMME fetches NR weight elements from all H tiles in parallel and casts them one-on-one to the leftmost systolic PEs in H systolic arrays. DMME is also required to access MC activation elements and broadcasts them to the top-most PEs in all H unified systolic arrays.

To balance the bandwidth of input memory when switching between dense–dense and sparse–dense modes, we make NH equal to M of STA. There are C banks of STA for input data storage. Fig. 10 illustrates data mapping of input memory and datapath from input memory to DMME by assuming that N is 2, M is 4, H is 2, and C is 4. The data storage structure in the input memory is varied for different computing modes. In the dense–dense mode, DMME performs two parallel dense–dense MatMuls for the attention mechanism of Transformers. There are two tiles for the loaded input data. As shown in Fig. 10(a), the first bank is connected to the first column of DMME, and the first address of the bank indexes the data from the first two elements at the first column in the tile one and two, respectively. In the sparse–dense mode, DMME performs MatMuls with the *N:M* sparse parameters. As depicted in Fig. 10(b), we do not tile input data for sparse–dense MatMul. The first address of the bank one that is connected to the first column of DMME, indexes the data from the first four elements in the first column of input data. It is the same as the indexing principle for the other banks in both computing modes.

As for the datapath from input memory to DMME, we take the first column of DMME as an example. There are four elements accessed from the first bank streaming to the first column with a head dimension of 2. In the dense–dense mode, as shown in Fig. 10(a), the unified systolic PE in the first head of the first column directly receives these four elements, and the lowest two elements are fed into N-MAC for computing. However, the unified systolic PE in the second head of the first column requires 2-to-1 MUXs to select the correct two elements from the four accessed elements for computing. In the sparse–dense mode, as presented in Fig. 10(b), data accessed from the first bank broadcast to unified systolic PEs in all head dimensions of the first column of DMME.

D. Scalable Softmax Module

The softmax function takes as input a vector $\mathbf{x}$ of n real numbers and normalizes it into a probability distribution consisting of n probabilities proportional to the exponentials of the input numbers. It is critical for Transformer dedicated accelerators to contain a softmax hardware implementation since the softmax function appears in every MHA module of Transformers. Fig. 11 presents the details of our proposed scalable softmax architecture, which is capable of performing softmax functions of arbitrary length. It keeps all the intermediate results fully local, avoiding off-chip data communication.

The architecture has two adjustable parameters, P and Q, where P denotes the parallelism of the architecture and Q represents the pipeline depth, as well as the output precision. P input data are streamed into the softmax module in parallel and transformed into the exponent outputs. The exponent outputs are not only temporarily stored in the data buffer but also used as input for further accumulation. Once the accumulation process is done, the divider module takes both accumulated results and exponent outputs as input to perform Q-level pipelined division and generates P softmax function outputs represented by Q bit.

As shown in Fig. 11, the scalable softmax module consists of three major parts: an area-efficient exponential function, a partial sum accumulator, and a scalable divider. The exponential function is approximated using a lookup table combined with a first-order Taylor expansion. An exponential operator can be implemented using only one multiplier and one adder. The configurable partial sum accumulator can adapt to input vectors of various lengths, which improves the flexibility of the hardware. To reduce the latency of the division, we design a highly parallel divider by cascading multiple divider blocks with pipelines, where a divider block is composed of subtractors and shifters with little cost on hardware.

VI. Experimental Results

In this section, we comprehensively evaluate both algorithm and hardware optimizations of the proposed framework. Three benchmark sets with varying sizes and complexity are applied to evaluate the proposed framework.

A. Experimental Setup

1) Benchmark Sets: The first set focuses on the evaluation of algorithm optimizations, by comprehensively presenting improvements on both model accuracy and CR under various *N:M* configurations. This benchmark set comprises a BERT model [3], the well-known Transformer-based model, and four evaluation datasets from the GLUE benchmark [37], including WNLI, QNLI, QQP, and MRPC. WNLI is a reading comprehension task. QNLI is a question–answering dataset consisting of the question–paragraph pairs. QQP is a collection of question pairs from the community question–answering website Quora. MPRC is a corpus of sentence pairs automatically extracted from online news sources. For these four tasks, we report the accuracy of the validation sets. We also report the CR on BERT by setting various *N:M* configurations and quantized bitwidth of parameters.

The second set divides into hardware resource consumption. First, we study the consumption of DMMEs, the core computing engine in the STA, at any common scale of *N:M* sparsity, and then, we explore hardware utilization of representative STAs under various FPGA devices. For the former evaluation, DMMEs are not allowed to be synthesized using DSP blocks, which can be done by setting the property *MAX_DSP* as zero. Hence, the consumption of LUTs and FFs can measure the cost of combinational logic and sequential logic for DMMEs, respectively. The utilization of LUTs and FFs is reported at the synthesis stage as the metric of hardware resource requirements. For the latter evaluation, the resource utilization of STAs on various FPGA devices is reported at the implementation stage, including the consuming amount of LUTs, FFs, BRAMs, and DSPs.

The third set studies performance improvements of the overall STA hardware system on multiple FPGA platforms when deploying various Transformer-based models. All key configurations of evaluated models in benchmark sets are presented in Table II. First, we evaluate the processing time with a single batch on all MHA and FFN ResBlocks in varying models from TinyBERT [38], Dino [39], and the Transformer-based model [11]. The selected models target different applications. TinyBERT is a lightweight BERT model for many language tasks. Dino, a tiny vision Transformer, can be the backbone for a lot of computer vision tasks. The Transformer-based model is the classic one for the NMT task. Considering that NMT is one of the sequence-to-sequence tasks, hence, we split the Transformer-based model into two parts, the stacked encoders and decoders, respectively. We, finally, make a fair comparison of the implemented STAs with previous works and commercial products using a shallow Transformer, which is the commonly used benchmark model in [29] and [33]. Latency, throughput, power, energy efficiency, and MAC efficiency are key metrics for applications and, thus, used for performance evaluation.

TABLE II

Key Configurations of Transformer-Based Models in Benchmark Sets

Benchmark	Model	Num. of Encoders	Num. of Decoders	Sequence length	Attention heads	Hidden size	Intermediate size
Set I	BERT	12	0	128	12	768	3072
Set III	TinyBERT4	4	0	128	12	312	1200
	Dino-vits8	12	0	64	6	384	1536
	Transformer-base stacked encoders	6	0	64	8	512	2048
	Transformer-base stacked decoders	0	6	64	8	512	2048
	Shallow Transformer	2	1	64	4	200	800

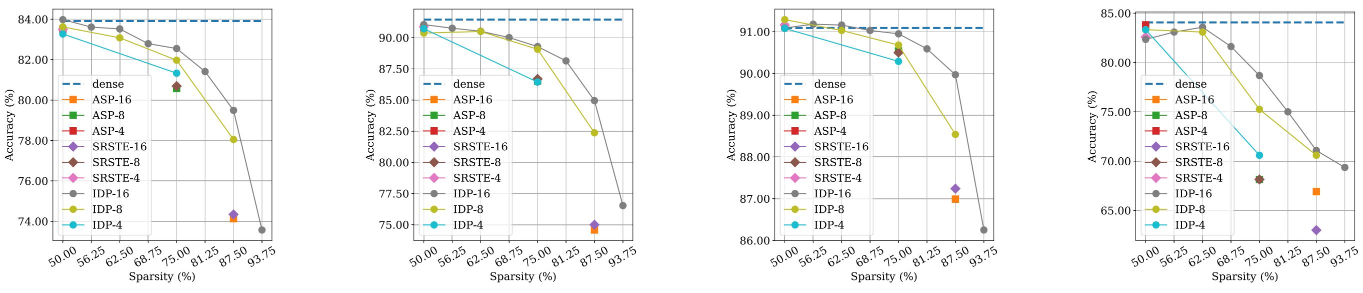
2) Implementation Details: For algorithm implementation (Set I), the pretrained models, the scripts, and datasets are provided by the HuggingFace repository [40]. All models are implemented and executed using PyTorch v1.5.

As for hardware implementation (Sets II and III), all modules of STA are designed in synthesizable SystemVerilog with the aid of hardware components from the BaseJump standard template library [41] and the PULP platform [42]. Xilinx Vivado 2018.2 is the tool for synthesis and implementation. We implement STA on three types of FPGA devices with various scales, including Xilinx ZYNQ Z7020 (XC7Z020), Xilinx Virtex-7 FPGA (XC7VX485T), and Xilinx UltraScale+ FPGA (XCVU13P). Specifically, XC7Z020 is a low-cost and low-resource System-on-Chip device equipped with a dual-core ARM Cortex-A9 processor and FPGA, which is fabricated in the 28-nm technology node. XC7VX485T is a relatively large FPGA device fabricated in the 28-nm technology node. XCVU13P, fabricated in the 16-nm technology node, is an extremely expensive and advanced FPGA device with abundant hardware resources.

B. Benchmark Set I: Algorithm Optimizations

For benchmark set I, ASP [9] and SR-STE [7], the two existing methods for acquiring *N:M* sparse models, are selected as our baselines. The reported accuracy of baselines is obtained by training with released open-source code. For a fair comparison, the generated *N:M* sparse models using ASP, SR-STE, and our method are achieved with identical fine-tune epochs. For all tasks, we use a batch size of 32 and an initial learning rate of $2e-5$. For WNLI, QNLI, and QQP, there are three epochs to recover accuracy for every step of N, while there are five epochs for MRPC.

Compared with existing methods, Fig. 12 shows that IDP can achieve comparable or better accuracy under a 75.00% sparse ratio. In addition, the IDP can outperform the ASP and SR-STE methods significantly with the sparse ratio increases.



(a) (b) (c) (d)

Fig. 12. Pruning results on various tasks, including (a) MNLI, (b) QNLI, (c) QQP, and (d) MRPC in comparison with ASP [9] and SR-STE [7].

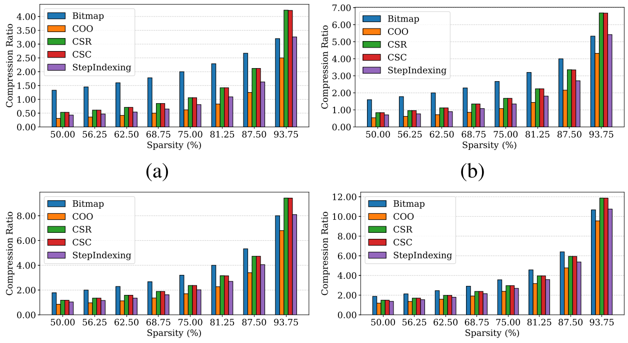
For instance, we can find that, under 87.50% (2:16) sparse ratio, IDP consistently obtains large performance improvements to the baseline on all tasks (5.36% accuracy gain on MNLI, 10.38% accuracy gain on QNLI, 2.98% accuracy gain on QQP, and 8.08% accuracy gain on MRPC). Therefore, we can obtain the state-of-the-art *N:M* sparse models for FPGA-based platform deployment with the plug-and-play IDP algorithm.

Based on our evaluations of model accuracy with respect to parameter sparsity, as shown in Fig. 12, it is observed that Transformers can hardly achieve a sparsity over 90% without impacting accuracy. It would be more likely practical for *N:M* sparse Transformers with a sparsity ranging from 50% to 87.5%. Next, BERT is taken as an example to evaluate the storage reduction of our compression scheme when using multiple quantized bits under various *N:M* sparsity configurations. We make an elaborated comparison between our bitmap-based scheme, COO, CSR, CSC, and step indexing [36]. Compared with the other mainstream methods, as shown in Fig. 13, our scheme can achieve the highest CR when the model sparsity is varied from 50% to 87.5%. The CR keeps increasing as the model sparsity increases. An *N:M* sparse BERT can achieve a higher CR when quantized in larger bitwidths. BERT with 50.00% *N:M* sparsity can reach a 1.78× reduction on storage of parameters. When BERT has a sparsity of 87.50%, it achieves a significant storage saving, up to 5.33×, on parameters. Our compression scheme can efficiently reduce the storage requirement for *N:M* sparse parameters. In subsequent hardware evaluations, we uniformly adopt a 16-bit fixed-point representation for Transformers to avoid negative impacts on model accuracy due to quantization.

C. Benchmark Set II: Hardware Resource Consumption

The second benchmark set verifies the hardware consumption of STAs that have not yet deployed Transformer models.

First, we evaluate the hardware requirements of DMME in common sparse configurations by comparing it against multiple dense computing engines. For a fair comparison, evaluated computing engines are all synthesized as a 2 × 2 unified MatMul computing engine, and only PEs in these engines are configured into various *N:M* configurations. Note that those computing engines that make N equal to M exclude the NZESs and merely support dense computing. It can be regarded as the computing engine in [10] if both N and M are 1 in the evaluated DMME. These computing engines are not allowed to be synthesized using DSP blocks, which can be done by setting the property *MAX_DSP* as zero. Hence, the utilization of LUTs and FFs from Vivado synthesis reports can be used to measure the consumption of combinational logic and sequential logic for DMMEs, respectively.

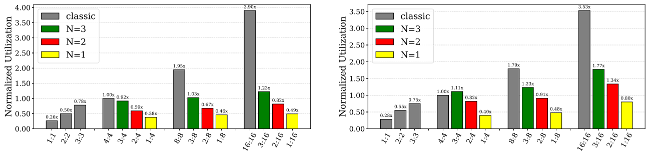


(a) (b) (c) (d)

Fig. 13. CR on BERT with various sparsity configurations. (a) Bitwidth = 4 bit. (b) Bitwidth = 8 bit. (c) Bitwidth = 16 bit. (d) Bitwidth = 32 bit.

Fig. 14 presents the comparison of required hardware resource consumption between DMMEs of various configurations. For simplicity, all results are normalized to the 4:4 dense baseline computing engine. Gray bars are resource consumption of various computing engines with sole support on dense MatMul. Green, red, and yellow bars represent hardware utilization of DMMEs when N is set as 1, 2, and 3, respectively. In Fig. 14, we can observe hardware resources saved by DMMEs compared to dense baseline computing engine under sparse matrix computing mode. When $M = 16$ and N is set to 1, 2, and 3, respectively, DMME, in contrast to 16:16 baseline, obtains saving of combinational logics up to 7.96×, 4.76×, and 3.17× while achieving reduction on sequential logics 4.41×, 2.63×, and 1.99×. According to Fig. 14, we further evaluate the impact of separately increasing N and M in DMMEs on hardware resource consumption. For instance, 3:4 DMME costs 2.42× combinational logic and 2.78× sequential logic of 1:4 DMME. However, 1:16 DMME merely requires 1.28× combinational logic and 2.00× sequential logic over 1:4 DMME.

We, finally, evaluate the hardware resource consumption of STA on three types of FPGA platforms, including XC7Z020,



(a)

(b)

Fig. 14. Normalized resource consumption of unified computing engine over the classic computing engine on various scales, including (a) combinational logic and (b) sequential logic.

TABLE III
FPGA Resource Utilization

Platform	Frequency	LUT	FF	BRAM	DSP
1:8 STA-Tiny (XC7Z020)	150MHz	21K (40.38%)	75K (71.21%)	96 (68.57%)	132 (60.00%)
2:8 STA-Small (XC7VX485T)	200MHz	116K (38.42%)	337K (55.52%)	532 (51.65%)	1,040 (37.14%)
2:8 STA-Large (XCVU13P)	200MHz	464K (26.88%)	1,321K (38.24%)	1,192 (44.35%)	4,160 (33.85%)

XC7VX485T, and XCVU13P. These FPGA platforms are used to represent diverse devices in Fig. 2. Considering the hardware resource and cost on these platforms, we intend to deploy XC7Z020 to the edge and XC7VX485T and XCVU13P on the clouds. There are many tunable parameters of STAs, especially in DMME, which have great impacts on performance. In order to determine the specific parameters of STA on each FPGA platform, we design a cycle-accurate simulator to evaluate actual inference performance based on the given specifications. STA in XC7Z020 adopts an aggressive 1:8 sparsity since latency is the critical metric on the edge platforms. However, STAs in both XC7VX485T and XCVU13P configured *N:M* as 2:8 because devices deployed on the clouds concern latency and model accuracy. Table III shows resource consumption of STAs deployed on three scales of FPGA platforms, namely, STA-Tiny, STA-Small, and STA-Large, respectively. The FPGA resource and power breakdown of STA-Small are presented in Table IV. The N-MAC module dominates the DSP consumption of STA since it is the core of the computing engine for MAC operations. The NZES module takes the majority of LUT consumption due to the cost of decoding and index selection. The routing module occupies most registers of DMME for datapath selection and temporary data storage. Moreover, the proposed DMME and softmax module occupy 47.29% and 9.42% power consumption, respectively.

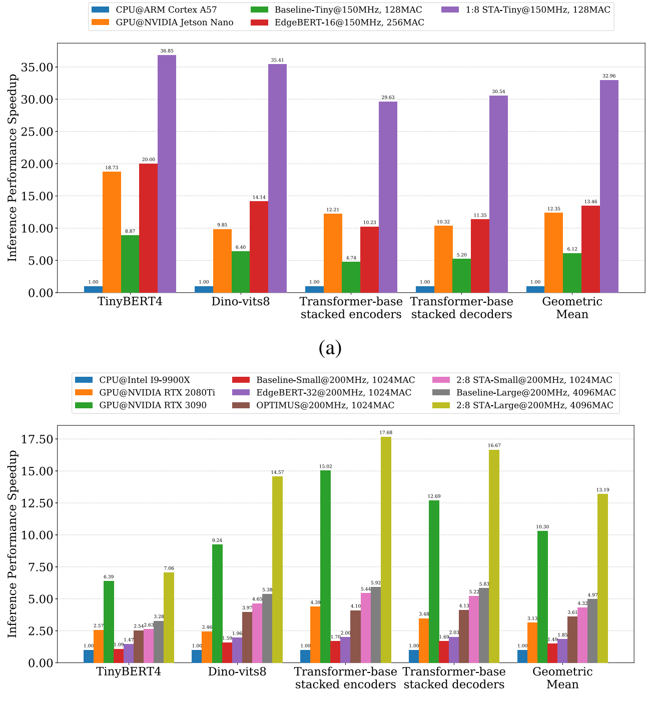
D. Benchmark Set III: Overall System Evaluation

The third benchmark set is used to evaluate performance when deploying various Transformer-based models on STAs.

First, we study the inference speedup of STAs by contrast with CPUs, GPUs, and the prior dedicated accelerators. The selected models to be deployed are composed of TinyBERT [38], Dino [39], and the classic Transformer model [11]. Here, we consider single-batch processing time in all the MHA and FFN ResBlocks of these Transformer-based models. For cross-platform comparison, the hardware setup is as follows to execute the Transformer inference tasks. The CPU results are measured using an ARM Cortex A57 and an Intel i9-9900X. The former commonly appeared in mobile devices for edge applications, while the latter is a high-end CPU product for deploying cloud applications. The GPU results are measured using an NVIDIA Jeston Nano, an embedded GPU product for edge applications, an NVIDIA RTX 2080Ti, and an NVIDIA RTX 3090 capable of 2:4 sparse acceleration. Following the comparison method in [32], we apply [10] as baselines on FPGA platforms by scaling the size of its computing engine. Two existing sparse accelerators for Transformers, OPTIMUS [5] and EdgeBERT [6], are evaluated as well for a more comprehensive comparison of STA. Fig. 15 shows the idealized performance speedup of different hardware platforms, where edge and cloud platforms are normalized to the ARM Cortex A57 and Intel i9-9900X, respectively.

TABLE IV
FPGA Resource and Power Breakdown of STA-Small

		LUT	FF	BRAM	DSP	Power (W)
DMME	N-MAC	16K (13.79%)	68K (20.18%)	-	**1024** **(98.46%)**	2.78 (28.17%)
	NZES	**66K** **(56.90%)**	68K (20.18%)	-	-	0.83 (8.41%)
	Routing	9K (7.76%)	**180K** **(53.41%)**	-	-	1.06 (10.74%)
Softmax		13K (11.21%)	8K (2.37%)	16 (3.00%)	16 (1.54%)	0.93 (9.42%)
Others		12K (10.34%)	13K (3.86%)	**516** **(97.00%)**	-	**4.27** **(43.26%)**
Total		116K (100.00%)	337K (100.00%)	532 (100.00%)	1040 (100.00%)	9.87 (100.00%)



(a)

(b)

Fig. 15. Processing time of Transformer-based models on various (a) edge platforms and (b) cloud platforms.

TABLE V

COMPARISON OF STAs WITH PREVIOUS WORKS AND COMMERCIAL PRODUCTS

Platform	CPU	GPU			FPGA					
	i9-9900X	Jetson Nano	RTX 2080 Ti	RTX 3090	SOCC'20 [10]	ISLPED'20 [29]	ISQED'21 [33]	Our work		
								STA-Tiny	STA-Small	STA-Large
Chip	Skylake	Tegra X1	TU102	GA102	XCVU13P	XCVU9P	XCU200	XC7Z020	XC7VX485T	XCVU13P
Technology	14 nm	20 nm	12 nm	8 nm	16 nm	16 nm	16 nm	28 nm	28 nm	16 nm
Frequency	3.50 GHz	640 MHz	1.35 GHz	1.70 GHz	200 MHz	-	-	150 MHz	200 MHz	200 MHz
Methods	-	-	-	2:4 group-based pruning	Low-bit quantization	Block-circulant matrix with FFT	Block-based pruning	N:M group-based pruning		
# MAC units	-	-	-	-	4096	∼ 5647	∼ 3368	128	1024	4096
Bit Precision	FP-32	FP-32	FP-32	FP-32	FIX-8	FIX-16	-	FIX-16		
Test Network	Shallow Transformer									
Latency (ms)	2.17	16.24	1.70	0.46	0.30	2.94	0.32	2.01	0.42	0.15
Batch-1 Throughput (GOP/s)	101.38	13.55	129.41	478.26	733.33	75.34	687.50	109.45	523.81	1466.67
Power (W)	165.00	7.56	250.00	350.00	16.70	22.45	-	2.71	9.87	26.59
Energy Efficiency (GOP/J)	0.61	1.79	0.52	1.37	43.91	3.35	-	40.39	53.07	55.16
MAC Efficiency (GOP/s/unit)	-	-	-	-	0.18	∼ 0.01	∼ 0.20	0.86	0.51	0.36

As shown in Fig. 15(a), among all edge platforms, STA-Tiny achieves a geometric mean increase of 32.96×, 2.69×, and 5.38× over CPU, GPU, and the FPGA baseline, respectively. Fig. 15(b) presents the performance comparison between various cloud platforms. For a fair comparison, Baseline-Small (red), EdgeBERT (purple), OPTIMUS (brown), and 2:8 STA-Small (pink) in Fig. 15(b) are evaluated under the same number of MAC units and clock frequency.

1) **STA Versus Baseline-Small [10]:** 2:8 STA-Small achieves 2.89× speedup on average over Baseline-Small, which enables dense Transformer acceleration using a large 2-D systolic array. It suffers low utilization of MAC units due to an inflexible mapping scheme and large skew latency for systolic-arranged data. STA achieves significant performance improvement by: 1) innovation of DMME from the architectural aspects and 2) reduction on MAC operations of *N:M* sparse Transformers from the algorithmic aspects. We further present a performance breakdown of STA compared to Baseline-Small. As shown in Fig. 16, the architectural innovation for DMME can achieve 1.08× better performance improvement, and there is a 2.68× speedup on top of the architectural innovation by efficiently enabling 2:8 sparse acceleration.
2) **STA Versus EdgeBERT [6]:** 2:8 STA-Small achieves 2.33× speedup on average over EdgeBERT-32, which is an energy-optimized Transformer accelerator exploiting unstructured sparsity. When performing sparse MAC operations, processing units of EdgeBERT skip the zero input value through a gating strategy, which can significantly reduce energy consumption, but offer little benefit to latency. Compared to EdgeBERT, STA performs *N:M* sparse MAC operations by choosing nonzero inputs, which can both reduce energy and optimize latency.
3) **STA Versus OPTIMUS [5]:** 2:8 STA-Small has a 1.20× better performance on average over OPTIMUS, which is a high-performance sparse accelerator for Transformers exploiting unstructured sparsity in weight parameters. When performing sparse MAC operations, OPTIMUS can hardly achieve high MAC utilization due to load imbalance and input load miss. STA can effectively overcome these two problems suffered by OPTIMUS. STA gets rid of load imbalance by arranging each MAC in DMME to perform operations in a balanced *N:M* group. In addition, STA loads a series of input *N:M* groups at each cycle and utilizes these elements multiple times in a systolic manner, which effectively addresses input load miss compared to OPTIMUS.

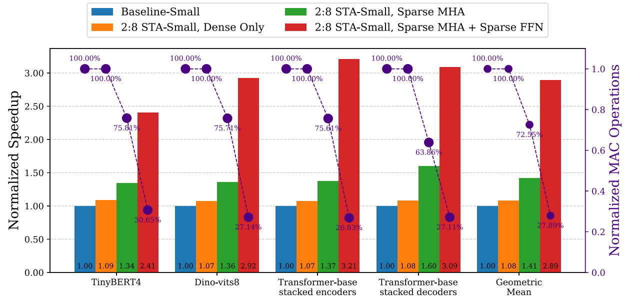


Fig. 16. Performance and MAC operation breakdown of STA-Small.

Finally, we compare STA with other previous FPGA-based works and commercial CPU and GPU products. Table V presents a fair performance comparison without batching on various platforms. Prior FPGA-based works for accelerating Transformers include [10], [29], and [33]. The dedicated accelerator in [10], equipped with a large 2-D systolic array for dense operation, is the pioneer work for Transformer acceleration. FTRANS [29] is another recently specialized accelerator for Transformers, which exploits the speedup potential of block-circulant weight representations. In [33], the proposed accelerator utilizes coarse-grained block-based sparsity to speed up Transformer inference. The shallow Transformer used in [29] and [33] is applied as a benchmark for a fair evaluation. The comparison is benchmarked on CPU, GPUs, prior cutting-edge FPGA solutions, and STA on various FPGA platforms. We evaluate these designs in terms of latency, throughput, power, energy efficiency, and MAC efficiency. All of them are key metrics for a computing system.

As shown in Table V, STA-Tiny far outperforms the embedded GPU, Jetson Nano, and the high-end CPU, i9-9900X, in all evaluated metrics. STA-Small surpasses the CPU and GPU platforms in all metrics. Moreover, STA-Small is close to [10] and [33] in terms of latency and throughput while using a relatively small number of MACs compared to them. The energy efficiency and MAC efficiency of STA-Small are also superior to all previous FPGA-based works. Compared to previous FPGA solutions, STA-Large achieves 2.00 ~ 19.47 × throughput improvement, 1.26 ~ 16.47 × energy efficiency improvement, and 1.80 ~ 36.00× MAC efficiency gain, respectively. The performance gain of STA comes from optimizations from two levels. At the algorithm level, we carefully exploit the potential of *N:M* sparsity pattern, which can significantly reduce the computational cost of Transformer-based models. At the hardware level, STA can efficiently handle *N:M* sparse parameters, which significantly improves the utilization of computing units. In addition, our deployment framework, taking STA-Tiny, STA-Small, and STA-Large as examples, can realize flexible hardware generation for Transformers. The proposed framework can flexibly and efficiently meet the requirements for deploying Transformer-based models on various FPGA devices.

VII. Conclusion

In this article, we present a flexible, agile, and efficient framework for deploying *N:M* sparse Transformers, which is benefited from both algorithm and hardware optimizations, making it practical to significantly accelerate Transformer-based models on diverse FPGA devices. At the algorithm level, we propose a sparsity inheritance mechanism and an IDP method to obtain a series of *N:M* sparse Transformers with high accuracy. A further proposed compression scheme greatly reduces the storage requirements of models. At the hardware level, we present a flexible and efficient architecture, namely, STA, to accelerate *N:M* sparse Transformers. STA is composed of a computing core, DMME, which unifies both sparse and dense intensive MatMuls in *N:M* sparse Transformers, and a scalable softmax module, which eliminates intermediate off-chip data accesses. The experimental results show that *N:M* sparse Transformers generated by IDP achieve an average of 6.7% improvement in accuracy over the state-of-the-art methods. STA implementation significantly outperforms CPU, GPU, and prior FPGA-based Transformer accelerators in terms of latency, throughput, energy efficiency, and MAC efficiency, showing its significant potential in applications using Transformer-based models.

Acknowledgment

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

References

[1] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, "Efficient transformers: A survey," 2020, *arXiv:2009.06732*.

[2] K. Song *et al.*, "Alignment-enhanced transformer for constraining NMT with pre-specified translations," in *Proc. AAAI Conf. Artif. Intell. (AAAI)*, vol. 34, no. 5, 2020, pp. 8886–8893.

[3] J. D. M.-W. C. Kenton and L. K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics Hum. Lang. Technol. (NAACL-HLT)*, 2019, pp. 4171–4186.

[4] A. Dosovitskiy *et al.*, "An image is worth 16×16 words: Transformers for image recognition at scale," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2021, pp. 1–21. [Online]. Available: https://openreview.net/pdf?id=YicbFdNTTy

[5] J. Park, H. Yoon, D. Ahn, J. Choi, and J.-J. Kim, "OPTIMUS: OPTImized matrix MUltiplication structure for transformer neural network accelerator," *Proc. Mach. Learn. Syst.*, vol. 2, pp. 363–378, Mar. 2020.

[6] T. Tambe *et al.*, "EdgeBERT: Sentence-level energy optimizations for latency-aware multi-task NLP inference," in *Proc. MICRO-54: 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2021, pp. 830–844.

[7] A. Zhou *et al.*, "Learning N:M fine-grained structured sparse neural networks from scratch," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2021, pp. 1–15. [Online]. Available:https://openreview.net/pdf?id=K9bw7vqp_s

[8] W. Sun *et al.*, "DominoSearch: Find layer-wise fine-grained N:M sparse schemes from dense neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 34, 2021, pp. 20721–20732.

[9] A. Mishra *et al.*, "Accelerating sparse deep neural networks," 2021, *arXiv:2104.08378*.

[10] S. Lu, M. Wang, S. Liang, J. Lin, and Z. Wang, "Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer," in *Proc. IEEE 33rd Int. Syst. Chip Conf. (SOCC)*, Sep. 2020, pp. 84–89.

[11] A. Vaswani *et al.*, "Attention is all you need," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2017, pp. 1–11.

[12] D. Wu, X. Fan, W. Cao, and L. Wang, "SWM: A high-performance sparse-winograd matrix multiplication CNN accelerator," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 5, pp. 936–949, May 2021.

[13] S. Colleman and M. Verhelst, "High-utilization, high-flexibility depth-first CNN coprocessor for image pixel processing on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 3, pp. 461–471, Mar. 2021.

[14] H. E. Yantir, A. M. Eltawil, and K. N. Salama, "IMCA: An efficient in-memory convolution accelerator," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 3, pp. 447–460, Mar. 2021.

[15] S. Yin, Z. Jiang, M. Kim, T. Gupta, M. Seok, and J.-S. Seo, "Vesti: Energy-efficient in-memory computing accelerator for deep neural networks," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 1, pp. 48–61, Jan. 2020.

[16] G. Paulin, R. Andri, F. Conti, and L. Benini, "RNN-based radio resource management on multicore RISC-V accelerator architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 9, pp. 1624–1637, Sep. 2021.

[17] C. Fang, L. He, H. Wang, J. Wei, and Z. Wang, "Accelerating 3D convolutional neural networks using 3D fast Fourier transform," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2021, pp. 1–5.

[18] Y. Yu, T. Zhao, M. Wang, K. Wang, and L. He, "Uni-OPU: An FPGA-based uniform accelerator for convolutional and transposed convolutional networks," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 7, pp. 1545–1556, Jul. 2020.

[19] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang, and H. Shen, "An efficient hardware accelerator for structured sparse convolutional neural networks on FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 9, pp. 1953–1965, Sep. 2020.

[20] A. Moreno, J. Olivito, J. Resano, and H. Mecha, "Analysis of a pipelined architecture for sparse DNNs on embedded systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 9, pp. 1993–2003, Sep. 2020.

[21] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance FPGA-based CNN accelerator with block-floating-point arithmetic," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1874–1885, Aug. 2019.

[22] S. Kala, B. R. Jose, J. Mathew, and S. Nalesh, "High-performance CNN accelerator on FPGA using unified winograd-GEMM architecture," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 12, pp. 2816–2828, Dec. 2019.

[23] X. Xie, J. Lin, Z. Wang, and J. Wei, "An efficient and flexible accelerator design for sparse convolutional neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 7, pp. 2936–2949, Jul. 2021.

[24] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 1–13.

[25] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 367–379.

[26] S. Liu *et al.*, "Cambricon: An instruction set architecture for neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 393–405.

[27] Z.-G. Liu, P. N. Whatmough, and M. Mattina, "Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference," *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 34–37, Jan. 2020.

[28] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 27–40.

[29] B. Li *et al.*, "FTRANS: Energy-efficient acceleration of transformers using FPGA," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design*, Aug. 2020, pp. 175–180.

[30] T. J. Ham *et al.*, "A^3: Accelerating attention mechanisms in neural networks with approximation," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 328–341.

[31] H. Wang, Z. Zhang, and S. Han, "SpAtten: Efficient sparse attention architecture with cascade token and head pruning," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2021, pp. 97–110.

[32] L. Lu *et al.*, "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," in *Proc. MICRO 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2021, pp. 977–991.

[33] H. Peng *et al.*, "Accelerating transformer-based deep learning models on FPGAs using column balanced block pruning," in *Proc. 22nd Int. Symp. Quality Electron. Design (ISQED)*, Apr. 2021, pp. 142–148.

[34] T. Lin, S. U. Stich, L. Barba, D. Dmitriev, and M. Jaggi, "Dynamic model pruning with feedback," 2020, *arXiv:2006.07253*.

[35] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *J. Mach. Learn. Res.*, vol. 22, pp. 1–124, Sep. 2021.

[36] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[37] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *Proc. EMNLP Workshop BlackboxNLP, Analyzing Interpreting Neural Netw. (NLP)*, 2018, pp. 353–355.

[38] X. Jiao *et al.*, "TinyBERT: Distilling BERT for natural language understanding," in *Proc. Findings Assoc. Comput. Linguistics, (EMNLP)*, 2020, pp. 4163–4174.

[39] M. Caron *et al.*, "Emerging properties in self-supervised vision transformers," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2021, pp. 9650–9660.

[40] T. Wolf *et al.*, "HuggingFace's transformers: State-of-the-art natural language processing," 2019, *arXiv:1910.03771*.

[41] M. B. Taylor, "Basejump STL: Systemverilog needs a standard template library for hardware design," in *Proc. 55th Annu. Design Autom. Conf.*, Jun. 2018, pp. 1–6.

[42] D. Rossi *et al.*, "PULP: A parallel ultra low power platform for next generation IoT applications," in *Proc. IEEE Hot Chips 27 Symp. (HCS)*, Aug. 2015, pp. 1–39.

Chao Fang (Graduate Student Member, IEEE) received the B.S. degree from Tianjin University, Tianjin, China, in 2019. He is currently working toward the Ph.D. degree at Nanjing University, Nanjing, China.

His current research interests include model compression algorithms and domain-specific accelerator design for machine learning.

Aojun Zhou received the M.S. degree from the National Laboratory of Pattern Recognition (NLPR), Institute of Automation, Chinese Academy of Sciences (CASIA), Beijing, China, in 2019. He is currently working toward the Ph.D. degree at the Department of Electronic Engineering, The Chinese University of Hong Kong, Hong Kong.

His research interests include deep learning and computer vision.

Zhongfeng Wang (Fellow, IEEE) received the B.E. and M.S. degrees from the Department of Automation, Tsinghua University, Beijing, China, in 1988 and 1990, respectively, and the Ph.D. degree from the University of Minnesota, Minneapolis, MN, USA, in 2000.

He was a Leading VLSI Architect with Broadcom Corporation, San Jose, CA, USA, from 2007 to 2016. He worked at Oregon State University, Corvallis, OR, USA, and National Semiconductor Corporation, Fremont, CA, USA. He has been a Distinguished Professor with Nanjing University, Nanjing, China, since 2016. He is a world-recognized expert on low-power high-speed VLSI design for signal processing systems. He has published over 200 technical papers with multiple best paper awards received from the IEEE technical societies, among which is the VLSI Transactions Best Paper Award of 2007. He has edited one book VLSI and held more than 20 U.S. and China patents. In the current record, he has had many papers ranking among the top 25 most (annually) downloaded manuscripts in IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS (T-VLSI). Moreover, he has contributed significantly to industrial standards. So far, his technical proposals have been adopted by more than 15 international networking standards. His current research interests are in the area of optimized VLSI design for digital communications and deep learning.

Dr. Wang has served as a TPC member and on various chairs for tens of international conferences. In 2015, he was elevated to fellow of IEEE for contributions to VLSI design and implementation of forward error correction (FEC) coding. He has served as an Associate Editor for IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS I: REGULAR PAPERS, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II: EXPRESS BRIEFS, and IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS for many terms.