Accelerating Sparse DNN Models without Hardware-Support via Tile-Wise Sparsity

Cong Guo*
Shanghai Jiao Tong University
guocong@sjtu.edu.cn

Bo Yang Hsueh
NVIDIA
bhsueh@nvidia.com

Jingwen Leng§
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute
leng-jw@sjtu.edu.cn

Yuxian Qiu
Shanghai Jiao Tong University
qiuyuxian@sjtu.edu.cn

Yue Guan
Shanghai Jiao Tong University
bonboru@sjtu.edu.cn

Zehuan Wang
NVIDIA
zehuanw@nvidia.com

Xiaoying Jia
NVIDIA
irishcoffee1006@gmail.com

Xipeng Li
NVIDIA
xipengl@nvidia.com

Minyi Guo§
Shanghai Jiao Tong University
Shanghai Qi Zhi Institute
guo-my@cs.sjtu.edu.cn

Yuhao Zhu
University of Rochester
yzhu@rochester.edu

***Abstract*—Network pruning can reduce the high computation cost of deep neural network (DNN) models. However, to maintain their accuracies, sparse models often carry randomly-distributed weights, leading to irregular computations. Consequently, sparse models cannot achieve meaningful speedup on commodity hardware (e.g., GPU) built for dense matrix computations. As such, prior works usually modify or design completely new sparsity-optimized architectures for exploiting sparsity. We propose an algorithm-software co-designed pruning method that achieves latency speedups on existing dense architectures. Our work builds upon the insight that the matrix multiplication generally breaks the large matrix into multiple smaller tiles for parallel execution. We propose a tiling-friendly "tile-wise" sparsity pattern, which maintains a regular pattern at the tile level for efficient execution but allows for irregular, arbitrary pruning at the global scale to maintain the high accuracy. We implement and evaluate the sparsity pattern on GPU tensor core, achieving a 1.95× speedup over the dense model.**

I. Introduction

Deep neural network (DNN) models have achieved and even surpassed human-level accuracy in important domains [51]. For instance, transformer-based models [56] in natural language processing (NLP) such as BERT [12] have dominated the accuracy in various NLP tasks and have been used in the Google's new search algorithm [15]. Despite their high accuracies, DNN models also have significant computational cost, both in training and inference. The large NLP model GPT-2 [46] has 1.5 billion parameters, and takes roughly a week to train on 32 TPUv3 chips and costs over $40,000 on Google's cloud TPU platform [54]. The inference latency of modern DNN models could also be excessively high due to the enormous computation cost and memory usage [37].

One particularly effective and promising approach to reduce the DNN latency is pruning [19], [28], which exploits the inherent redundancy in the DNN models to transform the original, dense model to a sparse model by iteratively removing "unimportant" weight elements and retraining the model to recover its accuracy loss. In the end, the sparse model has fewer parameters and, theoretically, less computation costs.

The primary challenge in network pruning is how to balance the model accuracy and execution efficiency. Such a balance is fundamentally affected by the *sparsity pattern* that a pruning approach enforces. Intuitively, a stronger constraint on the sparsity pattern forces certain weights to be pruned and, thus, leads to lower accuracy, and vice-versa. The most fine-grained pruning approach leads to the so-called element-wise (`EW`) sparsity pattern, which prunes weight elements individually and independently, solely by their importance scores [19]. In other words, `EW` imposes *no* constraints on the sparsity pattern and can remove any weight element, leading to the minimal model accuracy degradation. However, the pruned sparse model also introduces irregular memory accesses that are unfriendly on commodity architectures, e.g., GPUs [29], [50]. As a result, `EW`-based sparse DNN models usually runs slower than the unpruned dense models on these architectures [21].

To realize the acceleration potential of sparse DNN models, researchers have proposed to co-design the sparsity pattern with hardware support. For instance, many architects have proposed various specialized accelerator designs [18], [43] to exploit the zeros in the aforementioned `EW` pattern for latency reduction. Similarly, prior work proposes the vector-wise pattern [66] that divides a weight column to groups and prunes the same number of elements in a group. This sparsity pattern requires the new hardware or the modification of existing hardware [26], [70]. In summary, these approaches lead to sparse memory

*Contribution during his internship at NVIDIA.
§Jingwen Leng and Minyi Guo are corresponding authors of this paper.

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

accesses and computation patterns that require hardware support to be effective, and thus cannot leverage commodity DNN accelerators such TPU [24] and Volta tensor core [39], [47].

In this work, we propose a novel algorithm that is able accelerate sparse DNN models on commodity DNN accelerators without hardware modification. Our key observation is that virtually all of today's DNN accelerators implement dense general matrix multiplication (GEMM) [5] operations. GEMM-based accelerators [16], [24], [39], [44] are dominant owing to their wide applicability: convolution operations that dominate computer vision models are lowered to the GEMM operation, and NLP models are naturally equivalent to the GEMM operation. Examples include NVIDIA's tensor core [39] and Google's TPU [24] mentioned above. We propose a new pruning algorithm, which enforces a particular sparsity pattern on pruned models to directly leverage existing GEMM accelerators without modifying the microarchitectures.

In particular, our work exploits the key insight that the matrix multiplication on existing dense GEMM accelerators adopts the tiling approach, which breaks the large matrix into multiple smaller tiles for parallel execution. We propose a tiling-friendly sparsity pattern called *tile sparsity* (or TW), which maintains a regular sparsity pattern at the tile level for efficient execution but allows for irregular, arbitrary pruning at a global scale through non-uniform tile sizes to maintain high model accuracies.

To exploit the TW sparsity, we first divide the entire matrix into multiple tiles as in conventional tiled GEMM. We then prune the *entire rows or columns* of each tile according to the collective importance scores of each row and column. In our sparsity pattern, the tile size dictates the trade-off between model accuracy and execution efficiency. At one extreme where the tile size equals one, our TW sparsity is equivalent to the EW sparsity. At the other extreme where the tile size is the same as the matrix size, TW pruning is equivalent to the global structural pruning that prunes the entire row or column [21].

Building on top of TW, we further propose a *hybrid* sparsity pattern that overlays the most fine-grained EW sparsity pattern on top of the TW sparsity. With a small fraction of EW (e.g., 1.5%), the hybrid pattern greatly improves the accuracies of the TW-only sparse models. We propose a pruning algorithm that iteratively shapes the weight matrix to meet our hybrid sparsity pattern constraint. Critically, our pruning algorithm dynamically allocates the sparsity budget to each layer to exploit the inherently uneven sparsity distribution across layers.

To maximize the algorithmic benefits of TW, we provide an efficient software implementation on commodity GPU hardware. Two key roadblocks arise as a result of the TW sparsity. First, TW naturally introduces frequently uncoalesced memory accesses due to the pruning pattern. Second, different tiles in TW could have different compute demands due to the different pruning degrees across tiles, which leads to load imbalance and GPU resource under-utilization. We address these challenges through a combination of intelligent data layout and concurrency/batching optimizations. TW achieves an average of 1.95× (2.86×) latency speedup on the tensor core (CUDA core) with only negligible accuracy loss (1%-3%).

The contribution of our work is as follows:

- We propose a tiling-based sparsity pattern to balance the model accuracy and execution efficiency on the existing dense accelerator. The tile sparsity can be combined with existing fine-grained pattern to minimize the accuracy loss.
- We propose a multi-stage pruning algorithm that gradually shapes the weight matrix to our proposed pattern and dynamically allocates the sparsity budget at the layer level to overcome the uneven distribution of sparsity.
- We provide an efficient implementation of tile sparsity on commodity GPUs equipped with tenor core, and demonstrate significant speedups on state-of-the-art DNN models.

We organize the paper as follows. Sec. II describes the background and Sec. III provides the motivation. Sec. IV describes the overview of TW. We explain its pruning algorithm and tensor core implementation in Sec. V and Sec. VI, respectively. We evaluate TW against other patterns in Sec. VII, discuss the related work in Sec. VIII, and conclude in Sec. IX.

II. Background

This section provides the relevant background on the different deep neural networks that we evaluate in this paper. We then summarize the recent efforts for reducing the execution latency of those models, which include building specialized hardware accelerators and applying algorithmic pruning optimization to reduce the size and computation cost of DNN models.

A. Deep Neural Network Model

DNN models have recently achieved state-of-the-art results in many important domains, such as convolution neural network [27] (CNN) in the computer vision domain, and long short term memory [22] (LSTM, most popular RNN) and BERT [12] in the natural language processing domain. The CNN and LSTM are relatively well studied models whose details are shown in Fig. 1. We refer the readers to the prior literatures [3], [6] for more explanations of those models.

BERT is a representative Transformer [57]-based model and has outperformed LSTM in NLP domains. Fig. 1 shows the structure of a Transformer layer constructing the BERT model. The Transformer applies multi-head attention (MHA) mechanisms, which stands for several groups of independent attentions enabling them to deal with different aspects of information [10]. The BERT model is extremely large and computational expensive. With adjustable depth and width, there are two popular BERT versions: BERT-large with 24 layers and 16 heads, BERT-base with 12 layers and 12 heads. Without loss of generality, the explore of this work is built on BERT-base and we refer BERT-base as BERT in the following.

B. Hardware Acceleration

We explain the computation characteristics of these models and common optimizations to reduce their execution latency.

Dense Model. General matrix multiplication (GEMM) is a key computation in the original dense DNN models, as indicated

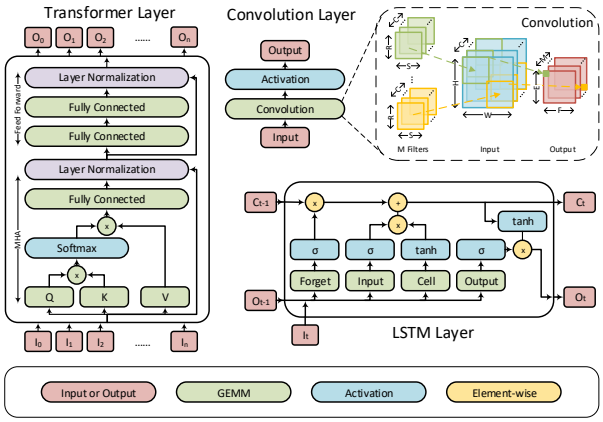


Fig. 1: The Transformer layer of BERT, convolution layer, and LSTM layer with different kinds of computations.

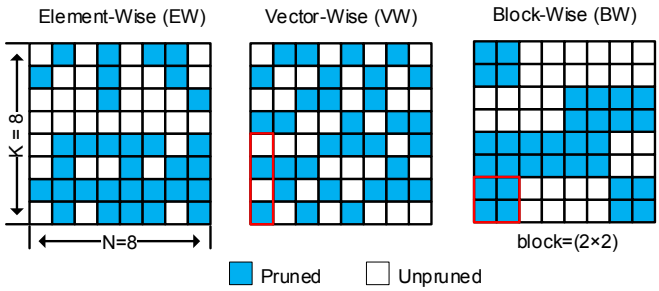


Fig. 2: Comparison of three patterns with 50% sparsity: element-wise (EW) pattern prunes individual elements, vector-wise (VW) pattern prunes half elements in a 4-element vector, and block-wise (BW) pattern prunes an entire 2×2 block.

by the light green blocks in Fig. 1. The fully connected layer and LSTM layer are native GEMM operations while the convolutional layer can be converted to GEMM through the `img2col` transformation. The attention heads in BERT can also be computed with GEMM operations and the computation of multiple heads could be combined to one large GEMM.

GEMM Accelerator. To reduce the model execution latency, NVIDIA adds tensor core on the GPU in Volta architecture, which runs a fixed size ($16 \times 16 \times 16$) matrix multiplication. tensor core is essentially an accelerator for the GEMM. Other examples for GEMM accelerator is TPU [24] which is based on a 128×128 systolic array. The cuDNN [7] library implements different DNN layers for efficient execution on GPU, where the GEMM computation can use the closed-sourced cuBLAS library [40] or open-sourced CUTLASS library [41].

Sparse Model. Recently, researchers start to apply pruning [19], [28] to DNN models, which exploits the inherent redundancy in the model to transform the original, dense model to a sparse model. In the end, the sparse model has fewer parameters and, theoretically, less computation costs. Executing sparse models relies on sparse matrix representation such as compressed sparse row (CSR) and sparse GEMM operations, which are supported on GPU by cuSparse [40] library. However, as the GPU is originally designed for dense operations, the speedup of sparse model over the dense model is usually negative unless the sparsity ratio is very large (over 95% reported by prior work [59]). As such, researchers begin to put various shape constraints on the pruning pattern and also propose to transform the existing architecture to execute those sparse models. For example, the recent work propose new sparsity patterns that need to modify the existing dense GEMM accelerator like tensor core [70] and TPU [26].

Different from the prior microarchitecture-centric work, we propose a software-only acceleration of sparse DNN models on the dense GEMM accelerator like tensor core. We exploit the tile execution of GEMM computation and propose a tiling-friendly, `Tile-Wise` sparsity pattern to balance the model accuracy and compatibility for the dense GEMM accelerator.

III. Pruning Efficiency Analysis

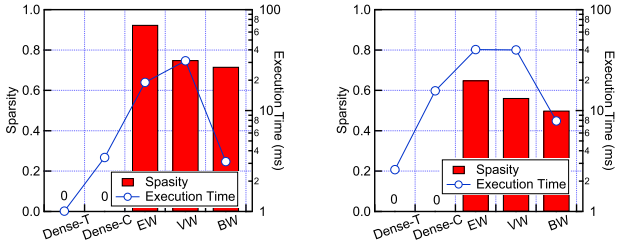
In this section, we study the impact of different DNN pruning algorithms on the model execution efficiency. Network pruning can remove the excessive weights in DNN models and therefore the resulted sparse model has less weight size and computation cost. However, our analysis shows that the sparse models generated by existing pruning algorithms cannot achieve meaningful speedup on the dense architectures.

A. Sparsity Pattern

There are two major components in the algorithms for pruning deep neural networks. The first component is how to evaluate the importance of individual weight element (i.e., importance score). And the second component is the sparsity pattern that is removed altogether. In this part, we mainly focus on the study of different sparsity patterns and we explain the details of importance score calculation in Sec. V.

Fig. 2 illustrates different sparsity patterns. The first sparsity pattern, called `element-wise (EW)`, removes the individual weight element solely by its importance score rank. For instance, prior work [19] proposes to remove weight elements with small magnitude. This approach imposes no constraints on the sparsity pattern and could remove most of weights among all pruning methods. Thus, it is also called unstructured pruning. However, the randomly distributed non-zero weights lead to substantial irregular memory accesses, which imposes great challenges for efficient hardware execution. As such, researchers propose other two more structured pruning methods.

The second sparsity pattern shown in the middle of Fig. 2, called `vector-wise (VW)` [66], [70], divides a column in the weight matrix to multiple vectors. Within each vector, it prunes a fixed portion of elements by the rank of their importance scores. This approach preserves the randomness within each vector for model accuracy. Meanwhile, it also maintains the regular structure for efficient execution, where



(a) VGG. (b) BERT.

Fig. 3: Sparsity and execution time comparison between dense and various sparse models (VGG and BERT). The accuracy of various sparse models is 1% lower than the dense model.

different vectors have the same number of non-zero weight elements. The third pattern, called `block-wise (BW)` [35], divides the weight matrix to small blocks, and treats a block as the basic pruning unit. In other words, the `EW` sparsity pattern is a special case of the `BW` sparsity pattern, which expands a 1×1 block to an $n \times n$ block. The structural sparsity pattern `EW` leads to the efficient execution of sparse models.

B. Execution Efficiency Analysis

We first use two popular deep neural network models to evaluate the execution efficiency of sparse models generated by the aforementioned three sparsity patterns. Our experimental results show that although these pruning approaches can lead to sparse models with a large volume of sparsity (i.e., zero weight elements), they fail to achieve a meaningful speedup compared to the unmodified dense model on the existing GPUs. Moreover, the emergence of dense matrix multiplication accelerator such as tensor core further widens their performance gap.

We first study a CNN model VGG-16 [53] with 13 convolutional layers and 3 fully connected layers. We evaluate it on the image classification task using the ImageNet dataset [25]. The second studied model is BERT (base) in Sec. II-A, which is a Transformer-based model with 12 encoding layers. We evaluate the BERT model with the sentence classification task on the MNLI dataset [58]. We use the vector size of 16 for `VW` and the block size of 32×32 for `BW` as suggested in their original papers [35], [70]. We prune both models with the three different sparsity patterns and keep the accuracy drop of each model within 1% of its unmodified dense version.

We perform the efficiency analysis on a V100 GPU [39] with CUDA 10.1 [40]. Besides the CUDA cores, the V100 GPU also integrates the specialized tensor core for the acceleration of dense matrix multiplication. This GPU has a peak throughput of 15.7 TFLOPS (floating point operation per second) and 125 TFLOPS, for the CUDA cores and tensor cores, respectively. We evaluate the performance of dense model with the cuDNN library on the CUDA core and tensor core separately. We execute the sparse model of `EW` and `VW` with the cuSparse [40] library, which executes only on the CUDA cores. We execute the sparse model of `BW` with the BlockSparse [55] library, which leverages the tensor cores owing to its regularity.

Fig. 3 compares the sparsity and performance between the dense and sparse models with various patterns. All patterns achieve over 50% sparsity with the greatest by `EW`. The performance of `EW` and `VW` is slower than their dense version on the CUDA core (Dense-C). In addition, the performance gap between the dense model and sparse model exacerbates when the dense matrix accelerator tensor core is used (Dense-T). Prior work [70] reports a $1.5\times$ speedup using the `VW` pattern, which requires non-negligible modifications of the tensor core. `BW` achieves the best performance among all sparse models as it runs on the tensor core. However, its performance is still $3\times$ slower than the dense model on tensor core.

In summary, the existing pruning approaches generate sparse models that are inefficient on the existing hardware. As such, we need a sparsity pattern that can match the existing hardware features while maintaining the fine granularity, which is critical for achieving the high model accuracy.

IV. Tile Sparsity

In this section, we present the details of our proposed tile sparsity pattern. Our approach leverages the tiled execution of matrix multiplication, which is originally designed for exploiting the parallel computation resources. The proposed tile sparsity pattern introduces irregularity at the global matrix level, but maintains the regularity of individual matrix tile. As such, it can balance the DNN model accuracy and compatibility for dense matrix accelerator, e.g., tensor core. We also show that the tile sparsity pattern can be overlaid with the most fine-grained element-wise pattern to increase the sparsity of pruned models and reduce their accuracy loss.

A. Tiling and Pruning Co-design

As Sec. II explains, the dominant computation in deep neural network models is the general matrix multiplication (GEMM). In this subsection, we first present the details of tiled matrix multiplication. We then propose to co-design the matrix tiling and deep neural network pruning, which leads to the `tile-wise (TW)` sparsity pattern. We explain how `TW` maintains the compatibility on the dense GEMM accelerator and the composability with the fine-grained sparsity pattern.

Fig. 4 ❶ shows one level tiling of matrix multiplication on the GPU. The GEMM computes $C = A \times B$ with input matrix A $(M \times K)$, weight matrix B $(K \times N)$, and output matrix C $(M \times N)$. Since modern high-performant microprocessors mostly adopt the manycore architecture, the tiled execution of output matrix C breaks the entire GEMM computation into multiple ones such that they can run on multiple cores for the parallel execution. Specifically, each core (or streaming multi-processor, SM in NVIDIA GPU) computes one tile with size of $T_y \times G$. Consequently, the core only loads T_y rows of input matrix A and G columns of weight matrix B (called B_{tile}).

With the output matrix tile size of $T_y \times G$, the $K \times N$ weight matrix B is divided to $\lceil \frac{N}{G} \rceil$ B_{tile}. The key idea of our `tile-wise` pattern is to prune each B_{tile} with the regular row pruning and column pruning. As shown in Fig. 4 ❷, the row pruning treats an entire row of each weight tile B_{tile} as the basic pruning unit, which leads to the reduced K-dimension size (i.e., height) of each B_{tile}. We prune each B_{tile} with different

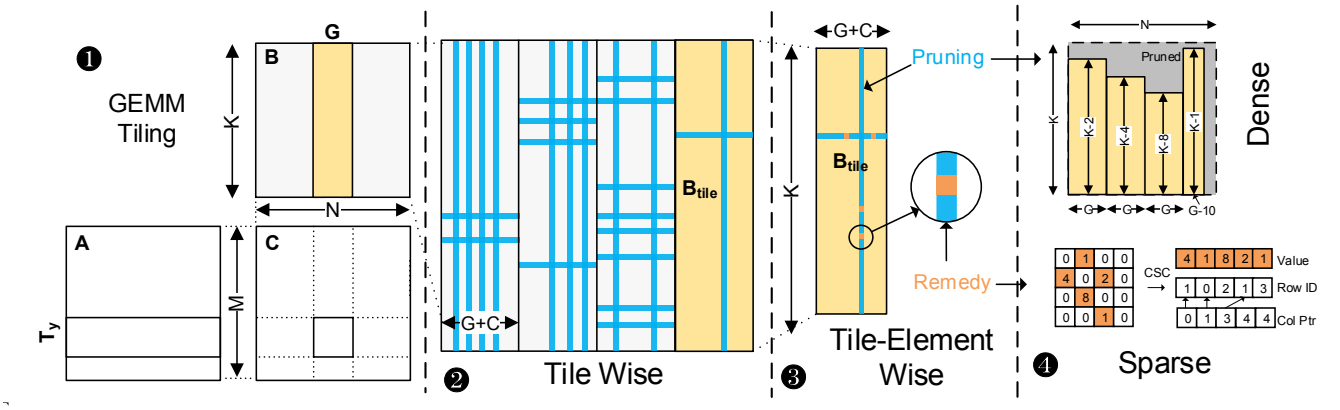


Fig. 4: The overview of `Tile-wise (TW)` sparsity pattern that exploits the tiled dense matrix multiplication (GEMM) to maintain the GEMM-compatible execution. (1) The output matrix tiling based GEMM execution. (2) Our `TW` sparsity pattern essentially performs the regular row and column pruning on each matrix tile. (3) The hybrid tile-element-wise (`TEW`) sparsity adds back the individual important elements in the pruned row/column to restore the accuracy of sparse `TW` models. (4) The execution of `TW` can be converted to multiple small dense GEMMs, and the execution of `TEW` can use extra sparse GEMM.

number of rows determined by the pruning algorithm that we describe later. The difference across different tiles maintains the irregularity of sparsity that is required by model accuracy. E.g., the heights of four weight matrix tiles in Fig. 4 ❷ are $K-2$, $K-4$, $K-8$, and $K-1$ respectively after the row pruning.

Besides the row pruning, we also perform the column pruning for the weight matrix tile B_{tile}, which reduces its N-dimension size. Our approach prunes different number of columns (C) in each weight matrix tile for better irregularity. The combined row and column pruning alleviate the constraint on the sparsity pattern and therefore allow more weight elements to be pruned. In specific, we perform column pruning before row pruning for maximizing the execution efficiency, which we explain later.

Pattern Overlay. Since the `TW` still enforces a particular pruning pattern, important weight elements could be removed, which may lead to accuracy loss. We propose to overlay `TW` and `EW` to mitigate the accuracy loss. Fig. 4 ❸ illustrates the resulted hybrid pattern `tile-element-wise (TEW)`. In order to prune α percent of weights, the `TEW` first prunes $\alpha+\delta$ percent of weights with only `TW`, and then **restores** δ percent of the weight elements with the highest importance scores.

Pruning Order. To improve the execution efficiency of the proposed sparsity pattern, we first perform the column pruning and then re-organize the weight matrix tiles for row pruning. We use the example in Fig. 4 ❷ to illustrate its advantages. With the column pruning, the four tiles are pruned with 4, 3, 2, and 1 columns, respectively. For the row pruning inside each tile, we re-organize the four tiles with $G+4$, $G+3$, $G+2$, and $G-9$ columns. After pruning (Fig. 4 ❹), the N-dimension sizes of the four tiles are G, G, G, and $G-10$, respectively. The first three tiles have the same number of columns such that their execution can be batched for better performance. For the hybrid `TEW` pattern, each tile stores the `EW` pattern with the compressed sparse column (CSC) format. We leverage linear property of matrix multiplication to execute the `TW` and `EW` separately. We explain the execution details in Sec. VI.

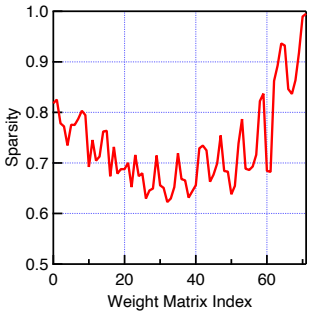


Fig. 5: The per-layer sparsity when pruning BERT using the `EW` pattern with 75% overall sparsity.

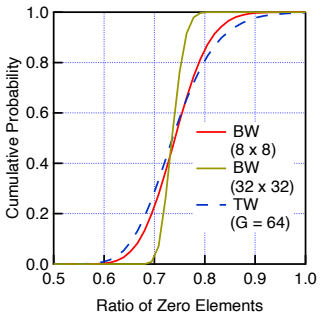


Fig. 6: Cumulative probability distribution of zero weight elements with `BW` and `TW` patterns.

B. Comparison with Other Sparsity Patterns

In this subsection, we demonstrate that the `TW` is not only friendly for the hardware execution, and also preserves the model accuracy. In specific, we compare the irregularity, which determines the pruned model accuracy under the same sparsity percentage, of `EW`, `TW`, `BW`, and `VW`. We use the `EW` as the baseline because it has the highest degree of irregularity and therefore the best accuracy.

Against `VW`. First, we find that there exists uneven distribution of sparsity in different weight matrices, which makes `TW` a more suitable pattern than `VW`. We illustrate this point by showing the sparsity distribution of 72 weight matrices in BERT, which has 12 layers and each layer has 6 weight matrices (4 for the self attention and 2 for FC layers). We apply the `EW` pattern to prune 75% weights. In specific, the importance score of all elements in the 72 weight matrices are calculated and globally ranked for the element-wise pruning. As Fig. 5 shows, the final pruned weight matrices have different degrees of sparsity

Algorithm 1: The multi-stage TW pruning algorithm.

Input: Pre-trained weight matrix, M_s, and shape, (K, N); Target sparsity, S; Variable granularity, G;
Output: Pruned weight matrix, M_d

1 $m = M_s$; $s_t = 0$;
2 **while** $s_t < S$ **do**
3 s_t=GraduallyIncrease(s_t);
4 $m = m$ Splited by shape(K, 1) for Column Pruning;
5 *tileScore* =ImportanceScore(m);
6 *tileScore* =AprioriTuning(*tileScore*, S);
7 *threshold* = Percentile(*tileScore*, s_t);
8 **while** *each* $tile_i \in m$ **do**
9 **if** $tileScore[i] < threshold$ **then**
10 Prune the Column $tile_i$ with shape(K, 1);
11 **end**
12 **end**
13 $m = m$ Splited by shape(1, G) for Row Pruning;
14 *tileScore* =ImportanceScore(m);
15 *threshold* = Percentile(*tileScore*, s_t);
16 **while** *each* $tile_i \in m$ **do**
17 **if** $tileScore[i] < threshold$ **then**
18 Prune the Row $tile_i$ with shape(1, G);
19 **end**
20 **end**
21 FineTune(m);
22 **end**
23 $M_d = m$; return M_d;

although the averaged sparsity for all weights is 75%. This suggests the uneven distribution of sparsity in the DNN model. VW splits every column to a certain number of groups and all the groups have the same sparsity by pruning the same number of elements. In contrast, TW can maintain the uneven sparsity distribution by globally ranking the matrix tiles.

Against BW. Compared to the BW [35], TW can remove more weights owing to its less constraints on the pruning shape. To illustrate this point, we characterize the number of zero elements in different pruning shapes on the BERT model with 75% EW sparsity. Fig. 6 compares the number of zero elements in a block of 8×8, 32×32 for the BW, as well as in a row vector of 64 elements for the TW with $G = 64$. Both with 64 elements, TW captures more zeros elements than BW. Meanwhile, prior work reports that BW requires a pruning unit of 32×32 for maintaining high performance [8], which captures even less zero elements as Fig. 6 shows. In contrast, TW with $G = 64$ is sufficient for achieving significant speedups as we show later.

In summary, we conclude with the irregularity relationship of EW > TW > VW $\approx$ BW. Owing to the existence of globally uneven sparsity distribution, TW leads to a pattern than is closer to EW than the VW pattern. TW also removes more weights than BW owing to its less constraints on the pruning shape.

V. Tile Sparsity Based Pruning

This section explains our multi-stage pruning algorithm for leveraging the proposed TW sparsity pattern. Algorithm 1 describes the algorithm, which we explain in details as follows.

Overview. We adopt the multi-stage pruning algorithm that gradually prunes the pre-trained dense model to reach a target sparsity. Each stage consists of a pruning and fine-tuning step,

Algorithm 2: Apriori tuning.

Input: EW pruned results, *EW*; Top-n and Last-n, n_1, n_2; Tile importance score, *tileScore*; Target sparsity, S;
Output: Tile importance score, *tileScore*;

1 $tileSparsity = EW[S]$;
2 *topNTiles* =GetTopNTiles(*tileScore*, $n1$);
3 *tileScore* =SetZero(*tileScore*, *topNTiles*)
4 *lastNTiles* =GetLastNTiles(*tileScore*, $n2$);
5 *tileScore* =SetInf(*tileScore*, *lastNTiles*); return *tileScore*;

where the algorithm first prunes the model with a small sparsity target and then fine-tunes the pruned model to restore the model accuracy. The pruning stage is repeated until the model reaches the target sparsity. Prior work points out that the multi-stage pruning improves the model accuracy than the single-stage pruning [19]. At each stage, the algorithm calculates the importance score of each tile. It then performs the column and row pruning according to the rank of importance score. We also perform the apriori tuning that borrows the information of EW pruning to reduce the accuracy loss. Each iteration from line 3 to 21 in Algorithm 1 is a complete pruning-tuning stage, where line 3 increments the target in the current stage.

Importance Score. How to compute the importance score is an active research topic [19], [30], [33], [34], [45], [67], [69]. The most intuitive approach [19] is to use the weight's absolute value. We use a more accurate approach [33] that uses the incurred error by removing a parameter as its importance score. Although not the focus of our work, we find that this approach leads to a better accuracy under the same sparsity on complex models like BERT, and can reduce the fine-tune time for other models like NMT and VGG in our experiments.

Let L be the loss function, w be the targeted weight variable, $w = w_i$ means the original value, and $w = 0$ means that we prune w. The importance score for w is the difference of loss function as Equ. (1) shows. However, the exact computation is expensive because M parameters require evaluating M versions of the network, one for each parameter. We avoid evaluating M different networks by approximating $\Delta L(w)$ in the vicinity of w by its first-order Taylor expansion in Equ. (2) where R_1 is the remain term of first order Taylor expansion, as suggested by the prior art [33]. The approximated importance score in Equ. (3) is essentially the product between weight value w_i and weight gradient $\partial L(w_i)/\partial w$, both of which already exist in the training stage and therefore are easy to derive.

$$\Delta L(w) = \sqrt{(L(w = w_i) - L(w = 0))^2} \tag{1}$$

$$L(w = 0) = L(w_i) + \frac{\partial L(w_i)}{\partial w} * w_i + R_1(w = 0) \tag{2}$$

$$\Delta L(w) \approx \sqrt{(\frac{\partial L(w_i)}{\partial w} * w_i)^2} \tag{3}$$

Pattern Pruning. As explained in Sec. IV, the TW pattern requires the column pruning before the row pruning. We first break the weight matrix into column-based tiles (line 4) and then evaluate the importance score of each tile. We then determine the threshold for column pruning based on the

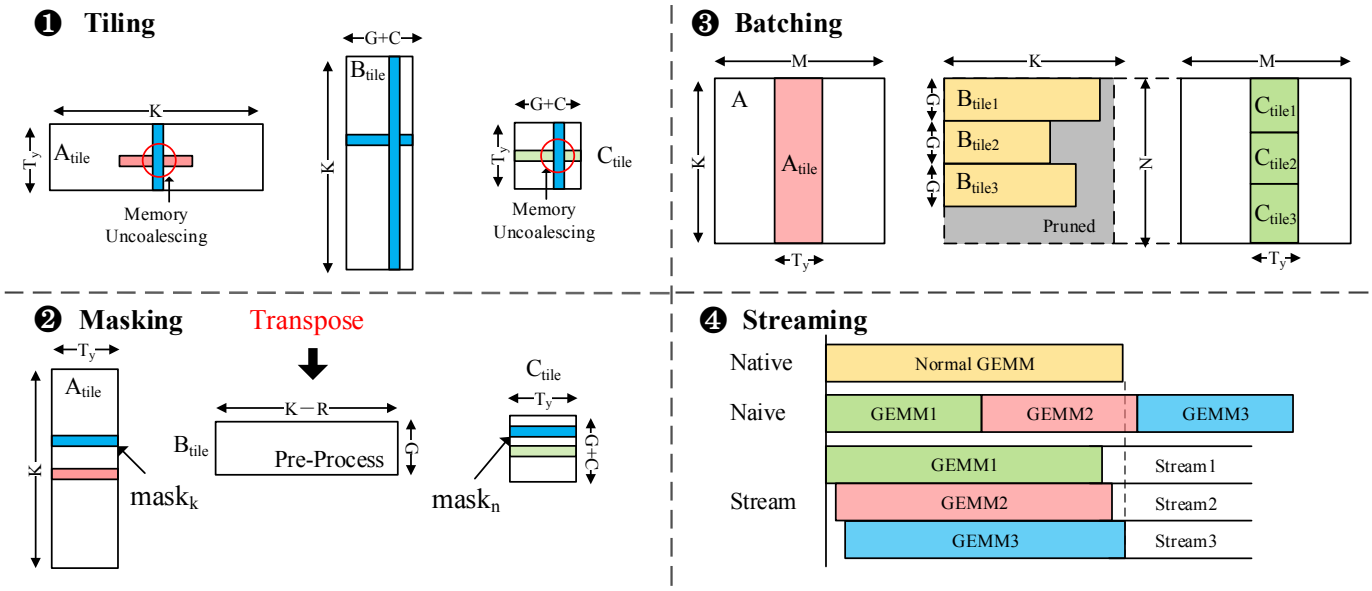


Fig. 7: Our implementation performs a series of steps to optimize the execution efficiency of TW-based sparse models. (1) Naive tiling leads to uncoalesced memory accesses and therefore performance loss. (2) We transpose the matrix for memory access coalescing and use masks for efficient process. (3) We convert the execution of TW to batched GEMM that lets us reuse the existing tensor core-based GEMM kernels. (4) We improve the execution efficiency by leveraging the stream concurrency.

sparsity target in the current stage (line 5). Line 6 applies the apriori tuning that we explain later, and line 8-12 remove the column tiles. Afterwards, we reorganize the the column-pruned matrix to tiles (line 13). Line 14-20 perform the row-pruning, which is similar to the column pruning. After the column and row pruning, the weight matrix becomes compatible to TW. In our algorithm, we design the tiling granularity G as a tunable hyper-parameter, through which we explore the trade-off between the accuracy and performance of the sparse model.

Global Weight Pruning. As shown in Sec. IV-B, there exists an uneven distribution of weight sparsity in different layers of a DNN model, which we use a global weight pruning to exploit. The codes in line 7 and line 15 sort the scores for all tiles in the column and row pruning, respectively. The codes in line 8-12 and line 16-20 prune the tiles from all layers in the DNN model according to their importance rank.

Apriori Tuning. We use EW pattern with the target sparsity as an apriori knowledge to better guide our TW-based pruning algorithm because EW achieves the best model accuracy under the same sparsity. In the EW sparsity results, we observe a strong locality pattern, where more than 10% tiles (columns) are completely pruned (i.e., 100% sparsity) when the pruning target sparsity is 75%. We leverage this observation to augment our pruning algorithm with apriori tuning in Line 6 of Algorithm 1. The detailed apriori tuning algorithm is shown in Algorithm 2. First, we get the tile-level sparsity distribution from the EW results in the target sparsity. We set the top-n maximum sparsity tile score 0, which means high priority to prune. In contrast, we set the last-n tiles a large score and would not be pruned.

VI. Efficient GPU Implementation

This section introduces our efficient GPU implementation that unleashes the algorithmic benefits of TW. Exploiting the unique sparsity pattern of TW, we first describes the basic tiling design, followed by three key optimizations that combines intelligent data layout and concurrency/batching optimizations to maximize the efficiency of TW tiling on tensor cores.

The advantage of TW sparsity pattern is that sparse matrix multiplication could be transformed to dense GEMM, which can be effectively accelerated on dense GEMM accelerators such as the tensor core on GPUs (Sec. IV). Fig. 7 shows how we transform sparse matrix multiplication that have the TM sparsity pattern to a dense GEMM, and how it exploits various GPU characteristics to maximize the performance.

Tiling. We start by tiling matrices as usual. Fig. 7 ❶ illustrates an example, where generating an output tile C_{tile} requires two input tiles A_{tile} and B_{tile}. Each input matrix tile has two mask vectors that indicate which rows and columns in the matrix tile are pruned. In the example of ❶, the blue rows and columns are pruned. We remove the pruned rows and columns in the weight matrix tile B_{tile}, which can be done offline before the model inference starts. The input tile A_{tile} and output tile C_{tile} are stored in the dense format to avoid the preprocess overhead. Their pruned rows/columns are skipped rather than removed.

We modify the dense GEMM kernel such that it skips computing partial sums for pruned elements according to the mask vectors. This reduction of computation is the source of acceleration. Our baseline GEMM implementation is based on the open-sourced CUTLASS [41], which is a high-performance linear algebra CUDA library. It implements three levels of

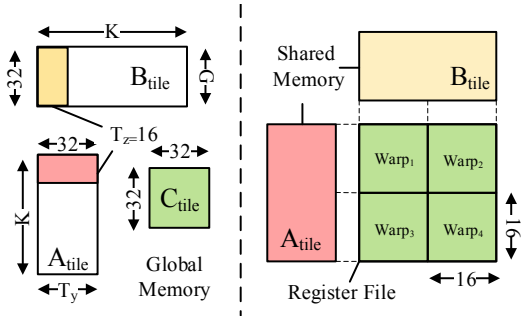


Fig. 8: Warp-level GEMM tiling that exploits tensor core.

tiling to maximize the data reuse in the global memory (thread block tile), shared memory (warp tile), and register file (thread fragment). Meanwhile, it can also leverage the tensor core in the GEMM computation, which we use to accelerate the TW.

However, a naive tiling implementation is inefficient and even causes slowdown compared to the original dense model. In our implementation, we exploit three optimizations that mitigate the inefficiencies and maximize the benefits of TW.

Memory Accesses Coalesce. Naive tiling leads to frequent uncoalesced memory accesses that are inefficient on the single-instruction-multiple-data based GPUs. Fig. 7 ❶ shows the memory access patterns in the original row-major matrix format. The pruned blue row in B_{tile} causes the skip of blue column in A_{tile}. Therefore, a continuous access to the A_{tile} (marked by the red row) that is originally coalesced now becomes uncoalesced, which can cause severe performance degradation as uncoalesced memory accesses require multiple memory transactions. The uncoalesced accesses also exist in the matrix tile C_{tile} (marked by the green row) owing to the pruned blue column in B_{tile}.

We propose to store the matrix tiles in their transposed format to optimize their memory access efficiency. In Fig. 7 ❷ where the three tiles are transposed, the column skipping is converted to the row skipping. Thus, it eliminates the uncoalesced accesses and improves the access efficiency.

Load Imbalance Mitigation. TW sparsity inherently introduces imbalanced tiles. That is, some tiles will require more computations since fewer rows/columns are pruned; other tiles that have more rows/columns pruned will lead to lower computation. Imbalanced tiles leads to resource under-utilization, and thus affects the overall speedup.

We propose to batch tile computations to improve the utilization. Fig. 7 ❸ shows an example where the weight matrix is decomposed into $\lceil \frac{N}{G} \rceil$ tiles, where G is the TW granularity. Different B_{tile} are batched together to share the same A_{tile}. Batching improves resource utilization as a batched GEMM packs multiple tiles and thus increases the computation.

Another practical benefit of batched-GEMM implementation is that we can reuse existing high-performance tensor core-based GEMM kernels and avoid implementing specialized GEMM kernels, each customized for a particular tile size. Fig. 8 illustrates the warp-level tiling and Listing 1 shows the kernel implementation that uses tensor core APIs. We assume that $G = 32$, $T_y = 32$ and $T_z = 16$, which is the minimum tiling granularity as it must be the multiple of 32 (i.e., warp size). A_{tile} and B_{tile} are transposed and stored into the shared memory and C_{tile} to the register file. Then a warp tile will compute the out-product with the tensor core MMA API, which can support the fixed size $(16 \times 16 \times 16)$ matrix multiplication.

Listing 1: GEMM kernel on tensor core.

```
#define G 32
#define T_y 32
#define T_z 16
__global__ void StreamMaskedGEMM(int M, int Pruned_N, int
    Pruned_K, half *A, half *B, half *C, half alpha, half
    beta, int *mask_n, int *mask_k){
  //Allocate C_tile in Register File.
  half C_tile[G * T_y];
  //Allocate A_tile and B_tile in Shared Memory.
  __shared__ half A_tile[T_y * T_z];
  __shared__ half B_tile[G * T_y];
  for(int k = 0; k < K; k+=T_z){
    //Load A_tile from Global to Shared Memory skipping
        the pruned row with mask_k.
    Load_A_Tile_with_Mask(A_tile, A, mask_k);
    //Load B_tile from Global to Shared Memory with
        Pre-Processed B.
    Load_B_Tile(B_tile, B);
    //Tensor core API: WMMA with fixed 16x16x16 GEMM.
    WMMA::MMA(C_tile, A_tile, B_tile, alpha, beta);
  }
  //Store C_tile from Register File to Global Memory
      skipping the pruned row with mask_n.
  Store_C_Tile_with_Mask(C, C_tile, mask_n);
}
```

While batching mitigates resource under-utilization, we find that it is possible that the computation of a batch still under-utilizes the GPU resources. We leverage concurrent kernel execution on modern GPUs [38] to further improve resource utilization. In the studied NVIDIA GPU platform, we overlap the computation of different tiles by assigning to different streams, and rely on the underlying scheduler to maximize the resource utilization. Fig. 7 ❹ shows an example where naively running different batches could have a lower performance than the original unpruned GEMM. Concurrently executing multiple batches with different streams improves the performance.

Kernel Fusion. Complex DNN models necessarily incorporate non-GEMM computations. For instance, the BERT model spends about 39% time on non-GEMM kernels, such as Add-bias and BatchNormalization, constituting the "Amdahl's law bottleneck." Meanwhile, our memory coalescing optimization also introduces additional transpose kernels, which can incur a large performance overhead if leave unoptimized.

As such, we propose to fuse consecutive non-GEMM kernels to improve the performance. Kernel fusion has two advantages. First, we reduce the number of kernels to reduce the launch time. Second, fused kernel reduces access to global memory and shares the register resources. For example, the previous Add-bias operation can execute with BatchNormalization when the data is loaded into the register file. We also modify the memory access behavior in those non-GEMM kernels to reduce the number of transpose kernels. With this modification, we only need to transpose matrix A in the first layer and transpose matrix C after the last layer, which significantly reduces the transpose overhead. We also apply the kernel fusion optimization to the dense model

baselines for the fair comparison, which, for example, reduces the 39% non-GEMM execution time in BERT to 29%.

VII. Evaluation

In this section, we demonstrate that TW is able to maintain the accuracy of sparse DNN models and provide the significant execution speedup over the dense model and other sparsity patterns at the same time. We first explain our evaluation methodology with the use of state-of-the-art DNN models on the GPU equipped with tensor cores. We then study the design space of TW to explore the trade-off between model accuracy and latency. In the end, we select the representative configurations of TW and compare it with other sparsity patterns, which demonstrates the acceleration capability of TW.

A. Methodology

Benchmark. We evaluate three popular neural networks, VGG (CNN), NMT (LSTM), and BERT (Transformer), which cover tasks from the computer vision and and NLP domain.

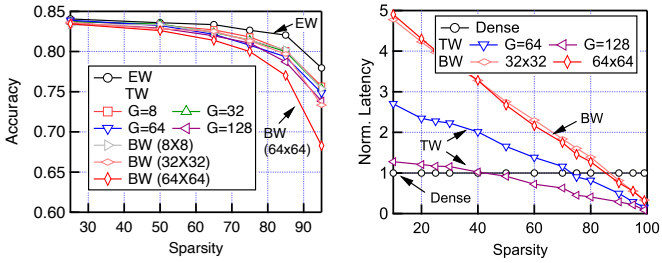
VGG16 [53] is a popular CNN model with 13 convolutional layers and 3 fully connected layers. We evaluate its accuracy for image classification on the ImageNet [25] dataset with 1.2 million training images and 50,000 validation images. We prune its weight matrix after applying the im2col method [7], which flattens the filters in the same channel to a column and different columns correspond to different channels (so the flattened feature map matrix left multiplies the flattened weight matrix in Fig. 4). This approach is similar to prior work [70].

We evaluate the accuracy of NMT model, which adopts the attention based encoder-decoder architecture, for the machine translation task [9]. We reproduce the model with open source framework [31]. We evaluate the NMT model on the IWSLT English-Vietnamese dataset [32], and use the BLEU (bilingual evaluation understudy) score [42] as the accuracy metric.

For the state-of-art Transformer model family, we use the BERT-base with 12 layers. The two evaluated downstream tasks are the sentence-level classification on the widely used GLUE (general language understanding evaluation) dataset [58] and the more challenging question answering task on the SQuAD dataset [48], [49]. The GLUE dataset is a composite dataset with 10 different sub-tasks, which we evaluate 6 out of them.

In our experiments, we use the pre-trained models that can achieve their reported reference accuracies. We then apply EW, VW, BW, and our proposed TW sparsity patterns to prune the dense models according to the algorithm described in Sec. V. We use the TensorFlow [1] framework for fine-tuning. Depending on the dataset size, we perform the fine-tuning for 4-10 epochs at each target sparsity level, which is sufficient to saturate the model accuracy in our experiment.

Baselines. We compare the proposed TW with EW, VW, and BW. For the latency evaluation, we execute EW and VW using the cuSparse [40] library, and execute BW using the BlockSparse [55] library released by the authors. Our TW implementation (Sec. VI) is based on CUTLASS [41], an open-source, high-performance GEMM template library. For all those libraries including TW, we modify the original model codes to explicitly call each library. In the rest of this section, we focus on the GEMM execution time unless explicitly mentioned.



(a) Accuracy. (b) Normalized latency.

Fig. 9: Accuracy and latency of sparse BERT model in TW and other patterns with different granularities. All latencies are measured on tensor cores and normalized to the dense model.

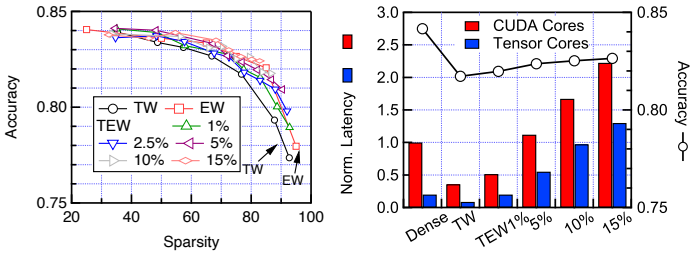
All the experiments are conducted on the Tesla V100 GPU [39], which has a peak throughput of 15.7 TFLOPS and 125 TFLOPS for the CUDA cores and tensor cores, respectively. The EW implementation runs only on the CUDA core with the cuSparse library and the BW implementation runs only on the tensor core with BlockSparse. The convolution operations in the CNN workloads are converted to GEMM by the im2col method [7]. The models are all trained using FP32. All inferences on CUDA cores are done using FP32, and all inferences on tensor cores are done using FP16.

B. BERT Results and Design Space Exploration

We now study the design space of TW, which is the tiling granularity G, to explore the trade-off between model accuracy and inference latency. In addition, we also evaluate the hybrid TEW pattern, which extends the trade-off space in sparse models. The analysis is case-studied on the BERT model for sentence pair entailment task on the MNLI dataset. We report the results of other models/tasks/datasets in the next subsection.

Impact of TW Granularity. We first explore the impact of tiling granularity G for TW-based pruning. Fig. 9a compares the accuracy of EW, BW, and TW. The most fine-grained EW achieves the best model accuracy as expected. When sparsity is less than 50%, all the granularities evaluated have similar accuracies, suggesting that the BERT model is at least 50% redundant. In particular, at a sparsity of 75%, our proposed TW with $G = 128$ has an accuracy loss of about 0.9% and 2.4% compared to EW and the baseline dense model, respectively. As the sparsity increases, the accuracy drop becomes more significantly. The most coarse-grained BW (64×64) experiences the most drastic accuracy drop of 4% at 75% sparsity.

The accuracy drop of TW increases slightly with a larger G value. This is because the larger G value puts more strict constraint on the pruning shape, but larger G also means greater latency reduction. We find that G of 128 is sufficient to maintain the model accuracy while providing significant latency reduction. Fig. 9b compares the latency of the dense model, BW, and TW on the tensor core. With only 40% sparsity,



(a) Accuracy. (b) Normalized latency.

Fig. 10: Accuracy and latency of TEW-based sparse BERT model with different δ values, which determine the portion of added EW elements. All latency values in (b) are normalized to the latency of dense model on CUDA cores.

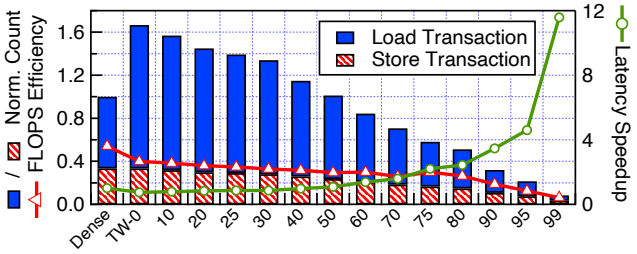


Fig. 11: The scalability (up to 99% sparsity) of latency speedup and corresponding performance counters for sparse TW-based BERT model. TW-10 means the model with 10% TW sparsity.

TW with $G = 128$ starts to outperform the dense model latency. At a 75% sparsity, TW−128 achieves a speedup of 2.26×. In contrast, BW with a block size of 64×64 is faster than the dense model only when the sparsity is greater than 90%, which leads to an accuracy loss as high as 10% , as shown in Fig. 9a.

Impact of δ in TEW. We evaluate the impact of δ in TEW, which determines the amount of EW pattern imposed on TW (Sec. IV). Fig. 10a compares the sparse BERT model accuracy of different sparsity levels with EW, TW, and TEW patterns. The accuracy of sparse model with TW is lower than EW. On the other side, TEW can mitigate the accuracy loss in TW through adding a small portion EW pattern, which is controlled by the δ parameter in Sec. IV-A. For instance, with $\delta = 5\%$, the TEW accuracy catches up with EW.

Fig. 10b compares the latency (left y-axis) and accuracy (right y-axis) of the dense model and various TW and TEW models with the fixed 75% sparsity. We show the latency results on both the tensor cores and the CUDA cores, which are all normalized to the dense model latency on CUDA cores.

On the tensor cores, TW achieves 2.26× speedup than the dense model. TEW achieves no speedup at $\delta = 1\%$ compared to the dense model, and its performance is worse as δ increases. This is because the irregular portion of TEW (i.e., the EW portion) could not be executed on the dense tensor cores and, instead, has to be executed on the CUDA Cores, which is about 8× slower than the tensor cores. To illustrate the point, we show the results of running different sparse models on CUDA cores only. Using CUDA Cores alone, TEW with $\delta = 1\%$ is about 2× faster than the dense model. Thus, we expect that TEW is useful in resource-constraint scenarios such as low-end GPUs with less or even no tensor cores, or mobile systems.

Speedup Scalability. We also study the speedup scalability by intentionally pruning DNN models to an extreme sparsity level. It is highly likely that the size of DNN models would continue to grow [2]. Fig. 11 shows the latency speedup of sparse BERT model with TW on tensor cores until 99% sparsity. At the 99% sparsity level, TW with $G = 128$ achieves 11.6× speedup, demonstrating its significant acceleration potential.

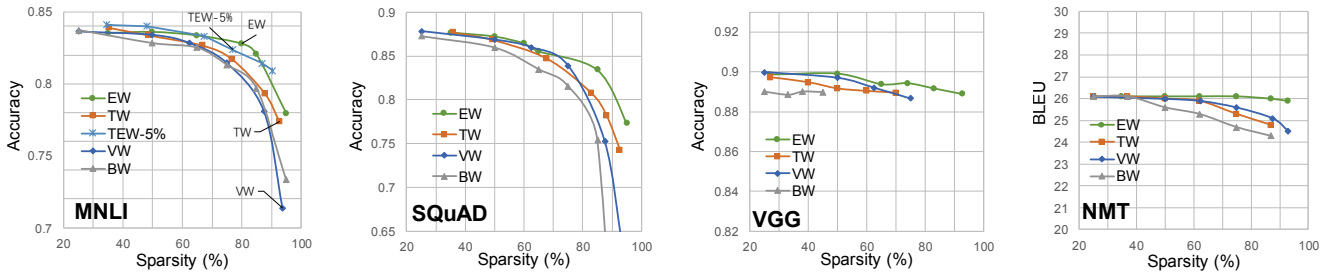
Performance Counters. Fig. 11 also shows the total number of global memory load/store requests and FLOPS (floating operations per second) efficiency for running the sparse TW-based BERT model, which are all normalized to the dense BERT model. Compared to the original BERT model, our TW implementation with zero sparsity generates twice of global memory request owing to the masking overhead, for which we use the int32 format. The extra load traffic leads to about 35% performance loss. With about 40% sparsity, the benefit outperforms the overhead, leading to the net latency speedup. The FLOPS efficiency equals the measured FLOPS divided by all tensors' peak FLOPS. The sparse TW model maintains a relatively high FLOPS efficiency until 80% sparsity and quickly drops after that owing to the reduced computation demand.

C. Comparison with Other Patterns

We compare the accuracy and latency speedup of TW with EW, VW, BW on three different models. We perform the comprehensive evaluation of BERT model for the sentence classification task on the composite GLUE dataset, which includes ten different datasets. We observe the similar results on 6 studied datasets and therefore only report the result on the largest dataset MNLI. We also report its result on the question answering task with the SQuAD dataset. For the latency speedup, we report the results on the V100 GPU using tensor cores and CUDA cores separately.

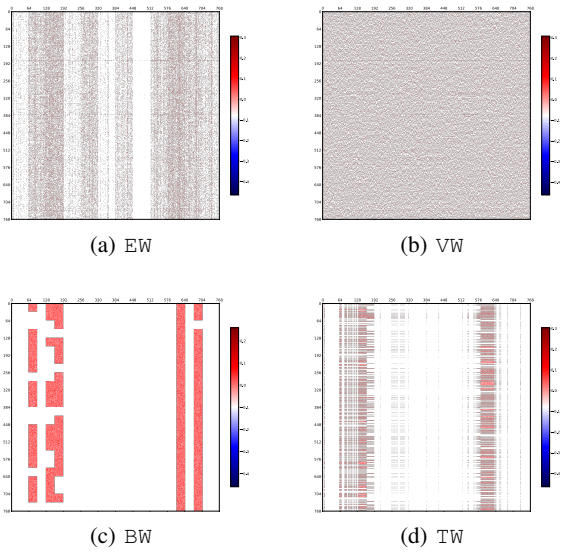
Accuracy. Fig. 12 shows the accuracy of different models with different pruning patterns. The granularity of TW is 128 and block size for BW is 32×32, which balance the accuracy and latency speedup as our previous design space analysis suggests. The vector size of VW is set to 16 as used in the original paper [70]. EW reaches the best accuracy of all the evaluated algorithms and BW has the worst accuracy under the same sparsity. The accuracy of TW and VW are similar when the sparsity is below 70%. With high sparsity (> 70%), TW generally outperforms the VW with the exception of NMT.

TW achieves the better accuracy when sparsity is high because it allows the uneven distribution sparsity in a weight matrix. Fig. 13 shows the resulted weight sparsity distributions of layer 0 of BERT under the 75% sparsity for different patterns. The EW result shows that there exits uneven distribution across the matrix. And VW is unable to fit this characteristic because it forces all prune units (vector) to have the sparsity. In contrast, both BW and TW can adapt to this sparsity locality.



(a) MNLI (b) SQuAD (c) VGG (d) NMT

Fig. 12: The accuracy comparison of different models on the model-specific downstream tasks with various pruning patterns and varying sparsity levels. Plot (a) and (b) are BERT models evaluated on GLUE dataset and SQuAD dataset, respectively. Plot (c) is VGG16 model evaluated on ImageNet. Plot (d) is LSTM model evaluated on NMT task.



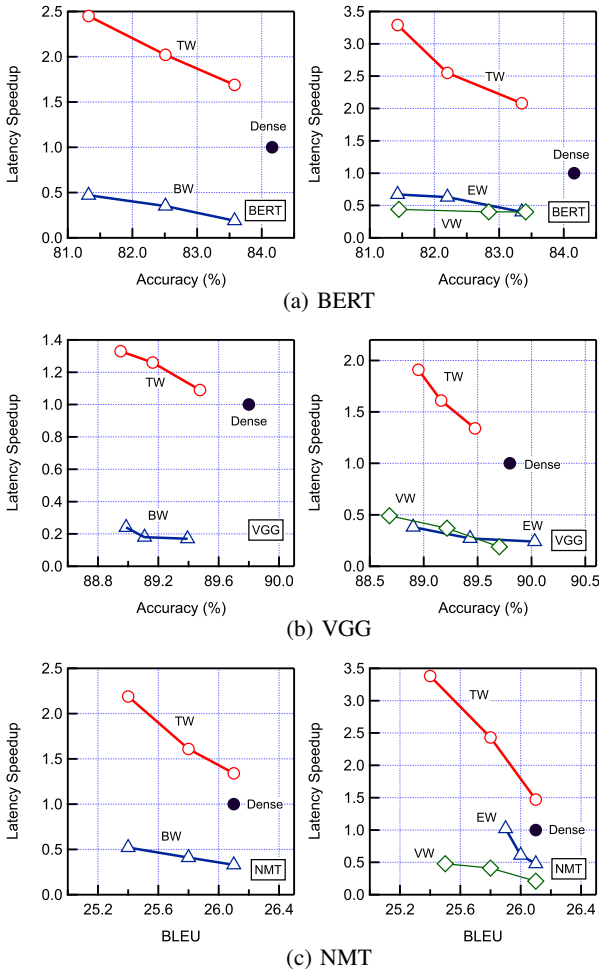
(a) EW (b) VW (c) BW (d) TW

Fig. 13: Different pruning patterns under 75% sparsity on layer 0 attention matrix ω_Q in BERT model.

Meanwhile,VW cannot adapt to the uneven sparsity distribution across different layers as explained in Sec. IV-B while TW can. For the NMT model, both VW and TW experiences a rapid accuracy drop compared to EW when the sparsity is over 60%, which suggests this model prefers irregular sparsities. VW slightly outperforms TW owing to its smaller granularity of 16.

Speedup vs Accuracy. Fig. 14 compares the trade-off of latency speedup and model accuracy based on TW and other patterns including BW, VW, and EW. In specific, we compare the TW and BW running on the tensor cores, and compare the TW, VW, and EW on the CUDA cores. The speedup is calculated against dense models on the tensor cores and CUDA cores separately. The experimental results demonstrate that only TW can extend the latency-accuracy Pareto frontier on tensor cores and CUDA cores. In contrast, other sparsity patterns lead to both longer latency and lower accuracy than the dense model.

Finally, we compare the latency speedup of various patterns with the same level of accuracy drop (BERT with $< 3\%$ drop, VGG with $< 1\%$ drop and NMT with < 1 BLEU drop). On



(a) BERT (b) VGG (c) NMT

Fig. 14: The trade-off between latency speedup and model accuracy. The speedup is calculated on tensor cores (left column) and CUDA cores (right column) separately.

tensor cores, TW achieves an average speed up of 1.95× while BW is 0.41×. On CUDA cores, TW achieves an average speed up of 2.86× while EW and VW are 0.69× and 0.47×. TW achieves the meaningful latency reduction on both tensor cores and CUDA cores owing to its compatibility with dense GEMM,

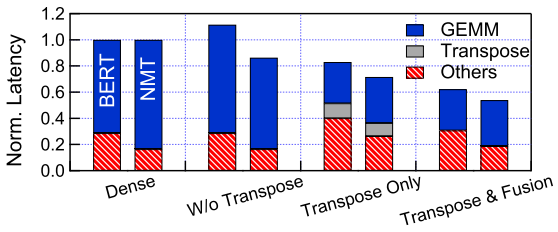


Fig. 15: The end-to-end latency breakdown for sparse TW-based BERT and NMT with 75% sparsity ($<3\%$ accuracy drop and <1 BLEU drop than the dense model respectively). We enable the transpose and fusion optimization by default.

while all other sparsity pattern cause the actual slowdown.

D. End-to-end Latency and Impact of Optimizations

The above performance comparison between the dense model and various sparse models only considers the GEMM-related computation. We now study the end-to-end latency and impact of optimizations presented in Sec. VI, which include the transposed matrix storage and kernel fusion. Fig. 15 shows the end-to-end latency breakdown with different optimization combinations when running the sparse TW BERT model with 75% sparsity. We do not use the VGG in this experiment because it only includes 5% non-GEMM computations. Without performing the matrix transpose optimization, the GEMM computation cannot benefit from the high sparsity. The transpose kernel takes about 10% of overall latency without fusion. With the transpose and fusion, the GEMM-only speedup for BERT and NMT is 2.26× and 2.38× respectively, while the end-to-end speedup is 1.61× and 1.86× on tensor core.

VIII. Related Work and Discussion

Architectural Support for Sparse DNN. Recently, designing efficient architectures for sparse DNN models has become an active research topic. There are many prior works to dealing with sparsity through architectural support [11], [20], [23], [52], [65], [68]. ExTensor [20] proposes a novel approach to accelerate tensor algebra kernels using the principle of hierarchical computation elimination in the presence of sparsity. Sparse ReRAM Engine [65] exploits both the weight and activation sparsity to accelerate the DNN model. The channel gating [23] is a fine-grained dynamic pruning technique for CNN inference. Those work are all based on customized ASIC or FPGA. There are also prior works [17], [70] that focus on optimizing the tensor core for better performance or flexibility.

Software Optimization for Sparse DNN. There are also previous works that propose software optimization for sparse model acceleration on modern architectures without hardware modification. Prior work proposes an efficient mixed-mode representation called MM-CSF for sparse tensor, which partitions nonzero elements into disjoint sections for performance acceleration [36]. CFS SpMV is a new optimization strategy for the sparse matrix-vector multiplication and shows high performance on multicore architectures [13]. DCSR (densified compressed sparse row) is well-suited to GPU architecture for sparse computation using the near-memory strategy [14].

TW on Other Platforms. Although we only implement TW on the GPU platform, it is quite possible to support our sparsity pattern on other platforms like TPU [24]. The fundamental requirement of supporting TW is the medium size GEMM. Our evaluation shows that TW with $G=128$ strikes a balance between model accuracy and latency, which implies the requirement of $128 \times N \times 128$ GEMM. The latest TPU [4] adopts a relatively large systolic array (128×128), which meets the aforementioned requirement. However, it only exposes high-level programming interface like GEMM, which makes the other optimization like streaming concurrency difficult. In other words, supporting TW on other platforms like TPU is feasible if their low-level programming interfaces are exposed.

Sparsity Patterns. The sparsity pattern plays an important role in both the model accuracy and architecture design for sparse DNNs. Zhu et al. propose the vector-wise pruning pattern and the corresponding sparse tensor core architecture [70]. The vector-wise pruning pattern adds constraints on the sparsity of each pruning unit to guarantee the pruned matrices to be acceleration-friendly. They reported accuracy results on popular CNN and RNN models. However, this method fails to capture the uneven sparsity distribution across different model layers, which limits its pruning effect. Narang et al. propose block-wise pruning pattern [35]. This pattern has the prune unit as a block, making it execution-friendly on the dense GPU architecture. However, their method has a strong constraint on the pruning shape which impacts the model accuracy significantly.

Recent work also explores energy-oriented pruning, targeting both accelerators [62], [63] and general-purpose processors [61], [64]. Our work removes redundant computations and thus could also reduce energy consumption. Yang et al. [60] demonstrates quantization-pruning joint compression; we leave it to future work to explore how to integrate *tile sparsity* with quantization.

IX. Conclusion

In this work, we propose to co-design the tiling of matrix multiplication and DNN model pruning pattern, with the purpose of balancing the irregularity for the model accuracy and compatibility for dense GEMM computation. We study an efficient software-only implement of our proposed sparsity pattern, TW, that leverages the tensor core accelerator and concurrency features in the GPU. We demonstrate its capability of model accuracy preserving and high performance speedup on the state-of-the-art DNN models.

Acknowledgement

We thank the anonymous reviewers for their constructive feedback for improving the work. This work was supported by National Key R&D Program of China (2019YFF0302600), the National Natural Science Foundation of China (NSFC) grant (61702328 and 61832006). Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[3] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 6372. [Online]. Available: https://doi.org/10.1145/3289602.3293898

[4] C. Chao and B. Saeta, "Cloud TPU: Codesigning Architecture and Infrastructure," https://www.hotchips.org/hc31/HC31_T3_Cloud_TPU_Codesign.pdf, 2019.

[5] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," 2006.

[6] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[7] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[8] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *arXiv preprint arXiv:1904.10509*, 2019.

[9] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[10] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning, "What does bert look at? an analysis of bert's attention," *arXiv preprint arXiv:1906.04341*, 2019.

[11] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, "Tie: energy-efficient tensor train-based inference engine for deep neural network," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 264–278.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[13] A. Elafrou, G. Goumas, and N. Koziris, "Conflict-free symmetric sparse matrix-vector multiplication on multicore architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–15.

[14] D. Fujiki, N. Chatterjee, D. Lee, and M. O'Connor, "Near-memory data transformation for efficient sparse matrix multi-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–17.

[15] Google, "Understanding searches better than ever before," https://www.blog.google/products/search/search-language-understanding-bert/, 2019.

[16] C. Guo, Y. Zhou, J. Leng, Y. Zhu, Z. Du, Q. Chen, C. Li, B. Yao, and M. Guo, "Balancing Efficiency and Flexibility for DNN Acceleration via Temporal GPU-Systolic Array Integration," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[17] ——, "Balancing Efficiency and Flexibility for DNN Acceleration via Temporal GPU-Systolic Array Integration," *arXiv preprint arXiv:2002.08326*, 2020.

[18] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA 16. IEEE Press, 2016, p. 243254. [Online]. Available: https://doi.org/10.1109/ISCA.2016.30

[19] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[20] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[21] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Deftnn: Addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 786–799.

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

[23] W. Hua, Y. Zhou, C. De Sa, Z. Zhang, and G. E. Suh, "Boosting the performance of cnn accelerators with dynamic fine-grained channel gating," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 139–150.

[24] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of The ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[26] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 821834. [Online]. Available: https://doi.org/10.1145/3297858.3304028

[27] Y. LeCun, Y. Bengio *et al.*, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[28] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in neural information processing systems*, 1990, pp. 598–605.

[29] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 487–498. [Online]. Available: https://doi.org/10.1145/2485922.2485964

[30] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5058–5066.

[31] M. Luong, E. Brevdo, and R. Zhao, "Neural machine translation (seq2seq) tutorial," *https://github.com/tensorflow/nmt*, 2017.

[32] M.-T. Luong and C. D. Manning, "Achieving open vocabulary neural machine translation with hybrid word-character models," in *Association for Computational Linguistics (ACL)*, Berlin, Germany, August 2016. [Online]. Available: https://nlp.stanford.edu/pubs/luong2016acl_hybrid.pdf

[33] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 264–11 272.

[34] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2016.

[35] S. Narang, E. Undersander, and G. Diamos, "Block-sparse recurrent neural networks," *arXiv preprint arXiv:1711.02782*, 2017.

[36] I. Nisa, J. Li, A. Sukumaran-Rajam, P. S. Rawat, S. Krishnamoorthy, and P. Sadayappan, "An efficient mixed-mode representation of sparse tensors," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–25.

[37] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren, "Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 20, 2020.

[38] NVIDIA, "GPU Pro Tip: CUDA 7 Streams Simplify Concurrency," https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/, 2015.

[39] ——, "NVIDIA Volta GPU Architecture Whitepaper," 2017.

[40] ——, "CUDA Toolkit Documentation v10.1," 2019.

[41] ——, "CUTLASS 1.3," https://github.com/NVIDIA/cutlass, 2019.

[42] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.

[43] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA 17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2740. [Online]. Available: https://doi.org/10.1145/3079856.3080254

[44] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 58–70.

[45] Y. Qiu, J. Leng, C. Guo, Q. Chen, C. Li, M. Guo, and Y. Zhu, "Adversarial Defense Through Network Profiling Based Path Extraction," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4777–4786.

[46] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[47] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 79–92.

[48] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for squad," *arXiv preprint arXiv:1806.03822*, 2018.

[49] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[50] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient gpu architectures," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 86–98.

[51] U. A. ROMAN STEINBERG, "6 areas where artificial neural networks outperform humans," https://venturebeat.com/2017/12/08/6-areas-where-artificial-neural-networks-outperform-humans/, 2018.

[52] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 347–358.

[53] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR 2015 : International Conference on Learning Representations 2015*, 2015.

[54] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," *CoRR*, vol. abs/1906.02243, 2019. [Online]. Available: http://arxiv.org/abs/1906.02243

[55] P. Tillet, "Torch-Blocksparse," https://github.com/ptillet/torch-blocksparse, 2020.

[56] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/pdf/1706.03762.pdf

[57] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[58] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," in *ICLR 2019 : 7th International Conference on Learning Representations*, 2019.

[59] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in neural information processing systems*, 2016, pp. 2074–2082.

[60] H. Yang, S. Gui, Y. Zhu, and J. Liu, "Automatic neural network compression by sparsity-quantization joint learning: A constrained optimization-based approach," *International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[61] H. Yang, Y. Zhu, and J. Liu, "Ecc: Energy-constrained deep neural network compression via a bilinear regression model," *International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[62] ——, "Energy-constrained compression for deep neural networks via weighted sparse projection and layer input masking," *International Conference on Learning Representations (ICLR)*, 2019.

[63] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5687–5695.

[64] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *Proceedings of the European Conference on Computer Vision*, 2018, pp. 285–300.

[65] T.-H. Yang, H.-Y. Cheng, C.-L. Yang, I.-C. Tseng, H.-W. Hu, H.-S. Chang, and H.-P. Li, "Sparse reram engine: joint exploration of activation and weight sparsity in compressed neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 236–249.

[66] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced sparsity for efficient dnn inference on gpu," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 5676–5683.

[67] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis, "Nisp: Pruning networks using neuron importance score propagation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 9194–9203.

[68] J. Zhang, X. Chen, M. Song, and T. Li, "Eager pruning: algorithm and architecture support for fast training of deep neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 292–303.

[69] T. Zhang, S. Ye, K. Zhang, X. Ma, N. Liu, L. Zhang, J. Tang, K. Ma, X. L. Lin, M. Fardad, and Y. Wang, "Structadmm: A systematic, high-efficiency framework of structured weight pruning for dnns," 2018.

[70] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 359–371.