AccelTran: A Sparsity-Aware Accelerator for Dynamic Inference With Transformers

Shikhar Tuli, *Student Member, IEEE*, and Niraj K. Jha, *Fellow, IEEE*

***Abstract*—Self-attention-based transformer models have achieved tremendous success in the domain of natural language processing. Despite their efficacy, accelerating the transformer is challenging due to its quadratic computational complexity and large activation sizes. Existing transformer accelerators attempt to prune its tokens to reduce memory access, albeit with high compute overheads. Moreover, previous works directly operate on large matrices involved in the attention operation, which limits hardware utilization. In order to address these challenges, this work proposes a novel dynamic inference scheme, DynaTran, which prunes activations at runtime with low overhead, substantially reducing the number of ineffectual operations. This improves the throughput of transformer inference. We further propose tiling the matrices in transformer operations along with diverse dataflows to improve data reuse, thus, enabling higher energy efficiency. To effectively implement these methods, we propose AccelTran, a novel accelerator architecture for transformers. Extensive experiments with different models and benchmarks demonstrate that DynaTran achieves higher accuracy than the state-of-the-art top-*k* hardware-aware pruning strategy while attaining up to 1.2× higher sparsity. One of our proposed accelerators, AccelTran-Edge, achieves 330K× higher throughput with 93K× lower energy requirement when compared to a Raspberry Pi device. On the other hand, AccelTran-Server achieves 5.73× higher throughput and 3.69× lower energy consumption compared to the state-of-the-art transformer co-processor, Energon. The simulation source code is available at https://github.com/jha-lab/acceltran.**

***Index Terms*—Accelerators, application-specific integrated circuits (ASICs), machine learning (ML), natural language processing, neural networks, transformers.**

I. Introduction

The TRANSFORMER architecture [1], which is based on the self-attention mechanism [2], has gained widespread interest in the domain of natural language processing [3] and, recently, even in computer vision [4]. One reason is its massive parallelization capabilities on modern-day graphical processing units (GPUs), unlike traditional sequential models like long short-term memories [5] and recurrent neural networks [6] that are slow to train and, thus, may not perform as well. Transformers have been able to achieve state-of-the-art performance on diverse benchmarking tasks due to pretraining on massive public and private language corpora [7], [8], [9].

The massive models come with their own challenges. For instance, pretraining a large state-of-the-art model usually requires millions of dollars worth of GPU resources [10]. Furthermore, large transformer models also have a high memory footprint, making them challenging to train even on modern GPUs. Convolutional neural networks (CNNs) have been able to overcome these challenges with a plethora of application-specific integrated circuit (ASIC)-based accelerators, each specialized for a different set of models in its design space [11], [12]. These accelerators have specially designed hardware modules that leverage sparsity in model weights, data reuse, optimized dataflows, and CNN mapping to attain high performance and energy efficiency [13]. However, CNN accelerators are incompatible with transformer workflows since they are optimized for the inner-product operation, the basis of a convolution operation, and not for matrix–matrix multiplication control flows.

Some recent works attempt to accelerate transformers by reducing their memory footprint and the compute overhead of the self-attention operation. For instance, A^3 [14] contains several approximation strategies to avoid computing attention scores close to zero. SpAtten [15] leverages a cascade token pruning mechanism that progressively prunes unimportant tokens based on low attention probabilities, reducing overall compute complexity. However, the proposed "top-*k*" pruning mechanism [15], a state-of-the-art hardware-aware dynamic inference method, has a high compute overhead, which partially offsets its throughput gains during model inference according to our experiments (details in Section V-A). Energon [16] approximates the top-*k* pruning method with its mixed-precision multiround filtering algorithm. However, it only exploits sparsity in the attention probabilities, not in all possible multiplication operations in the transformer architecture (details in Section II-B). To tackle this problem, OPTIMUS [17] uses a set-associative rearranged compressed sparse column (SA-RCSC) format to eliminate ineffectual multiply-and-accumulate (MAC) operations. However, it only exploits sparsity in the weight matrices and not the activations, i.e., the matrices formed from intermediate MAC operations. It also only works with encoder–decoder models, where the decoder is known to support limited parallelism. Leveraging encoder-only models, which have recently shown to perform well even on translation and language generation tasks [18], [19], not only reduces the critical path by 2× but also improves hardware utilization. Further, these works

Manuscript received 24 September 2022; revised 27 February 2023 and 24 April 2023; accepted 1 May 2023. Date of publication 8 May 2023; date of current version 20 October 2023. This work was supported by NSF under Grant CCF-2203399. This article was recommended by Associate Editor P. A. Beerel. *(Corresponding author: Shikhar Tuli.)*

The authors are with the Department of Electrical and Computer Engineering, Princeton University, Princeton, NJ 08544 USA (e-mail: stuli@princeton.edu; jha@princeton.edu).

Digital Object Identifier 10.1109/TCAD.2023.3273992

implement an entire matrix multiplication over an array of processing elements (PEs), which are the basic compute blocks of an accelerator. OPTIMUS [17], with its SA-RCSC sparse matrix format, does not break down the matrices involved into multiple *tiles* [implemented in general matrix multiplication (GEMM) pipelines] in order to improve hardware utilization. FTRANS [20] and SpAtten [15] break down a matrix–matrix multiplication operation into multiple matrix–vector multiplication operations, losing out on data reuse capabilities. This also limits the scope of parallelization (details in Section V-A). Data reuse, parallelization, and optimal hardware utilization are crucial to obtaining high throughput and energy efficiency. Energon [16] is a co-processor and not a full-fledged accelerator. This limits the scope of optimization across the entire pipeline, resulting in superfluous off-chip accesses. Field-programmable gate array (FPGA)-based transformer accelerators have also been proposed owing to their low cost [20], [21], [22]. However, they suffer from performance and power inefficiencies due to bit-level reconfigurable abstractions and correspondingly high interconnect overheads [23].

To overcome the above challenges, we propose AccelTran, a novel cycle-accurate accelerator for transformer models. Our main contributions are as follows.

1) We propose a granular and hardware-aware dynamic inference framework, DynaTran, for transformers that dynamically prunes all activations in order to remove ineffectual MAC operations. DynaTran has much less compute overhead compared to previous works [15], [16], enabling higher throughput for model inference.
2) To *efficiently* execute DynaTran, we design and implement an ASIC-based architecture called AccelTran. Instead of using traditional encoder–decoder models [17], we leverage recently proposed encoder-only models [1], thus, reducing the critical path by 2× and improving throughput and hardware utilization. Further, unlike previous works [16], AccelTran's dynamic inference pipeline is agnostic to the preprocessed weight pruning (WP) strategy.
3) We propose the use of *tiled* matrix multiplication for our transformer accelerator. For this, we leverage a novel mapping scheme from the transformer model to the tiled operations that maximizes hardware utilization and improves parallelization.
4) We also formulate and implement, for the first time, various *dataflows* for the transformer optimal dataflow that maximizes data reuse to improve energy efficiency.
5) We further leverage monolithic-3D RRAM [24] for higher memory bandwidth. This alleviates the performance bottleneck in transformer inference since state-of-the-art models are huge and, thus, memory bound [8], [25]. Our proposed control block maps the transformer computational graph to scheduled hardware-implementable operations. It leverages the high-bandwidth monolithic-3D RRAM to schedule these operations intelligently, enabling high throughput and energy efficiency. We also support LP-DDR3 memory for low-cost edge solutions.

TABLE I
MEMORY AND COMPUTE OPERATIONS IN A TRANSFORMER

Word Embedding and Position Encoding	
M-OP-0	$\mathbf{H} = \mathbf{H}_{emb} + \text{PE}(\mathbf{H}_{emb})$
Multi-Head Attention	
M-OP-[1-4]	load $\mathbf{W}_i^{\text{Q}}, \mathbf{W}_i^{\text{K}}, \mathbf{W}_i^{\text{V}}, \mathbf{W}_i^{\text{O}}$
C-OP-[1-3]	$\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i = \mathbf{H}\mathbf{W}_i^{\text{Q}}, \mathbf{H}\mathbf{W}_i^{\text{K}}, \mathbf{H}\mathbf{W}_i^{\text{V}}$
C-OP-4	$\mathbf{A}_i = \mathbf{Q}_i\mathbf{K}_i$
C-OP-5	$\mathbf{S}_i = \text{softmax}\left(\frac{\mathbf{A}_i}{\sqrt{h}}\right)$
C-OP-6	$\mathbf{P}_i = \mathbf{S}_i\mathbf{V}_i$
C-OP-7	$\mathbf{H}_i^{\text{MHA}} = \mathbf{P}_i\mathbf{W}_i^{\text{O}}$
Add and Layer-norm	
C-OP-8	$\mathbf{H}^{\text{LN}} = \text{layer-norm}(\mathbf{H}^{\text{MHA}} + \mathbf{H})$
Feed Forward	
M-OP-[5-6]	load $\mathbf{W}^{\text{F1}}, \mathbf{W}^{\text{F2}}$
C-OP-9	$\mathbf{H}^{\text{F1}} = \text{GeLU}(\mathbf{W}^{\text{F1}}\mathbf{H}^{\text{LN}})$
C-OP-10	$\mathbf{H}^{\text{F2}} = \text{GeLU}(\mathbf{W}^{\text{F2}}\mathbf{H}^{\text{F1}})$
Layer-norm	
C-OP-11	$\mathbf{H}^{\text{O}} = \text{layer-norm}(\mathbf{H}^{\text{F2}})$

The rest of this article is organized as follows. Section II presents background on transformer acceleration. Section III illustrates the methodology underpinning the DynaTran and AccelTran frameworks in detail. Section IV describes the experimental setup and baselines that we compare against. Section V discusses the results. Section VI compares related works and suggests future work directions. Finally, Section VII concludes this article.

II. BACKGROUND AND MOTIVATION

In this section, we provide background on various compute operations employed in a transformer model and previous works on transformer pruning and dynamic inference (sometimes interchangeably termed as dynamic pruning [15], [16]).

A. *Transformer Model*

We present the details of the memory and compute operations in the transformer model next.

1) Compute Operations: Table I summarizes the required memory load and compute operations in a transformer model. The first is the loading of word embeddings and position encodings, which take up a significant fraction of the weights in a transformer. Here, $\mathbf{H}_{emb}$ corresponds to the embeddings of all tokens in the vocabulary (vocabulary size is 30 522 for the BERT [1] family of models). We represent each token by a vector of length h, which is the hidden dimension of the transformer (e.g., $h = 128$ for BERT-Tiny [26] and $h = 768$ for BERT-Base [1]). Then, we load the weight matrices for the multihead attention operations. Here, $\mathbf{W}_i^{\text{Q}}$, $\mathbf{W}_i^{\text{K}}$, and $\mathbf{W}_i^{\text{V}} \in \mathbb{R}^{h \times h/n}$ are needed in each attention head, where n is the number of attention heads. Subsequent compute operations [color-coded **blue** for matrix multiplication and **green** for softmax (SMX)] are employed in self-attention [2]. Intermediate matrices are called *activations*; those that are loaded from

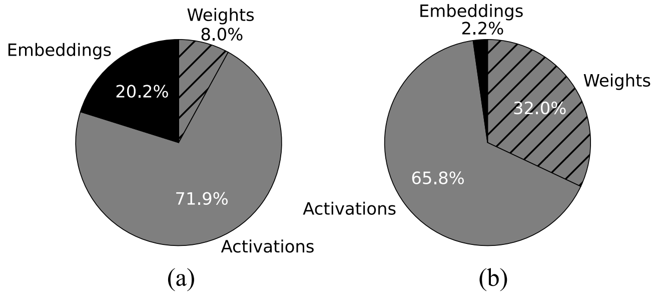


Fig. 1. Memory requirements for (a) BERT-Tiny and (b) BERT-Base.

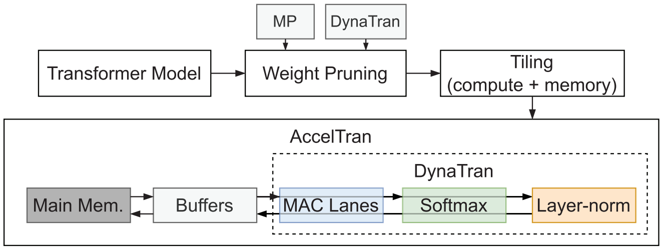


Fig. 2. AccelTran workflow for an input transformer model and its acceleration in hardware.

memory are called *weights*. $\mathbf{W}_i^{\mathrm{O}} \in \mathbb{R}^{h/n \times h/n}$ maps the attention probabilities to output scores. Then, we add the input to the output of the multihead attention (which is formed by concatenating the output of all attention heads) and normalize the resultant matrix. This is the layer-norm operation (color-coded **orange**) that is used to reduce covariance shifts [27]. Finally, the layer norm feeds the feedforward operation that, in turn, feeds the layer norm. GeLU is the activation function commonly used in transformers [1], [28].

2) Memory Requirements: Fig. 1 shows the memory requirements for BERT-Tiny and BERT-Base. BERT-Tiny has higher memory requirements for word and position embeddings (compared with BERT-Base) relative to requirements for weights and activations. Further, activations take up much memory, 8.98× that of the weights for BERT-Tiny and 2.06× for BERT-Base. The total main memory requirements for the two models are 52.8 MB and 3.4 GB, respectively, when only the weights and embeddings are stored. Activations are formed at runtime and stored in internal registers or on-chip buffers. With increasing transformer model sizes (calculated solely in terms of weights) [8], taking into account their operation on hardware accelerators, the memory budget should also have to account for the commensurate increase in activations.

B. Sparsity in Self-Attention

Researchers have striven to reduce the computational complexity of transformers by pruning, during pretraining or fine-tuning, the transformer weights [29], [30]. Previous works have also proposed various methods to reduce the quadratic complexity of the self-attention operation [31]. Distillation [32] recovers the accuracy loss due to such pruning techniques. However, all these works prune the model while training; more so, they only prune the weights. During inference, sparse matrices with ineffectual values may be formed *dynamically* from both activations and weights. Such ineffectual values must be pruned at runtime to improve energy efficiency and hardware utilization.

SpAtten [15] proposed the top-k pruning method. It essentially identifies query-key pairs that produce large attention probabilities at runtime. Given an attention score matrix ($\mathbf{S}_i$ in Table I), it keeps the k largest elements in each row to obtain the probability matrix ($\mathbf{P}_i$) and neglects the rest. Even though this method only results in a minor accuracy loss, it has a high overhead (as we show experimentally in Section V-A) due to its $\mathcal{O}(N^3)$ complexity. Further, a matrix multiplication operation benefits from sparsification when small values, which do not have much effect on the final result, are completely pruned out so that the hardware does not have to implement the corresponding MAC operations. SpAtten only considers the attention probabilities ($\mathbf{P}_i$), but not all the matrix multiplication operations presented in Table I. Thus, it loses out on gains that could be obtained by pruning other matrices as well. We compare it with our proposed method, DynaTran, in Section V-A.

III. Methodology

Fig. 2 presents a flowchart for the AccelTran simulation pipeline. We first weight-prune the transformer that is provided as input, either using movement pruning (MP) [30] or DynaTran. Then, we tile the transformer model into granular compute and memory operations. These tiled operations are passed to the AccelTran simulator, which implements the tiled operations, in hardware, in a cycle-accurate manner.

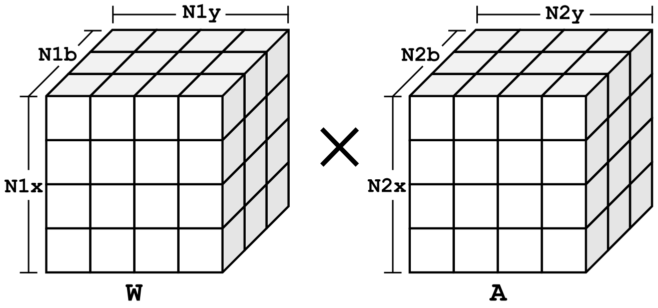
We now present the DynaTran framework for efficient dynamic inference with the transformer model. We also present AccelTran, a cycle-accurate accelerator for implementing this framework efficiently in hardware.

A. DynaTran

Unlike the top-k pruning algorithm [15], we propose a low-overhead dynamic inference method that quickly prunes ineffectual weight and activation values at runtime. For a given matrix, which is either loaded as a weight matrix from memory or is an activation matrix obtained from previous MAC operations, DynaTran prunes values with a magnitude less than a given threshold τ. Mathematically, an input matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$ is pruned to $\mathbf{M}^P$ as follows:

$$\mathbf{M}_{ij}^P = \begin{cases} \mathbf{M}_{ij}, & \text{if } |\mathbf{M}_{ij}| \geq \tau \\ 0, & \text{if } |\mathbf{M}_{ij}| < \tau. \end{cases}$$

This simple comparison operation incurs negligible compute overhead at runtime. This is important since transformer evaluation involves many such matrices at runtime, most of which are on the critical path for model computation. Further, each comparison operation can be parallelized, ensuring that pruning only takes up one clock cycle. This has a much lower overhead compared to SpAtten [15] and Energon [16] that have dedicated engines for this operation. We now define the



```
assert (N1b == N2b)
assert (N1y == N2x)
for (int b = 0; b < N1b; b++):
  for (int i = 0; i < N1x; i++):
    for (int j = 0; j < N2y; j++):
      for (int k = 0; k < N2x; k++):
        O[b,i,j] += W[b,i,k] × A[b,k,j]
```

Fig. 3. Tiling of a matrix multiplication operation along with a selected dataflow (specifically, **[b,i,j,k]**). Here, a tensor is shown instead, with the first dimension being the batch size.

pruning ratio (or level of sparsity) for the output matrix as

$$\rho\left(\mathbf{M}^P\right) = \frac{\sum_{x\in\mathbf{M}^P} \delta_{x,0}}{m \times n}$$

where δ is the Kronecker delta function. We profile the resultant sparsity in the weights and activations for different transformer models on diverse applications to obtain a desired ρ. One or more such profiled curves can be stored in memory. For the desired values of ρ, we determine the corresponding τ at runtime through a simple look-up operation. We present such curves in Section V-A to compare the throughput of our proposed approach with top-k pruning.

B. AccelTran Simulator

We present details of the proposed accelerator simulator next.

1) Tiling and Dataflow: As per Table I, most compute operations in the transformer model are matrix multiplication operations. Thus, it is important to optimize these operations for high gains. Unlike previous works that perform matrix multiplications directly using large MAC units, we propose using tiled matrix multiplication (primarily employed by modern GPUs [33]). Tiling the operations helps with better utilization of resources and enables massive parallelization. Fig. 3 shows the tiling operation along with an example *dataflow*. We can also think of a dataflow as a loop-unrolling scheme. The four for-loops can be unrolled in any permutation (giving 24 possible ways to unroll the loops, i.e., 24 dataflows). Multiplication between two tiles (say, weights **W[b,i,k]** and activations **A[b,k,j]**) is performed by an MAC lane (in parallel, based on the number of MAC units).

Each dataflow results in different data reuse capabilities. For example, if only four MAC lanes are available, with the dataflow shown in Fig. 3, when **j** changes from **0** to **1** (**b** and **i** remaining constant), the MAC lanes can reuse the corresponding weights **W[b,i,k]**, **k** ∈ **[0,...,N2x]**. Similarly, other dataflows would result in different reuse capabilities for different input matrix sizes. We show the reuse instances and corresponding energy savings for this example in Section V-B. No previous work has leveraged different dataflows to improve data reuse in transformer evaluation.

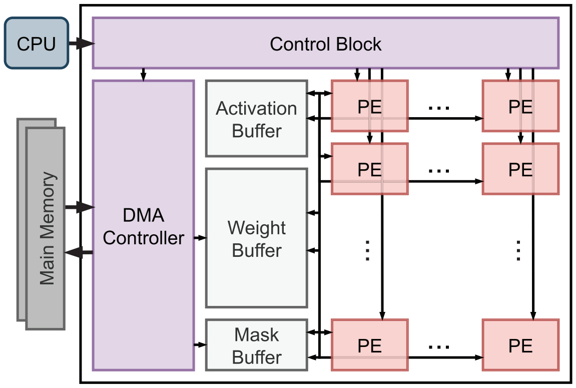


Fig. 4. Accelerator organization.

2) Accelerator Organization: Taking inspiration from a state-of-the-art CNN accelerator, SPRING [12], we leverage monolithic-3D integration to connect to an on-chip 3-D resistive random-access memory (RRAM) [24]. In monolithic-3D integration, multiple device tiers are fabricated on one substrate wafer, connected through monolithic intertier vias that allow much higher density than traditional through-silicon-via-based 3-D integration [34]. This leaves much more space for logic and also permits high memory bandwidth, which are crucial for large state-of-the-art transformer models. For scalable edge deployments, we also support an off-chip dynamic RAM (DRAM).

Fig. 4 shows the organization of the accelerator tier in the proposed architecture. The control block takes the instruction stream for the transformer model from the host CPU. The weights and embeddings are brought on-chip from the off-chip DRAM, or from the monolithic-3D RRAM, by the direct memory access (DMA) controller. The activation and the weight buffers store the activations and weights, respectively, in a compressed format (discussed in Section III-B6). Data compression relies on binary masks (stored in the mask buffer). The PEs use the compressed data and the associated masks to perform the main compute operations in the transformer.

3) Processing Elements: Fig. 5 shows the main modules present inside a PE, which is the basic compute block in our accelerator. The compressed data are stored in local registers of the PE by the activation first-in-first-out (FIFO) and weight FIFO registers. The data then enter the DynaTran module that induces sparsity based on the desired ρ. As explained in Section III-A, this module prunes the given weights or activations based on a precalculated threshold τ. The sparse data then enter the precompute sparsity module with the binary masks. This module converts the input data into a zero-free format based on the associated masks. The PE then forwards this zero-free data to the MAC lanes (for matrix multiplication), SMX modules (for SMX operation), or the layer-norm module (for layer-norm operation). The zero-free data eliminate any ineffectual computations in these modules. Finally, the

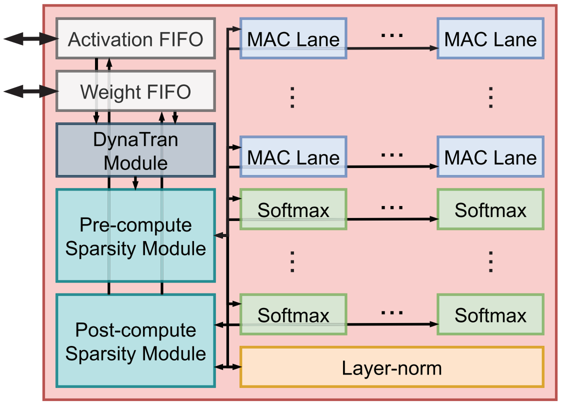


Fig. 5. Internal components of a PE.

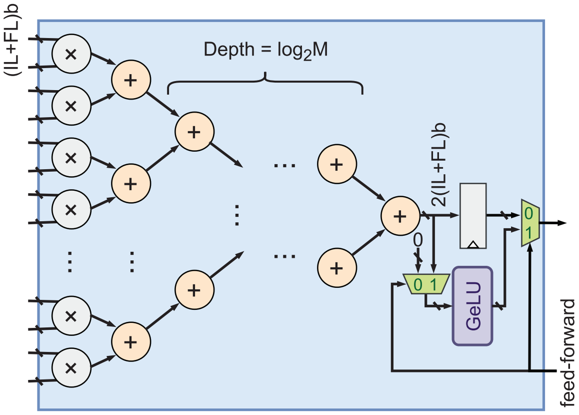


Fig. 6. Architecture of the MAC lane.

post-compute sparsity module implements the inverse of this operation on the output activations, before storing them in the activation FIFO register and, eventually, the main activation buffer.

4) MAC Lanes: MAC lanes are responsible for multiplication between two tiles in a parallelized fashion. Let the tiles be denoted by $\mathbf{W} \in \mathbb{R}^{b \times x \times y}$ and $\mathbf{A} \in \mathbb{R}^{b \times y \times z}$ for conserved matrix (in general, tensor) multiplication. Then, the number of multiplication operations is $n_o = b \times x \times y \times z$. Each MAC lane in AccelTran has M multipliers. Thus, the minimum number of cycles to compute the tiled operation is n_o/M. Fig. 6 shows the implementation of an MAC lane. We store all activation and weight data in fixed-point format with (IL + FL) bits, denoting integer length and fractional length, respectively [12]. The module first feeds the data to the M multipliers, then the corresponding outputs to the adder tree over multiple stages. We represent the products with $2 \times (\text{IL} + \text{FL})$ bits to prevent overflow. The accumulations also use this bit-width. The depth of the adder tree is $\log_2 M$ for the M multipliers in our MAC lane. The module then passes the data to the output register. For feedforward operations, where activation is required, the GeLU module implements this nonlinearity at the output of the MAC units. All other compute modules also work with the (IL + FL) bits.

5) Dynamic Inference Modules: To execute DynaTran pruning, we implement a low-overhead DynaTran module that

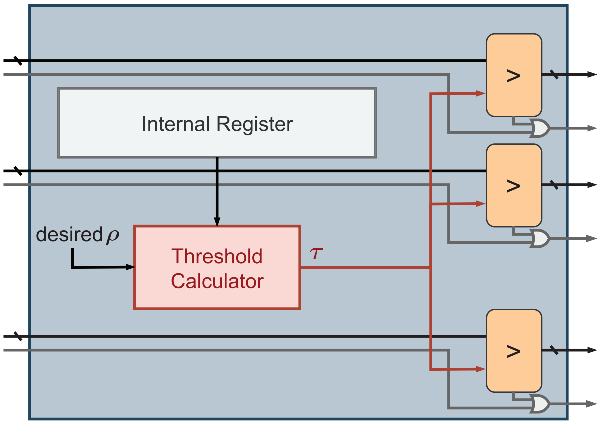


Fig. 7. DynaTran module. The wires for mask bits are in gray.

prunes ineffectual values in the input activations or weights. As explained in Section III-A, we prune the values of the input matrices by comparing their magnitude with a predetermined threshold τ. Fig. 7 shows how this is implemented, in parallel, for the entire tile. For an input tile $\mathbf{M} \in \mathbb{R}^{b \times x \times y}$, we use $b \times x \times y$ comparators. The threshold calculator determines the required threshold, using the desired ρ and the preprofiled transfer functions for different transformer models on diverse applications. The internal register stores these transfer functions loaded from memory before running transformer evaluation. If the output of the comparator is zero, we set the corresponding mask bit to one. Here, we represent the lines carrying mask information in **gray** and those carrying activation/weight information in **black**.

6) Sparsity-Aware Acceleration: To exploit sparsity and skip ineffectual activations and weights, and reduce memory footprint, AccelTran uses a binary-mask scheme to encode the sparse data and perform computations directly in the encoded format. Compared to the regular dense format, the precompute sparsity module compresses data by removing all the zero elements. In order to retain the shape of the uncompressed data, we use an extra binary mask [12]. The binary mask has the same shape as the uncompressed data, where each binary bit in the mask is associated with one element in the original data vector. If the entry in the mask is 1, it means that the corresponding activation/weight entry is ineffectual and should not be used for further computation.

Fig. 8 illustrates the precompute sparsity module. It takes the zero-free data and binary mask vectors as inputs and generates an output mask and zero-free activations/weights for the MAC lanes, SMX modules, or the layer-norm module. The output binary mask indicates the common indices of nonzero elements in both the activation and weight vectors. The module computes this mask using a bit-wise **AND** function over the input activation and weight masks. The two **XOR** gates then generate the filter masks. Based on the filter masks, the filter prunes the activations/weights. Finally, the zero-collapsing shifter compresses the activations/weights to feed zero-free data to the compute modules for further computation [12]. Thus, we completely skip ineffectual computations, improving throughput and energy efficiency.

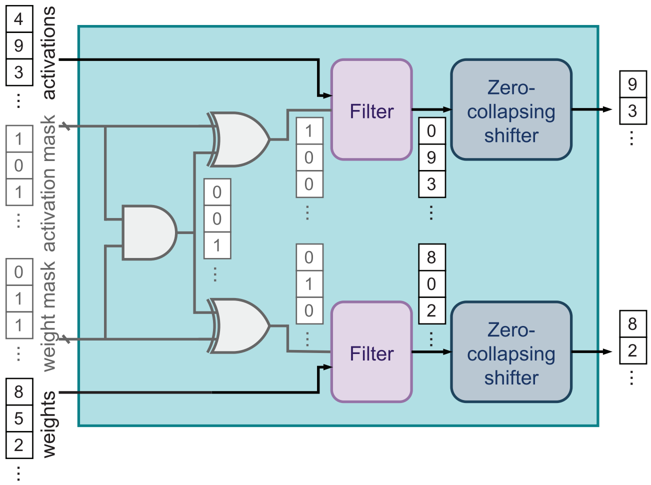


Fig. 8. Precompute sparsity module.

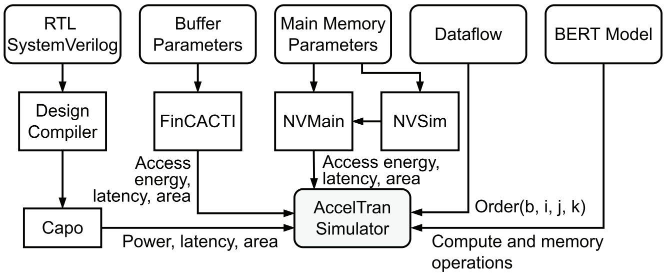


Fig. 9. Flow of simulation in AccelTran.

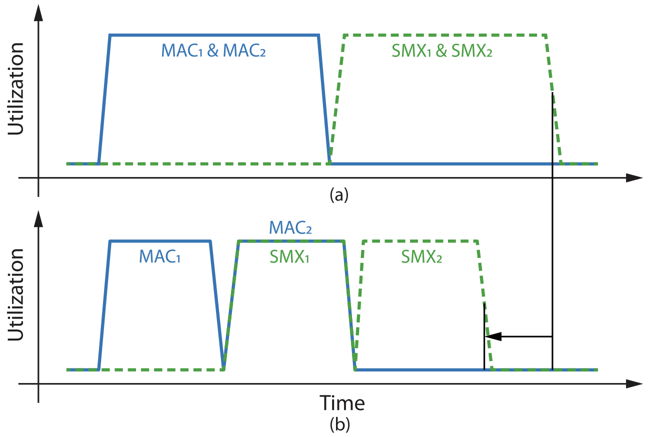


Fig. 10. Scheduling with (a) equal priority and (b) staggered operations for BERT-Tiny's MAC and SMX operations.

7) Simulator Flow: Fig. 9 shows the simulation flow for evaluating the AccelTran architecture. We implement different modules presented above at the register-transfer level (RTL) with SystemVerilog. Design Compiler [35] synthesizes the RTL design using a 14-nm FinFET technology library [36]. Capo [37], an open-source floorplacer, performs floorplanning. We did part of the floorplanning by hand. The net area reported is after floorplanning (including whitespaces). FinCACTI [38], a cache modeling tool for deeply scaled FinFETs, models the on-chip buffers. NVSim [39] and NVMain [40] model the main memory. We then plug the synthesized results into a Python-based cycle-accurate simulator.

8) Smart Scheduling of Tiled Operations: AccelTran simulates various operations in the transformer model in a tiled fashion. As discussed earlier, we tile each compute operation's activation/weight matrices. We then assign each such tiled operation to a designated module based on the type of compute operation. Modules that are not being used are power-gated to reduce leakage power draw. Transformer inference may run into either memory or compute stalls if the corresponding prerequisites are not met. As the names suggest, a memory stall halts a memory operation from being executed. Similarly, a compute stall halts a compute operation. There is a memory stall if the buffer is not ready to load/store more data as some data are already being written or read. Compute operations require some activations/weights in the buffers. There could be a compute stall if the required matrix is not yet loaded into the buffer. A memory stall can also occur if the compute modules are using current data in the buffer and there is no space left to add more data. This is true until the current data (that are required until compute operations finish) are evicted when the corresponding compute operations are done and the data are no longer required. A memory stall can also occur if the compute operation is not done before storing activation data. Finally, if all compute modules for a specific type of compute operation are busy, it could also lead to a compute stall.

The control block schedules various compute and memory operations to maximize hardware utilization. Since transformer models execute the same sequence of operations for every attention head, assigning equal priority to each head would result in poor usage of specialized resources. Hence, AccelTran staggers the operation of different heads. For instance, in BERT-Tiny, it gives more priority to one head so that the relevant MAC operations are completed first for that head. Then, when the first head reaches the SMX operation, MAC lanes can be assigned to the second head. This results in simultaneous utilization of the MAC lanes and SMX modules, thus, increasing hardware utilization and improving throughput. Fig. 10 presents a working schematic of the staggered implementation in BERT-Tiny's MAC and SMX operations (i.e., for two attention heads). In the staggered case, in Fig. 10(b), MAC lanes and SMX modules can be utilized simultaneously, resulting in a higher parallelization, thus, leading to a higher throughput.

IV. Experimental Setup

In this section, we present the setup behind various experiments we performed, along with the baselines considered for comparison.

A. Evaluation Models and Datasets

To test the efficacy of our proposed dynamic inference method, DynaTran, we evaluate encoder-only models (because of their high parallelization capabilities [17]) on different tasks. We use BERT-Tiny [26] and BERT-Base [1], two commonly used pretrained models. BERT-Tiny has two encoder

TABLE II
DESIGN CHOICES FOR ACCELTRAN-EDGE AND ACCELTRAN-SERVER

Accelerator	Module	Configuration
AccelTran-Edge	Main Memory	1-Channel LP-DDR3-1600; Bandwidth = 25.6GB/s
	PEs	64
	MAC Lanes	16 per PE
	Softmax Modules	4 per PE
	Batch Size	4
	Buffer	Activation Buffer: 4MB; Weight Buffer: 8MB; Mask Buffer: 1MB
AccelTran-Server	Main Memory	2-channel Mono. 3D RRAM; Bandwidth = 256GB/s
	PEs	512
	MAC Lanes	32 per PE
	Softmax Modules	32 per PE
	Batch Size	32
	Buffer	Activation Buffer: 32MB; Weight Buffer: 64MB; Mask Buffer: 8MB

layers, each with a hidden dimension $h = 128$ and two attention heads in the multihead attention operation, as discussed in Section II-A. BERT-Base is a larger model with 12 encoder layers, each with a hidden dimension $h = 768$ and 12 attention heads. These encoder-only models can also be extended to machine translation [18] and language generation [19]. Testing these recent extensions on hardware forms part of future work.

We test the two models on two *representative* tasks, namely, SST-2 [41] and SQuAD-v2 [42]. SST-2 is a popular benchmarking dataset that enables testing of model performance on sentiment analysis tasks. The dataset has 67K sequences in the training set and 872 in the validation set. The performance metric is the accuracy of correctly predicting label sentiment (positive or negative). SQuAD-v2 is a popular question-answering dataset. The training and validation sets have 130K and 12K examples, respectively. The performance metric is the F1-score [43].

While running DynaTran, we targeted both activation and weight sparsity. Weight sparsity is static and depends on pruning performed during model pretraining or fine-tuning (or even DynaTran's WP, as described in Section V-A2). Activation sparsity changes for every input sequence and is reported as the average over the entire validation set.

B. AccelTran Architectures

We now present various design choices for our proposed framework. We introduce two accelerators, namely, AccelTran-Edge and AccelTran-Server. The first is for mobile/edge platforms with a limited energy budget. The second is aimed at cloud/server applications where throughput may be of utmost importance. Table II shows the associated design choices. We fixed the clock rate to 700 MHz based on the delay of all modules in the proposed architecture. We set the number of multipliers M to 16. We set IL = 4 and FL = 16. As mentioned in Section V-B, the dataflow **`[b,i,j,k]`** is the loop-unrolling scheme of choice. We set the tile sizes across **`b`**, **`i`**, and **`j`** to 1, 16, and 16, respectively. For the chosen RRAM process [44] in AccelTran-Server, we implement the memory in two tiers above the main accelerator tier in order to fit it within the footprint area. However, different transformer models would generally have a unique set of hardware hyperparameters that are optimal for the given architecture. Thus, one can search for an optimal transformer-accelerator pair over a diverse set of transformer models [45] and accelerator design choices [46].

C. Evaluation Baselines

We compare the performance of our proposed accelerator with many previously proposed baselines. For mobile platforms, we compare the inference of BERT-Tiny on AccelTran-Edge with off-the-shelf platforms that include Raspberry Pi 4 Model-B [47] that has the Broadcom BCM2711 ARM SoC, Intel neural compute stick (NCS) v2 [48] with its neural processing unit (NPU), and Apple M1 ARM SoC [49] with an 8-core CPU, an 8-core GPU, and 16 GB unified memory on an iPad (for easier evaluations, we performed experiments on a MacBook Pro laptop with the same SoC instead). For server-side platforms, we compare the inference of BERT-Base on AccelTran-Server with a modern NVIDIA A100 GPU (40GB of video RAM) and previously proposed accelerators, namely, OPTIMUS [17], SpAtten [15], and Energon [16]. We chose the maximum batch size possible for each platform, based on its memory capacity.

To support inference on the Raspberry Pi, we implement the transformer models on an ARM distribution of the machine learning (ML) framework, PyTorch. We run transformer evaluation on the Intel NCS using the OpenVINO framework. Finally, for the Apple M1 SoC, we use the Tensorflow-metal plug-in to exploit the CPU and its embedded GPU. We quantize all models to FP16 before running our experiments. We normalize the throughput, energy, and chip area to 14-nm FinFET technology using scaling equations [50]. We use the inverter delays for different technology nodes as proxies for throughput normalization.

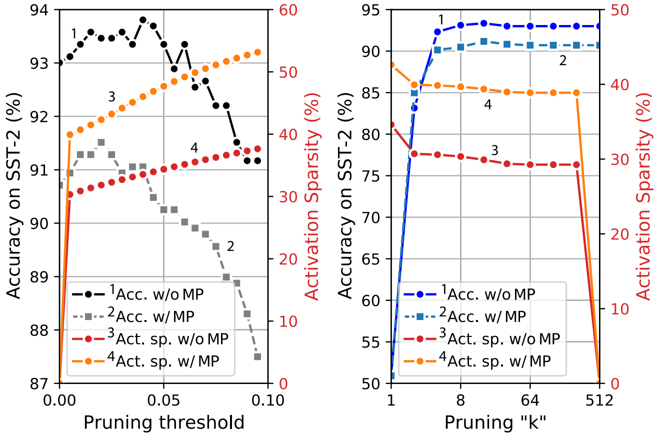
V. EXPERIMENTAL RESULTS

In this section, we present the experimental results.

A. Dynamic Inference With the Transformer

We first present the results of our experiments for the DynaTran method.

1) Comparing DynaTran With the Baseline: Figs. 11 and 12 present the profiled accuracy curves for BERT-Base on the SST-2 task for DynaTran and top-k pruning techniques. In Fig. 11, we show the effect of the pruning hyperparameters on sparsity. For DynaTran, the pruning threshold (τ) is varied from 0 to 0.1 and the activations are pruned based on the pruning threshold (see Section III-A). For top-k pruning, we change k in powers of two in order to see the effect of *net* activation sparsity, i.e., the sparsity in all activations rather than only the attention scores. Further, we also test prepruned models to see the impact on net activation sparsity when weights are also pruned. For this, we use the BERT-Base model pruned using the MP algorithm [30]. Using MP results in a higher activation sparsity (since the activations formed by matrix multiplications with weights are sparser when the weights are also sparse), but at the cost of



(a) (b)

Fig. 11. Accuracy on the SST-2 task and activation sparsity with (a) pruning threshold for DynaTran and (b) pruning "k" for top-k pruning.

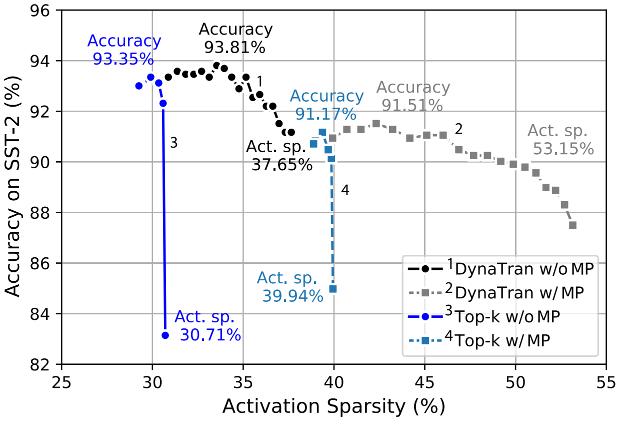


Fig. 12. Accuracy on the SST-2 task with activation sparsity for DynaTran and top-k methods. The annotations correspond to the maximum achieved accuracy or activation sparsity for each case.

lower accuracy. As also observed in previous works [16], both DynaTran and top-k methods see an initial increase in accuracy before a drop, as the sparsity increases. This could be attributed to the over-parameterization of the BERT model [51] and the corresponding pruning method acting as a regularizer, thus, giving a slightly higher validation performance.

We see similar results for other models and datasets. We store geometric mean curves, like the ones presented here, in the internal register of the DynaTran module with a low memory footprint. For the required activation sparsity, or even accuracy, we obtain the corresponding pruning threshold through the threshold calculator in the DynaTran module (explained in Section III-B5) to implement the desired dynamic inference.

Fig. 12 plots accuracy curves against activation sparsity for the DynaTran and top-k methods with and without MP. We obtain these curves from those in Fig. 11 by plotting accuracy against the corresponding resultant activation sparsity for every pruning threshold (τ) or the pruning k, as per the chosen method. We can see the trend of a slight increase in accuracy, here, as well. DynaTran achieves a higher accuracy (0.46% higher for BERT-Base without MP and 0.34% higher with MP) and a higher possible activation sparsity without much accuracy loss for both cases, i.e., with and without MP. For the same accuracy (the highest achievable by top-k), DynaTran enables 1.17× and 1.20× higher activation sparsity for each case, respectively. On the other hand, DynaTran can achieve up to 1.33× (1.23×) higher sparsity in absolute terms without MP (with MP). Here, we use $\tau < 0.1$, which yields reasonable accuracy values.

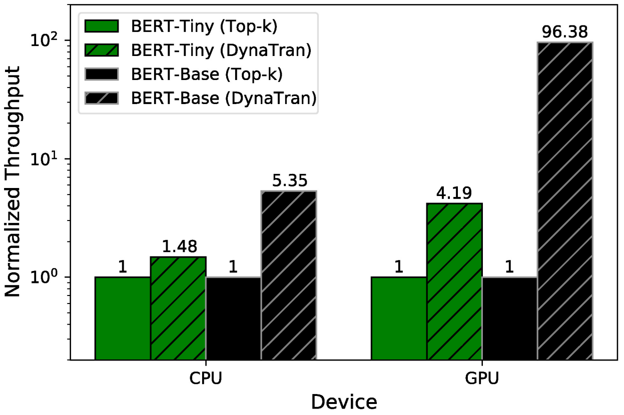


Fig. 13. Normalized throughput of DynaTran compared with the top-k method on a CPU and a GPU. Annotations are presented over each bar.

We now compare the compute cost of the top-k method with that of DynaTran. Fig. 13 shows the normalized throughputs of the two methods for BERT-Tiny and BERT-Mini on two devices. These are a 2.6-GHz AMD EPYC Rome CPU with 128 cores and 768-GB memory and an A100 GPU with 40-GB VRAM. DynaTran achieves up to 96.38× higher throughput on the GPU and up to 5.35× higher throughput on the CPU. This is due to the use of low-overhead comparators with a predetermined threshold. Even with the specialized top-k engine used in SpAtten and the approximation scheme used in Energon [16], they use more than one clock cycle, whereas DynaTran uses just one clock cycle. This is because the threshold calculator only needs a simple look-up operation and the comparators can execute within a clock cycle.

2) Testing If Weight Pruning Is Effective in DynaTran: DynaTran implements magnitude-based pruning of all activations at runtime. However, we can also leverage it to prune model weights before running the transformer. We call this WP since we only prune the transformer weights. In this approach, we do not need downstream training, as opposed to MP, which iteratively trains model weights while also pruning them. Fig. 14 presents the accuracies and F1-scores on the SST-2 and SQuAD datasets, respectively, with and without the use of WP. Net sparsity represents the combined sparsity of weights and activations. WP results in slightly higher net sparsity, however, with a significant loss in performance. The high ratio of activations compared to weights (see Fig. 1) results in only marginal gains in net sparsity. Hence, we do not employ WP in DynaTran. We use movement-pruned models instead, resulting in high weight and activation sparsities (with DynaTran) at negligible performance loss.

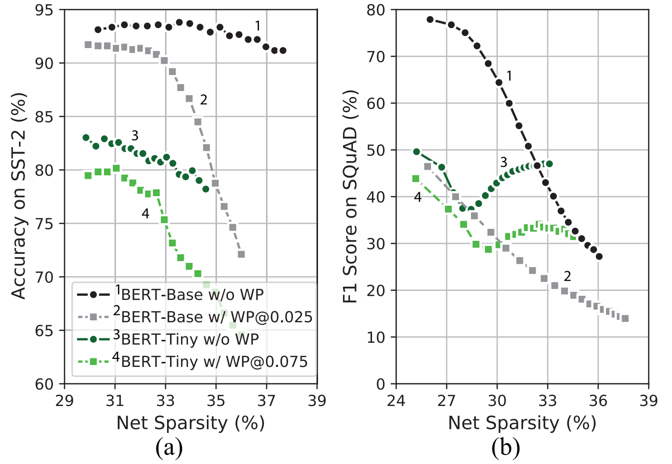


Fig. 14. Accuracy/F1-score plotted against net sparsity on the (a) SST-2 and (b) SQuAD benchmarks. In DynaTran, WP was implemented with a fixed threshold.

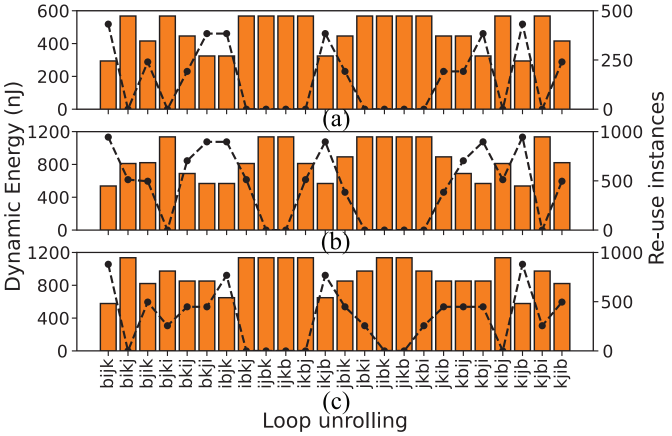


Fig. 15. Comparison of energy and reuse instances for all 24 dataflows under three matrix multiplication (**W** × **A**) scenarios: (a) $\mathbf{W} \in \mathbb{R}^{4\times64\times64}$, $\mathbf{A} \in \mathbb{R}^{4\times64\times64}$, (b) $\mathbf{W} \in \mathbb{R}^{4\times64\times64}$, $\mathbf{A} \in \mathbb{R}^{4\times64\times128}$, and (c) $\mathbf{W} \in \mathbb{R}^{4\times128\times64}$, $\mathbf{A} \in \mathbb{R}^{4\times64\times64}$. Bar plots represent dynamic energy and dashed lines represent reuse instances.

B. Dataflows and Data Reuse

We can pass on different tiles to available resources based on the four for-loops shown in Fig. 3. We can arrange these four for-loops in ${}^4P_4 = 24$ ways without changing the output. However, based on the compute resource constraints, different loop-unrolling strategies, or dataflows, can result in the reuse of local tiled weights or activations. Fig. 15 compares these dataflows for various matrix multiplication operations. The multiplication, **W** × **A**, is carried out using four MAC lanes in this simple example. We observe that dynamic energy is minimized by dataflows **`[b,i,j,k]`** and **`[k,i,j,b]`**. We use the former dataflow for subsequent experiments. These two dataflows also have maximum reuse instances for all three matrix multiplications. A reuse instance indicates if a weight or activation tile is reused in the internal register of an MAC lane. Many dataflows have the same energy and reuse instances due to symmetry. Since AccelTran hides data transfer overheads, due to the optimized control flow, the net latency is the same for all dataflows (this also results in the same leakage energy).

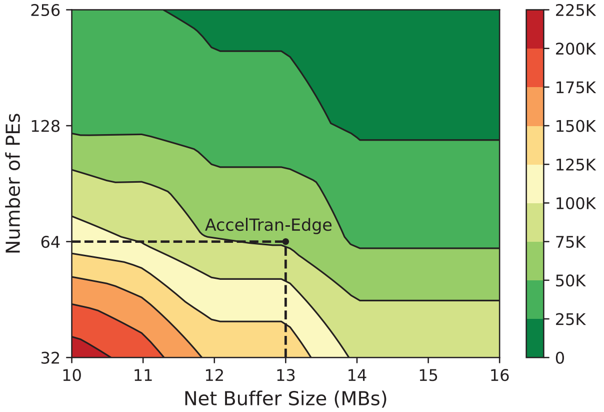


Fig. 16. Number of stalls with hardware resources.

Next, we test the effect of the different dataflows on real-world traces with the BERT-Tiny model on AccelTran-Edge. However, we observed negligible energy differences among the dataflows. This could be attributed to massive parallelization being at odds with data reuse. For instance, to reuse the same set of tiled weights in a PE's register, the next operation using those weights would have to be assigned to the same PE rather than exploit other free PEs, thus, limiting parallelization. Hence, as per Fig. 15, the advantages of data reuse can only be exploited in highly resource-constrained accelerators.

C. Design Space Exploration

Fig. 16 shows a plot of the number of compute and memory stalls when evaluating BERT-Tiny with different number of PEs and buffer sizes. We use a 4:8:1 size ratio for the activation, weight, and mask buffers. We found this ratio to be close to the optimal based on empirical studies on memory access patterns for the BERT-Tiny model. Next, we sweep the net buffer size from 10 MB to 16 MB. Finally, we choose the following number of PEs: 32, 64, 128, and 256. The figure shows that the number of compute stalls gradually increases as both the number of PEs and buffer size are reduced. We justify this as follows.

A lower number of PEs results in increased compute stalls since the compute operations have to wait for resources to free up in order to execute them, limiting available parallelization. In addition, a small buffer size results in memory stalls since memory store operations have to wait for the corresponding compute operations to finish before the current activations or weights, initially required by those compute operations, can be evicted from the buffer. Fig. 16 shows the chosen point for AcelTran-Edge. This set of design choices (64 PEs and 13 MB net buffer size) represents a reasonable tradeoff between the number of stalls (that directly increase latency) and hardware resources (that directly increase area and power consumption). An automatic hardware–software co-design approach [52] could also *efficiently* test different buffer sizes, along with the corresponding ratios that may be optimal

TABLE III
AREA, THEORETICAL PEAK TOP/S, AND MINIMUM MAIN MEMORY REQUIREMENTS, ALONG WITH POWER CONSUMPTION BREAKDOWN FOR DIFFERENT PARTS OF THE PROPOSED ACCELERATOR ARCHITECTURES. THE LP MODE FOR ACCELTRAN-EDGE IS ALSO CONSIDERED

Accelerator/Operation	Area (mm^2)	TOP/s	Main Mem. (MB)	Power Breakdown (W)			
				PEs	Buffers	Main Mem.	Total
AccelTran-Server	1950.95	372.74	3467.30	48.25	10.40	36.86	95.51
AccelTran-Edge	55.12	15.05	52.88	3.79	0.08	2.91	6.78
AccelTran-Edge (LP mode)	55.12	7.52	52.88	2.31	0.05	1.77	4.13

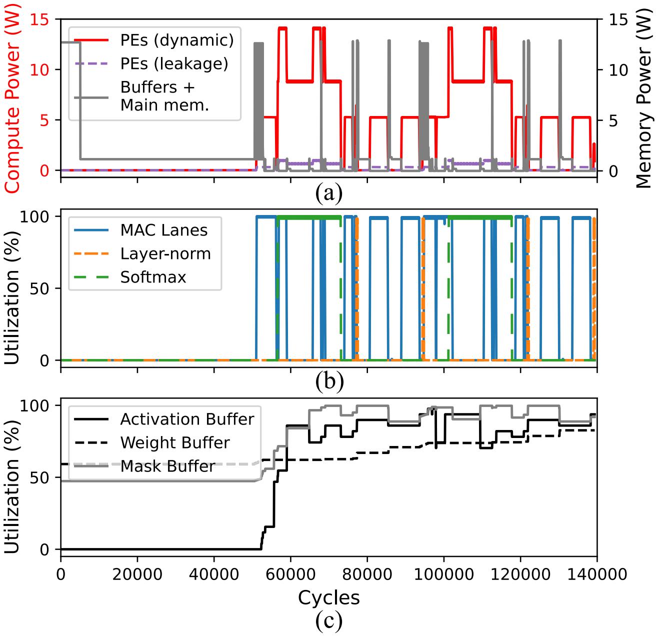


Fig. 17. Evaluation of BERT-Tiny on AccelTran-Edge: (a) power consumption, (b) resource utilization of compute modules, and (c) resource utilization of buffers.

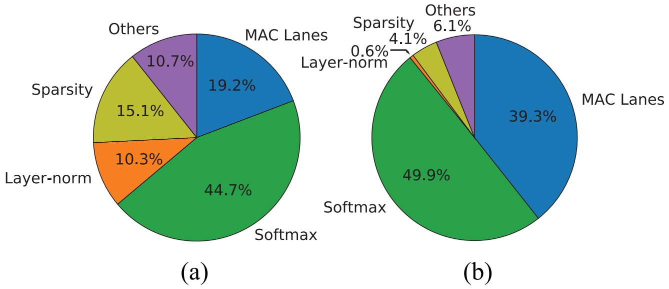


Fig. 18. Breakdown of (a) area and (b) power consumption by compute modules in AccelTran-Edge.

for each transformer model. We defer this automated co-design method to future work.

D. Hardware Performance and Utilization

Fig. 17 shows the power consumption and resource utilization of BERT-Tiny on AccelTran-Edge during inference of one batch. Hardware utilization remains at zero until around 51K cycles [see Fig. 17(b)] when the accelerator loads the word and position embeddings into the weight buffer (accounting for around 60% of the weight buffer). However, these load operations only occur once and subsequent transformer evaluations on different sequences reuse these embeddings. The rest of the process sees high utilization of MAC lanes or SMX modules. At certain times, the accelerator uses both MAC lanes and SMX modules due to the staggered implementation of attention head operations. The leakage power is low, as we show in Fig. 17(a), due to the power-gating of unused modules. Buffer usage drops suddenly, in Fig. 17(c), at certain instances when data are evicted in order to make space for new data for the active compute operations.

Table III shows the hardware performance measures for the proposed accelerator architectures, namely, AccelTran-Server and AccelTran-Edge, along with a low-power (LP) mode that we support for AccelTran-Edge. The LP mode only works with half of the compute hardware at any given time, resulting in lower net power draw, which is often a constraint in edge devices that rely on a battery source. We show the chip area first. AccelTran-Server is a massive chip with an area of 1950.95 mm^2, although still lower than that of the A100 GPU (3304 mm^2 normalized to a 14-nm process [50]). This can reduce the yield. However, we can leverage intelligent placement of PEs and binning to improve apparent yield rates [53]. We also show the tera-operations per second (TOP/s) performance measure for both architectures. AccelTran-Server can theoretically achieve a peak performance of 372.74 TOP/s, assuming all compute modules are operational simultaneously. We also present the minimum main memory size required for each accelerator. The net size of the embeddings and weights for BERT-Base and BERT-Tiny are 3467.30 MB and 52.88 MB (assuming a conservative 50% weight sparsity ratio [30]), respectively. However, transformer evaluation does not require all weights at any given time. Thus, the weight buffer can be much smaller. Similarly, even though the net size of activations is much higher (see Fig. 1), we can use a much smaller activation buffer. Finally, we present the power breakdowns for both the accelerators and the LP mode for AccelTran-Edge. The LP mode reduces power consumption by 39.1%, while lowering throughput by 38.7%, for BERT-Tiny.

Fig. 18 shows the area and power breakdowns for different compute modules in AccelTran-Edge. The 1024 MAC lanes only take up 19.2% of the area, while the specialized 256 SMX and 64 layer-norm modules take up 44.7% and 10.3% of the area, respectively. Pre- and post-compute sparsity modules comprise 15.1% area, while the dataflow, the DynaTran modules, and the DMA occupy 10.7% of the chip area. Fig. 18(b) shows the average power breakdown. Since most operations in the transformer involve matrix multiplication or SMX, they

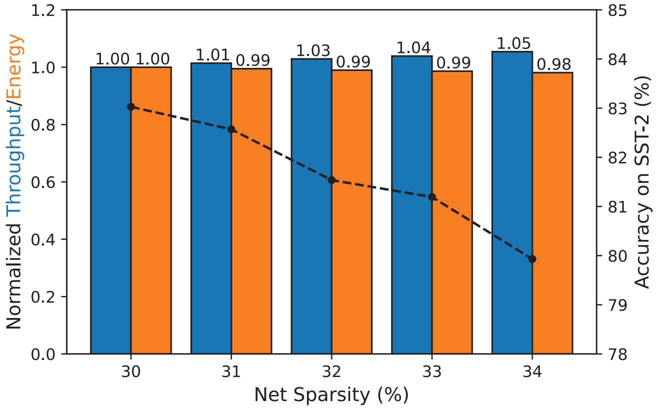


Fig. 19. Effect of sparsity on throughput and energy consumption. BERT-Tiny is simulated on AccelTran-Edge. Normalized throughput and energy are shown as bar plots on the left, and accuracy is shown as a dashed line plot on the right.

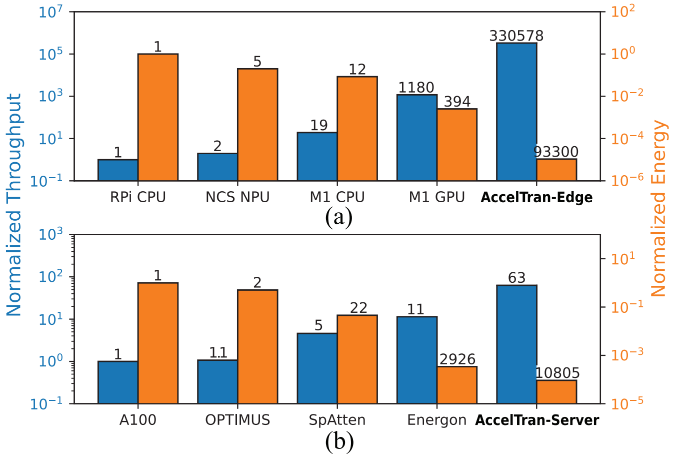


Fig. 20. Normalized throughput (left) and energy (right) comparison for AccelTran with baseline platforms targeted at (a) edge and (b) server applications.

also draw most of the power (39.3% for MAC lanes and 49.9% for SMX modules). The high power consumption of the SMX modules can be attributed to the calculation of the exponential sum over the entire tile in a parallel manner.

E. Effect of Sparsity on Throughput and Energy

Fig. 19 shows the effect of increasing sparsity on accelerator throughput and energy consumption. As the net sparsity increases from 30% to 34% for the BERT-Tiny model (with a conservative 50% weight sparsity estimate and accordingly tuned DynaTran's thresholds), throughput improves by 5% whereas energy consumption drops by 2%, when implemented on AccelTran-Edge. Here, accuracy drops by only 3% due to the low performance loss of DynaTran.

F. Performance Improvements

Fig. 20 shows performance comparisons of AccelTran architectures with baseline platforms. For edge applications, we compare the inference of BERT-Tiny on AccelTran-Edge with that on Raspberry Pi CPU, Intel NCS NPU, M1 CPU, and M1 GPU. AccelTran-Edge achieves 330 578× higher throughput at 93 300× lower energy consumption relative to Raspberry Pi. On the server side, we compare the performance of BERT-Base on AccelTran-Server with that of A100 GPU and some recently proposed accelerators, namely, OPTIMUS [17], SpAtten [15], and Energon-Server [16]. The throughput and energy values for SpAtten and Energon are normalized with respect to the A100 GPU. AccelTran-Server achieves 63× (5.73×) higher throughput at 10 805× (3.69×) lower energy consumption when compared to off-the-shelf A100 GPU (state-of-the-art Energon co-processor). These gains can be attributed to the execution of the DynaTran algorithm at runtime along with sparsity-aware modules that skip ineffectual computations. The specialized SMX and layer-norm modules also speed up the respective operations, otherwise implemented as matrix multiplications in the A100. Further, monolithic-3D RRAM has much lower data-retrieval latency than HBM in the A100. These contributions enable AccelTran to achieve high throughput gains over the A100 GPU. We study the effects of these contributions next.

TABLE IV
ABLATION ANALYSIS FOR INFERENCE OF BERT-TINY ON ACCELTRAN-SERVER

Accelerator Configuration	Throughput (seq/s)	Energy (mJ/seq)	Net Power (W)
AccelTran-Server	**172,180**	**0.1396**	24.04
w/o DynaTran	93,333	0.1503	**14.03**
w/o MP	163,484	0.2009	32.85
w/o Sparsity-aware modules	90,410	0.2701	24.43
w/o Monolithic-3D RRAM	88,736	0.1737	15.42

G. Ablation Analysis

Table IV presents an ablation analysis for the inference of BERT-Tiny on AccelTran-Server. The first row corresponds to the selected AccelTran configuration as per Table II, with 50% weight sparsity implemented through MP and 50% activation sparsity at runtime through DynaTran. The second row corresponds to the case not leveraging DynaTran. Then, we test the accelerator when the BERT model is not weight-pruned using MP. Third, we test it without employing the pre- and post-sparsity modules to skip ineffectual MAC operations. Finally, we present results when AccelTran-Server utilizes an off-chip LP-DDR3 DRAM instead of a high bandwidth monolithic-3D RRAM. Although the use of DRAM leads to a lower net average power consumption than when monolithic-3D RRAM is used, its total energy is higher due to a much lower throughput.

VI. DISCUSSION

In this section, we discuss the implications of the proposed accelerator in the field of ML acceleration and future work directions.

A. Dynamic Inference With Transformers

Previous works leverage complex pruning mechanisms, like top-k pruning, MP, etc. Implementing such pruning steps at

TABLE V
COMPARISON OF OUR PROPOSED ACCELTRAN FRAMEWORK WITH RELATED WORKS ALONG DIFFERENT DIMENSIONS. *ENERGON IS NOT AN ACCELERATOR BUT A CO-PROCESSOR

Work	**Transformer Acceleration**	**ASIC-based Acceleration**	**Monolithic 3D-RRAM**	**Tiled Mat. Mult.**	**Dataflow Support**	**Sparsity-aware**	**Dynamic Inference**
SPRING [12]		✓	✓			✓	
FTRANS [20]	✓						
FPGA Transformer [21]	✓						
A^3 [14]	✓	✓					✓
iMTransformer [54]	✓	✓				✓	
OPTIMUS [17]	✓	✓				✓	
SpAtten [15]	✓	✓				✓	✓
Energon* [16]	✓					✓	✓
AccelTran (Ours)	✓	✓	✓	✓	✓	✓	✓

runtime significantly slows down transformer evaluation. This has been a bottleneck in the widespread adoption of transformers on mobile platforms. In this work, we proposed a lightweight but powerful pruning mechanism: DynaTran. In essence, DynaTran implements magnitude-based pruning. However, we propose many novelties beyond vanilla magnitude-based pruning in terms of the algorithm and specialized hardware in order to obtain high gains relative to previous works. First, unlike previous works [55], [56], we prune not only the weights but also all the activations, which are formed at runtime. Second, we store preprofiled curves in the internal register of the DynaTran module. The threshold calculator selects the threshold for pruning at runtime based on user-defined constraints on accuracy or throughput. This enables dynamic adjustment of the desired accuracy or throughput at runtime (see tradeoff shown in Fig. 19). Third, the specialized DynaTran hardware module implements the algorithm in a single clock cycle, enabling high gains in throughput and reducing the bottlenecking effects of model pruning. Finally, DynaTran can easily incorporate any preprocessed WP strategy [55], [56] into its pipeline. In our work, we show how we leverage movement-pruned models to enable higher sparsity in weights and activations. DynaTran results in better accuracy than the top-*k* hardware-aware pruning mechanism and significantly improves throughput.

B. ML Accelerators

Various proposed ML accelerators target specific architectures. CNN accelerators [11], [12], [57], [58] focus on the convolution operation. Some works exploit sparsity in CNN models to reduce computation and memory footprint [12], [59], [60]. Certain works also exploit dynamism in model representation to minimize performance loss while leveraging low-bit computation. Two recent works, DUET [61] and Energon [16], employ dynamic mixed-precision computation. On the other hand, SPRING [12] implements stochastic rounding [62] with a fixed-precision format to maintain accuracy during training of CNNs. These extensions are orthogonal to the AccelTran framework and can easily be added to boost performance further. Table V compares the AccelTran framework with popular transformer accelerators.

We take motivation from SPRING and reuse some hardware modules with minor changes, like the MAC lane (we add the GeLU activation), the presparsity module, and the post-sparsity module. However, we design many new modules, namely, specialized RTL modules for the SMX and layer-norm operations, a module to carry out the DynaTran operations in a single clock cycle, and a novel control block that maps the transformer computational graph to hardware-implementable tiled operations. The control block is also responsible for choosing among various dataflows, originally not supported in SPRING. Unlike SPRING, it implements smart scheduling of operations to enable higher throughput in transformer evaluations (see Section III-B8). This is, especially, relevant to transformers with homogeneous operations throughout the model depth. Finally, AccelTran implements a lightweight dynamic inference algorithm for transformers, which SPRING does not support.

One could evaluate vision transformers (ViTs) [4] in AccelTran. However, this would require specialized hardware modules and data-processing pipelines to support image-to-sequence conversion in order to run ViT inference. AccelTran only supports model inference and specialized modules are required to accelerate the backpropagation process in transformer training. We leave these extensions to future work.

C. Hardware-Software Co-Design

In addition to leveraging sparsity in transformers, as explained in Section II-B, many more techniques have been proposed to obtain efficient transformers for pragmatic hardware implementation. These include low-bit quantization, knowledge distillation [26], approximation of the self-attention operation [31], [63], and WP [29], [30], [64]. Further, researchers have proposed hardware-aware neural-architecture search to guide the exploration of efficient transformer architectures with hardware feedback [25]. However, these works are only limited to certain embedded devices [25], FPGAs [20], [21], [22], or off-the-shelf microcontrollers [65] that are far from being optimized for large and compute-heavy transformer models. Leveraging the various design decisions in the AccelTran framework can enable efficient and fast co-design of the transformer architecture and hardware accelerator. This could incorporate user-defined constraints on model accuracy and target power envelopes in diverse deployments [45]. We leave this to future work.

VII. Conclusion

In this work, we presented AccelTran, a cycle-accurate accelerator simulator that efficiently runs dynamic inference with a given transformer model. We proposed a novel, low-overhead dynamic inference scheme, DynaTran, that increases the sparsity of activations at runtime with controllable accuracy loss. DynaTran achieves higher accuracy than the state-of-the-art top-*k* hardware-aware pruning strategy while enabling up to 1.33× higher sparsity. We further implement this method on two accelerator architectures: AccelTran-Edge and AccelTran-Server, specialized for mobile and cloud platforms, respectively. AccelTran-Edge achieves 330K× higher throughput at 93K× lower energy when compared to a Raspberry Pi device. Finally, AccelTran-Server achieves 5.73× higher throughput and 3.69× lower energy consumption relative to the state-of-the-art transformer co-processor, Energon.

Acknowledgment

The simulations presented in this article were performed on computational resources managed and supported by Princeton Research Computing at Princeton University.

References

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguist. Human Lang. Technol.*, vol. 1, 2019, pp. 4171–4186.

[2] A. Vaswani et al., "Attention is all you need," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 5998–6008.

[3] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, "Efficient transformers: A survey," *ACM Comput. Surveys*, vol. 55, no. 6, pp. 1–28, 2022.

[4] A. Dosovitskiy et al., "An image is worth 16x16 words: Transformers for image recognition at scale," in *Proc. Int. Conf. Learn. Represent.*, 2021, pp. 1–22.

[5] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[6] D. E. Rumelhart and J. L. McClelland, *Learning Internal Representations by Error Propagation*. Cambridge, MA, USA: MIT Press, 1987, pp. 318–362.

[7] L. Zhuang, L. Wayne, S. Ya, and Z. Jun, "A robustly optimized BERT pre-training approach with post-training," in *Proc. Chin. Nat. Conf. Comput. Linguist.*, 2021, pp. 1218–1227.

[8] S. Smith et al., "Using DeepSpeed and Megatron to train Megatron-turing NLG 530B, a large-scale generative language model," 2022, *arXiv:2201.11990*.

[9] D. So, Q. Le, and C. Liang, "The evolved transformer," in *Proc. Int. Conf. Mach. Learn.*, vol. 97, 2019, pp. 5877–5886.

[10] T. Brown et al., "Language models are few-shot learners," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 1877–1901.

[11] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[12] Y. Yu and N. K. Jha, "SPRING: A sparsity-aware reduced-precision monolithic 3D CNN accelerator architecture for training and inference," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 1, pp. 237–249, Jan.–Mar. 2022.

[13] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, *Efficient Processing of Deep Neural Networks*. San Rafael, CA, USA: Morgan Claypool, 2020.

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

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

[16] Z. Zhou, J. Liu, Z. Gu, and G. Sun, "Energon: Towards efficient acceleration of transformers using dynamic sparse attention," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 1, pp. 136–149, Jan. 2022.

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

[18] J. Zhu et al., "Incorporating BERT into neural machine translation," in *Proc. Int. Conf. Learn. Represent.*, 2020, pp. 1–18.

[19] S. Rothe, S. Narayan, and A. Severyn, "Leveraging pre-trained checkpoints for sequence generation tasks," *Trans. Assoc. Comput. Linguist.*, vol. 8, pp. 264–280, Jun. 2020.

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

[21] 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. Int. Syst.-Chip Conf.*, 2020, pp. 84–89.

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

[23] R. Prabhakar et al., "Plasticine: A reconfigurable architecture for parallel patterns," in *Proc. ACM/IEEE Int. Symp. Comput. Archit.*, 2017, pp. 389–402.

[24] Y. Yu and N. K. Jha, "Energy-efficient monolithic three-dimensional on-chip memory architectures," *IEEE Trans. Nanotechnol.*, vol. 17, no. 4, pp. 620–633, Jul. 2018.

[25] H. Wang et al., "HAT: Hardware-aware transformers for efficient natural language processing," in *Proc. Int. Conf. Assoc. Comput. Linguist.*, 2020, pp. 7675–7688.

[26] I. Turc, M. Chang, K. Lee, and K. Toutanova, "Well-read students learn better: The impact of student initialization on knowledge distillation," 2019, *arXiv:1908.08962*.

[27] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016, *arXiv:1607.06450*.

[28] D. Hendrycks and K. Gimpel, "Bridging nonlinearities and stochastic regularizers with Gaussian error linear units," in *Proc. Int. Conf. Learn. Represent.*, 2016, pp. 1–10.

[29] M. Gordon, K. Duh, and N. Andrews, "Compressing BERT: Studying the effects of weight pruning on transfer learning," in *Proc. Workshop Represent. Learn. Nat. Lang. Process.*, 2020, pp. 143–155.

[30] V. Sanh, T. Wolf, and A. Rush, "Movement pruning: Adaptive sparsity by fine-tuning," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 20378–20389.

[31] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," 2020, *arXiv:2006.04768*.

[32] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter," 2019, *arXiv:1910.01108*.

[33] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, and W. Liu, "TileSpGEMM: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 90–106.

[34] P. Batude et al., "3D sequential integration opportunities and technology optimization," in *Proc. Int. Interconnect Technol. Conf.*, 2014, pp. 373–376.

[35] "Synopsys design compiler." 2022. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[36] A. Guler and N. K. Jha, "Hybrid monolithic 3-D IC floorplanner," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 10, pp. 1868–1880, Oct. 2018.

[37] J. A. Roy et al., "Capo: Robust and scalable open-source min-cut floorplacer," in *Proc. Int. Symp. Phys. Design*, 2005, pp. 224–226.

[38] A. Shafaei, Y. Wang, X. Lin, and M. Pedram, "FinCACTI: Architectural analysis and modeling of caches with deeply-scaled FinFET devices," in *Proc. Comput. Soc. Annu. Symp. VLSI*, 2014, pp. 290–295.

[39] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 7, pp. 994–1007, Jul. 2012.

[40] M. Poremba, T. Zhang, and Y. Xie, "NVMain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems," *IEEE Comput. Archit. Lett.*, vol. 14, no. 2, pp. 140–143, Jul.–Dec. 2015.

[41] 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 Blackbox NLP Anal. Interpreting Neural Netw. NLP*, 2018, pp. 353–355.

[42] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ questions for machine comprehension of text," in *Proc. Int. Conf. Empir. Methods Nat. Lang. Process.*, 2016, pp. 2383–2392.

[43] D. Powers, "Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation," *J. Mach. Learn. Technol.*, vol. 2, no. 1, pp. 37–63, 2011.

[44] J. Yang et al., "24.2 A 14nm-FinFET 1Mb embedded 1T1R RRAM with a 0.022μm^2 cell size using self-adaptive delayed termination and multi-cell reference," in *Proc. Int. Solid-State Circuits Conf.*, vol. 64, 2021, pp. 336–338.

[45] S. Tuli, B. Dedhia, S. Tuli, and N. K. Jha, "FlexiBERT: Are current transformer architectures too homogeneous and rigid?" *J. Artif. Intell. Res.*, vol. 77, pp. 39–70, May 2023.

[46] Y. Lin, M. Yang, and S. Han, "NAAS: Neural accelerator architecture search," in *Proc. 58th ACM/IEEE Design Autom. Conf.*, 2021, pp. 1051–1056.

[47] "Raspberry Pi 4 model-B." Accessed: May 11 2023. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[48] "Intel neural compute stick 2." Accessed: May 11 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/neural-compute-stick/overview.html

[49] "Apple unleashes M1." Apple. 2020. [Online]. Available: https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/

[50] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm," *Integration*, vol. 58, pp. 74–81, Jun. 2017.

[51] Y. Hao, L. Dong, F. Wei, and K. Xu, "Visualizing and understanding the effectiveness of BERT," in *Proc. Int. Conf. Empir. Methods Nat. Lang. Process. Int. Joint Conf. Nat. Lang. Process.*, 2019, pp. 4143–4152.

[52] S. Tuli, C.-H. Li, R. Sharma, and N. K. Jha, "CODEBench: A neural architecture and hardware accelerator co-design framework," *ACM Trans. Embedded Comput. Syst.*, vol. 22, no. 3, p. 51, 2023.

[53] J. Sartori, A. Pant, R. Kumar, and P. Gupta, "Variation-aware speed binning of multi-core processors," in *Proc. Int. Symp. Qual. Electron. Design*, 2010, pp. 307–314.

[54] A. F. Laguna, M. M. Sharifi, A. Kazemi, X. Yin, M. Niemier, and X. S. Hu, "Hardware-software co-design of an in-memory transformer network accelerator," *Front. Electron.*, vol. 3, Apr. 2022, Art. no. 847069.

[55] W. Kwon, S. Kim, M. W. Mahoney, J. Hassoun, K. Keutzer, and A. Gholami, "A fast post-training pruning framework for transformers," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2022, pp. 1–16.

[56] M. Behnke and K. Heafield, "Losing heads in the lottery: Pruning transformer attention in neural machine translation," in *Proc. Int. Conf. Empir. Methods Nat. Lang. Process.*, 2020, pp. 2664–2674.

[57] Y. Chen et al., "DaDianNao: A machine-learning supercomputer," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 609–622.

[58] 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 Int. Symp. Comput. Archit.*, 2016, pp. 1–13.

[59] S. Zhang et al., "Cambricon-X: An accelerator for sparse neural networks," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–12.

[60] X. Zhou et al., "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2018, pp. 15–28.

[61] L. Liu et al., "DUET: Boosting deep neural network efficiency on dual-module architecture," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2020, pp. 738–750.

[62] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1737–1746.

[63] Y. J. Kim and H. Hassan, "FastFormers: Highly efficient transformer models for natural language understanding," in *Proc. SustaiNLP Workshop Simple Efficient Nat. Lang. Process.*, 2020, pp. 149–158.

[64] F. Lagunas, E. Charlaix, V. Sanh, and A. Rush, "Block pruning for faster transformers," in *Proc. Int. Conf. Empir. Methods Nat. Lang. Process.*, 2021, pp. 10619–10629.

[65] B. Lu, J. Yang, W. Jiang, Y. Shi, and S. Ren, "One proxy device is enough for hardware-aware neural architecture search," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 5, no. 3, pp. 1–34, 2021.

Shikhar Tuli (Student Member, IEEE) received the B.Tech. degree in electrical and electronics engineering from the Indian Institute of Technology Delhi, New Delhi, India, in 2020, with a department specialization in very large-scale integration and embedded systems. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, Princeton University, Princeton, NJ, USA.

His research interests include deep learning, edge artificial intelligence, hardware-software co-design, brain-inspired computing, and smart healthcare.

Niraj K. Jha (Fellow, IEEE) received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology (IIT) Kharagpur, Kharagpur, India, in 1981, and the Ph.D. degree in electrical engineering from the University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 1985.

He is a Professor of Electrical and Computer Engineering with Princeton University, Princeton, NJ, USA. He has coauthored five widely used books. He has published more than 470 papers (H-index: 83). His research interests include smart healthcare and machine learning algorithms/architectures.

Dr. Jha has received the Princeton Graduate Mentoring Award. His research has won 15 best paper awards, six award nominations, and 25 patents. He was given the Distinguished Alumnus Award by IIT Kharagpur, in 2014. He has served as the Editor-in-Chief for IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS and an associate editor for several IEEE TRANSACTIONS and other journals. He has given several keynote speeches in the areas of nanoelectronic design/test, smart healthcare, and cybersecurity. He is a Fellow of ACM.