# E.T.: Re-Thinking Self-Attention for Transformer Models on GPUs

Shiyang Chen[1,+], Shaoyi Huang[2,+], Santosh Pandey[1], Bingbing Li[2], Guang R. Gao[3], Long Zheng[3], Caiwen Ding[2] and Hang Liu[1]

[+]These authors contributed equally.

[1]Stevens Institute of Technology

[2]University of Connecticut

[3]University of Delaware

## ABSTRACT

Transformer-based deep learning models have become a ubiquitous vehicle to drive a variety of Natural Language Processing (NLP) related tasks beyond their accuracy ceiling. However, these models also suffer from two pronounced challenges, that is, gigantic model size and prolonged turnaround time. To this end, we introduce E.T. that rE-thinks self-attention computation for Transformer models on GPUs with the following contributions: First, we introduce a novel self-attention architecture, which encompasses two tailored self-attention operators with corresponding sequence length-aware optimizations, and operation reordering optimizations. Second, we present an attention-aware pruning design which judiciously uses various pruning algorithms to reduce more computations hence achieves significantly shorter turnaround time. For the pruning algorithms, we not only revamp the existing pruning algorithms, but also tailor new ones for transformer models. Taken together, we evaluate E.T. across a variety of benchmarks for Transformer, $\text{BERT}_{\text{BASE}}$ and DistilBERT, where E.T. presents superior performance over the mainstream projects, including the popular Nvidia Enterprise solutions, i.e., TensorRT and FasterTransformer.

**ACM Reference Format:**
Shiyang Chen[1,+], Shaoyi Huang[2,+], Santosh Pandey[1], Bingbing Li[2], Guang R. Gao[3], Long Zheng[3], Caiwen Ding[2] and Hang Liu[1]. 2021. E.T.: Re-Thinking Self-Attention for Transformer Models on GPUs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476138

## 1 INTRODUCTION

Transformer-based models have become a ubiquitous vehicle to drive a variety of NLP-related tasks beyond their accuracy ceilings, such as machine translation [28], text summarization [63], speech recognition [13], and question-answer systems [50]. Recently, transformer models have also achieved impressive performance for computer-vision related tasks [22], e.g., iGPT [6] and ViT [14] obtain similar performance as the state-of-the-art CNN-based networks. This phenomenon demonstrates the enormous potential of Transformer-based models. Given the success of attention, a variety of attention mechanisms have surged [28]. Among them, self-attention [9], which relates various tokens in a single sequence to derive a sequence representation, stands out. Finally, Transformer [51] makes a breakthrough to, as opposed to relying upon recurrent neural network (RNN), base the model entirely on a multi-head self-attention mechanism.

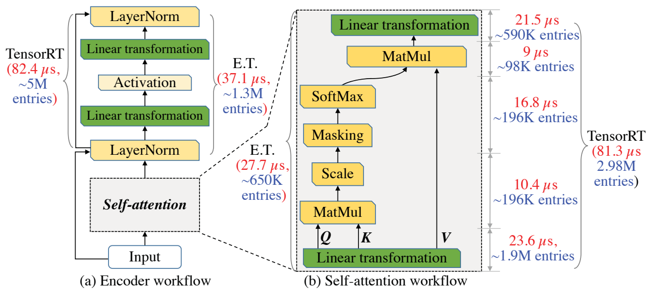


**Figure 1: The architecture of a four-head encoder. The time consumption is measured on WikiText-2 dataset [30], where the input sequence has 128 tokens. Our pruning ratio is 80%.**

Despite that Transformer-based models have made remarkable triumphs, their gigantic model size is a widely recognized roadblock that concerns real-world applications. Figure 1 depicts the architecture of one encoder, which is the building block in Transformer models. Briefly, one encoder takes as input the word embeddings of a sequence. These embeddings pass through the self-attention mechanism to produce an attention matrix. This matrix is fed through layer normalization, linear transformation, and activation to derive the output. Various Transformer model variants often stack a collection of encoders and decoders together. Note, the architecture of the decoder is similar to encoder [51], and more details about the attention mechanism are discussed in Section 2.1. As shown in Figure 1, one encoder can easily reach millions of entries with multiple heads. When it goes to the extreme, e.g., GPT-3 [3], the size of the model can soar up to 175 billion parameters.

Such a large model size, together with the "modular system implementation concept", often leads to the prolonged turnaround time for transformer-based models. Particularly, "modular system implementation" separates a program into independent modules such that each module fulfills necessary but one aspect of the functionality in order to reduce software development efforts. For instance, a vast majority of Transformer-based libraries [1, 17, 21, 42, 44, 53] simply dissect a Transformer encoder into a collection of basic matrix operations, such as dense matrix multiplication, sampled dense matrix multiplication, and matrix-scalar multiplication. Afterwards, they resort to existing CUDA libraries (e.g., cuBLAS [33] and CUTLASS [35]) to implement various transformer models. However, this design leads to two performance issues: (i) One often has to transfer the output matrix from one operator to another in the GPU global memory; (ii) Switching between operators introduces on- and off- chip data movement because the lifetime of an on-chip variable cannot go across kernels. Although the state-of-the-art optimizations, such as vertical and horizontal kernel fusion from TensorRT [40], can mitigate the overhead of issue (i), issue (ii) unfortunately remains. The root cause lies in the fact that TensorRT cannot change how each operator is implemented.

This phenomenon provokes us to ask a question: **Can we introduce some transformer-specific primitives that can perform various matrix computations in a single operator hence largely alleviate the overheads caused by issues (i) and (ii)?** The motivation is as follows: despite that a single encoder of TensorRT is already very fast, i.e., ~160 $\mu s$, real-world applications are still in pursuit of faster encoders. Evidence piles up: transformer models are often used for time-critical applications, such as self-driving [46] and real-time translation [28], where shorter turnaround time is always preferred.

To achieve this goal, we redesign the self-attention architecture with new operators, as well as introduce interesting attention-aware and tensor core friendly pruning designs. As shown in Figure 1, E.T. can reduce the computation time of a single encoder by 2.5×, as well as reduce the model size by 80% on the WikiText-2 dataset. Particularly, E.T. makes the following contributions:

First, we introduce a novel self-attention architecture, which encompasses two tailored self-attention operators with corresponding sequence length-aware optimizations, and operation reordering optimizations. Particularly, (i) our on-the-fly attention operator resolves the data dependency across five operators in self-attention, i.e., the yellow boxes in Figure 1(b), and performs these five operations in one operator with the help of shared memory and registers, hence avoids expensive global memory accesses for the intermediate results. This is fundamentally different from kernel fusion optimizations in TensorRT [40]. (ii) We study the self-attention architecture and find the opportunity of combining the linear transformation for matrix **V** and the final linear transformation operator in Figure 1(b). The newly combined operator lowers the required pruning ratio and can be pre-computed to potentially avoid computations. (iii) Our sequence length aware optimization explores the corner case performance issues for our on-the-fly attention operator and unveils that performing fewer computations on-the-fly would yield more benefits when sequence length is relatively long. (iv) Finally, we identify that using pure FP16 of the tensor core would experience overflow problems during self-attention computation. Correspondingly, we reorder the scaling operator to achieve pure FP16-based self-attention computation which is more efficient than the mixed precision-based counterpart. As shown in Figure 1, our novel self-attention computation gains a speedup of 2.9× over the existing TensorRT implementation.

Second, we propose an attention-aware and tensor core friendly pruning design which judiciously uses various pruning algorithms to reduce more computations hence achieving significantly shorter turnaround time. Despite that pruning is a well-known strategy to reduce the mounting size of Transformer-based models, it is concerned that either irregular pruning [23] (i.e., pruning weights of arbitrary locations) will prevent the usage of tensor core or block-based weight pruning [31] (prune at the tensor tile level) would suffer from low pruning ratio and accuracy loss. In this paper, we first strive to derive tensor core friendly matrix representations for weight matrices in column and row pruning, which allows us to both enjoy the computation reduction in pruning and exploit existing tensor core-based highly optimized GEneral Matrix Multiplication (GEMM) routines for rapid linear transformation. Further, to achieve both tensor core friendly and high pruning ratio & accuracy, we present the first tensor tile-based pruning algorithm for transformer models using a reweighted method on group lasso regularization [4]. This algorithm first partitions the weight matrix into tensor tiles, subsequently uses $l_2$ norm to update the penalty factor. We then update the total loss and conduct pruning based on $l_2$ norm. Finally, we retrain the non-zero entries in the model in order to preserve the accuracy after pruning. Last but not least, we introduce an attention-aware adaptive pruning algorithm design that chooses different pruning algorithms for various weight matrices that can reduce more computations and achieve shorter turnaround time.

Third, we demonstrate the effectiveness of E.T. across a wide range of benchmarks and models. Particularly, we evaluate our pruning algorithm on WikiText-2 dataset for Transformer, and GLUE benchmark suite for $\text{BERT}_{\text{BASE}}$ and DistilBERT. This evaluation covers a significantly wider range of benchmarks and types of models comparing to the recent art [21]. It is of particular importance to mention that our novel self-attention architecture, together with attention-aware pruning offers, on average, 1,131 $\mu s$ ($\text{BERT}_{\text{BASE}}$) and 500$\mu s$ (DistilBERT) inference latency across all the GLUE benchmarks while sustains 95% of the prediction accuracy/score (Table 1). **To the best of our knowledge, E.T. is the first work that achieves such short turnaround time for transformer-based models on GPUs while maintaining the high prediction accuracy**. This is benefited from our novel attention-aware pruning algorithm designs and efficient self-attention computation system implementations.

The remaining of this paper is organized as follows: Section 2 presents the background. Sections 3 and 4, respectively, discuss the efficient self-attention computation and attention-aware tensor core friendly pruning algorithm designs. We evaluate E.T. in Section 5, describe related work in Section 6, present the potential of E.T. in Section 7, and conclude in Section 8.

## 2 BACKGROUND

### 2.1 Transformer-based Language Models

Transformer is a sequence-to-sequence NLP model [51] which uses an attention mechanism to draw global dependencies between input and output. It takes a sequence of word embeddings from one vocabulary set as input and generates the probability of tokens in the other vocabulary set. The model mainly consists of encoding and decoding components. The encoding component is a stack of encoders that are all identical in structure but their weights are trained independently. Likewise, the decoding component is also a stack of decoders. Note, the number of encoders and decoders can be adjusted to arrive at different transformer models. For instance, BERT only contains encoders [25]. $\text{BERT}_{\text{BASE}}$ has 12 layers in the encoder stack while $\text{BERT}_{\text{LARGE}}$ has 24 layers in the encoder stack. OpenAI GPT-3 [3] only has 12 layers of decoders.

Before the word embedding is processed by the encoder, we add the position information of the tokens to the sequence so that the model is aware of the position of the token in each sequence. We use sine and cosine functions to encode this position information [51]:

$$\mathbf{PE}_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}}), \tag{1}$$

$$\mathbf{PE}_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}}), \tag{2}$$

where $pos$ is the position of the token in the sequence; $\text{d}_{\text{model}}$ is the dimension of each word's embedding, and $i$ is the $i$-th dimension in the word embedding vector. These positional values are added to the embedding of each token.

As shown in Figure 1, self-attention is the key for an encoder. During linear transformation, the input $\mathbf{X}$ is multiplied by three weight matrices, i.e., query ($\mathbf{W_Q}$), key ($\mathbf{W_K}$), and value ($\mathbf{W_V}$), to arrive at $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, respectively. That is $\mathbf{Q} = \mathbf{X} \cdot \mathbf{W}_{\mathbf{Q}}^{\text{T}}$, $\mathbf{K} = \mathbf{X} \cdot \mathbf{W}_{\mathbf{K}}^{\text{T}}$ and $\mathbf{V} = \mathbf{X} \cdot \mathbf{W}_{\mathbf{V}}^{\text{T}}$. Afterwards, we follow Equation 3 to perform attention computation:

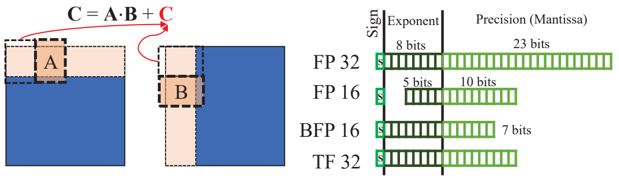
$$\mathbf{Z} = Attention(\ \mathbf{Q}, \mathbf{K},\ \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q} \cdot \mathbf{K}^{\text{T}}}{\sqrt{d_k}}) \cdot \mathbf{V}. \tag{3}$$

Note, a mask is applied to exclude certain word-to-word interactions before softmax. One popular masking mechanism is to set the lower triangle part of the masking as 0 and the upper triangle part as negative infinity. When applied to $\mathbf{Q} \cdot \mathbf{K}^{\text{T}}$, this mask actually prevents the position information of later words from affecting the earlier ones. Multi-head attention extends Equation 3. The attention model possesses multiple sets of $\mathbf{W_Q}$, $\mathbf{W_K}$, $\mathbf{W_V}$, which allows the model to jointly attend to information from different representation subspaces at different positions and each set of weights is a head. Since there are multiple $\mathbf{Z}$'s of Equation 3, multi-head attention combines them by multiplying $\mathbf{Z}$ with a weight matrix $\mathbf{W_O}$. $\sqrt{\text{d}_\text{k}}$ is the dot product's scaling factor, where $d_k = \frac{d_{\text{model}}}{\text{H}}$, and H is the number of heads.

As shown in Figure 1, after the self-attention module, there is a multilayer perceptron (MLP) module consisting of two linear transformation layers with an activation layer between them. A layer normalization is applied after self-attention and MLP respectively, whose input will be added by the input of the module.

## 2.2 GPUs and Tensor Cores

In recent GPU architectures, a Streaming Processor (SMX) often contains both general and tensor cores. Using V100S GPU as an example, one SMX contains 64 single floating-point cores, 64 integer cores, 32 double floating-point cores, and 8 tensor cores. As one tensor core can perform 64 Fused Multiply Add (FMA) operations every cycle. It implies that one SMX can perform 1,024 operations every cycle with tensor cores, or tensor core is 8× faster than the general cores. Note, when one type of core is in use, the other cores cannot execute since all these cores share certain hardware resources [34].



(a) FMA for matrix multiplication (b) Floating point variants

**Figure 2: Tensor core specification: (a) FMA for matrix multiplication and (b) floating point variants.**

With the tensor core, a large matrix multiplication problem can be accomplished by a collection of small matrix multiplication. As shown in Figure 2(a), to perform a matrix multiplication of 16×$n$ and $n$×16, the FMA can perform one 16×16 matrix multiplication of $\mathbf{A}$ and $\mathbf{B}$ at a time, and accumulate the result to an eventual result for a tensor tile of 16×16 in $\mathbf{C}$.

Tensor core comes with precision variations. First, V100S FMA operation supports mixed-precision which means the matrix multiplication is FP16 while the addition goes to FP32 [34]. As shown in Figure 2(b), the IEEE 754 FP32 format often requires 1 sign bit, 8 exponent bits, and leaving the remaining 23 bits for precision (mantissa). V100S supports the FP16 which reduces bits for both exponent and precision. The recent A100 and Google Tensor Processing Unit (TPU) also support Brain Floating Point 16 (BFP16) which cuts more bits from mantissa so that it can represent a wider value range with lower precision. The A100 GPU also support new TensorFloat-32 (TF32) type, Int8, Int4, and Binary types [37].

## 2.3 TensorRT

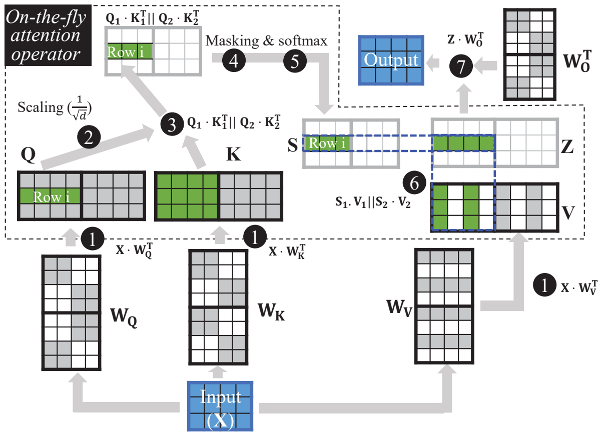
TensorRT is a high-performance neural network inference framework including a network optimizer and a runtime engine. Given a trained neural network, it optimizes the network by fusing layers, choosing the optimal kernel implementation, and reorganizing the computation based on network structure. Briefly, TensorRT performs three key optimizations to the neural network computation graph. First, it eliminates the layers that generate unused output to avoid unnecessary computation. Second, when possible, it fuses convolution, bias, ReLU layers to form a single layer. Third, if multiple layers take the same input tensor, or perform the same operations with similar parameters simultaneously, it also fuses these layers, which is referred to as horizontal layer fusion. Note that these graph optimizations do not change the underlying computations. Instead, they aim to restructure the graph to perform the operations much faster and more efficiently.

## 3 NOVEL SELF-ATTENTION ARCHITECTURE

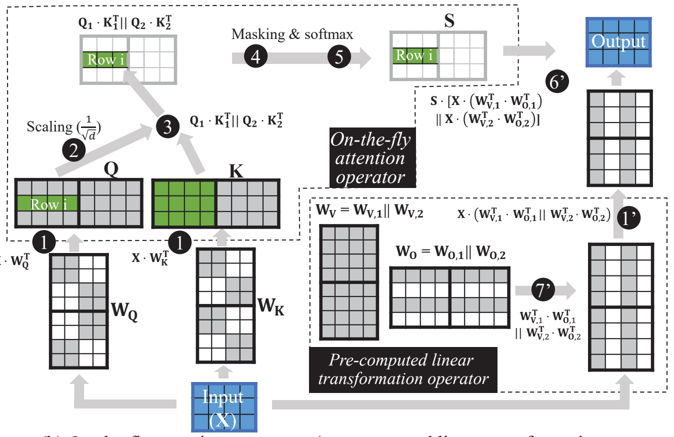
### 3.1 Tailored Self-Attention Operators

We introduce two tailored operators: (i) on-the-fly attention operator, i.e., ❷ - ❻ in Figure 3(a) and ❷ - ❺ in Figure 3(b), and (ii) pre-computed linear transformation, a.k.a., step ❼' in Figure 3(b).

**On-the-fly attention operator**. By performing attention operator on the fly, we do not need to either write or read the intermediate results, respectively, to or from the GPU global memory. Here, the intermediate results are $\mathbf{Q}_1 \cdot \mathbf{K}_1^{\text{T}} \| \mathbf{Q}_2 \cdot \mathbf{K}_2^{\text{T}}$ and $\mathbf{S}$ in Figures 3(a) and 3(b). As a result, all operations are performed in either shared



(a) On-the-fly attention operator *w/o* pre-computed linear transformation operator.


Q₁ · K₁ᵀ|| Q₂ · K₂ᵀ
Row i
Masking & softmax
4
5
S
Output
6'
S · [X · (W_{V,1}ᵀ · W_{O,1}ᵀ) || X · (W_{V,2}ᵀ · W_{O,2}ᵀ)]
Scaling (1/√d)
2
3
Q₁ · K₁ᵀ|| Q₂ · K₂ᵀ
Q
K
On-the-fly attention operator
1
X · W_Qᵀ
X · W_Kᵀ
W_Q
W_K
W_V = W_{V,1}|| W_{V,2}
X · (W_{V,1}ᵀ · W_{O,1}ᵀ|| W_{V,2}ᵀ · W_{O,2}ᵀ)
1'
W_O = W_{O,1}|| W_{O,2}
7'
W_{V,1}ᵀ · W_{O,1}ᵀ || W_{V,2}ᵀ · W_{O,2}ᵀ
Pre-computed linear transformation operator
Input (X)

(b) On-the-fly attention operator *w/* pre-computed linear transformation operator.

**Figure 3: Our self-attention architecture for a sequence of three tokens, four features per token, and two heads, each of which is a thicker border box. We assume (a) use tensor tile pruning for $\mathbf{W_Q}$, $\mathbf{W_K}$ and $\mathbf{W_O}$, row pruning for $\mathbf{W_V}$. In (b), $\mathbf{W_Q}$ and $\mathbf{W_K}$ remain tensor tile pruned while $\mathbf{W_O}$ is row pruned, and $\mathbf{W_V}$ is dense. Note, the symbol $\|$ represents the concatenation of different heads.**

memory or registers. Note, our on-the-fly attention operator is different from TensorRT that fuses various kernels into one kernel. *That is, TensorRT still has to write intermediate results to and from global memory. TensorRT kernel fusion can only avoid copying the data from the global memory of one kernel to another.*

On-the-fly attention requires to resolve the data dependency from $\mathbf{Z}$ to $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, as shown in Figure 3(a). There are two levels of data dependency, that is, head and row-level dependencies: (i) Each head of $\mathbf{Z}$ depends on the corresponding head of $\mathbf{Q}$ and $\mathbf{K}$. Therefore, we can compute each head of $\mathbf{Z}$ independently. (ii) Each row of $\mathbf{Z}$ is also independent. It is also important to note that softmax requires finding the maximum from the entire row of one head in $\mathbf{Q}_1 \cdot \mathbf{K}_1^T \| \mathbf{Q}_2 \cdot \mathbf{K}_2^T$. Hence, one row of one head in $\mathbf{Z}$ is the minimal independent unit we can derive. Since head-level dependency is straightforward, we explain the row-level dependency below.

As shown in Figure 3(a), we compute each row $i$ in $\mathbf{Q}_1 \cdot \mathbf{K}_1^T$ at a time. This row is calculated by scaling one row of $\mathbf{Q}_1$ (❷) and multiplying this scaled row with $\mathbf{K}_1^T$ (❸). We store row $i$ in shared memory for masking (❹) and softmax (❺) computations. Since a Cooperative Thread Array (CTA) is the maximum thread unit in modern GPUs that can share the same shared memory region, we schedule a CTA to work on row $i$ in $\mathbf{S}$. The threads in each CTA compute row $i$ of $\mathbf{Q}_1\mathbf{K}_1^T$ in parallel, perform a parallel reduction for softmax, and arrive at row $i$ in $\mathbf{S}$. Still in this on-the-fly attention operator, this one row of $\mathbf{S}$ loads a head from $\mathbf{V}$ and performs multiplication to derive one row of $\mathbf{Z}$ (❻). As shown in Figure 3(b), on-the-fly attention operator is slightly revised when pre-computed linear transformation operator is introduced. When deployed on tensor cores, E.T. computes a row of tiles in $\mathbf{Z}$ at a time. It is also important to note that even one CTA is responsible for 16 rows of a head at a time, recent transformer-based models which often process ≥12 heads, with ≥768 features according to [56], would have adequate workloads to saturate one V100S GPU.

**Pre-computed linear transformation operator**. For multi-head attention, we use $\mathbf{W_O}$ to combine various heads of $\mathbf{Z}$ and derive the final output matrix in step ❼ of Figure 3(a) as following:

$$\text{Output} = \mathbf{Z} \cdot \mathbf{W_O^T} = (\|_{h=1}^{H} \mathbf{Z}_h) \cdot (\|_{h=1}^{H} \mathbf{W}_{O,h}^T) = \sum_{h=1}^{H} \mathbf{Z}_h \cdot \mathbf{W}_{O,h}^T, \tag{4}$$

where we assume both $\mathbf{Z}$ and $\mathbf{W_O}$ have H heads. We use the concatenation operator $\|_{h=1}^{H}$ to concatenate them together. Intuitively, Equation 4 suggests that the concatenated multiplication, in fact, multiplies each head of $\mathbf{Z}$ with the corresponding head of $\mathbf{W_O^T}$ to arrive at the resultant matrix of that head. Afterwards, we add the resultant matrices of all heads together to arrive at the output. This is also observed in step ❼ of Figure 3(a), where Output = $\mathbf{Z}_1 \cdot \mathbf{W}_{O,1}^T + \mathbf{Z}_2 \cdot \mathbf{W}_{O,2}^T$.

Assuming $\mathbf{S} = \text{softmax}(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d_k}})$, we can obtain: $\mathbf{Z} = \mathbf{S} \cdot \mathbf{V}$, further $\mathbf{Z}_h = \mathbf{S}_h \cdot \mathbf{V}_h$ for each head h. Using this equation to replace $\mathbf{Z}_h$ in Equation 4, we get:

$$\begin{aligned} \text{Output} &= \sum_{h=1}^{H} (\mathbf{S}_h \cdot \mathbf{V}_h) \cdot \mathbf{W}_{O,h}^T = \sum_{h=1}^{H} \mathbf{S}_h \cdot (\mathbf{V}_h \cdot \mathbf{W}_{O,h}^T) \\ &= \sum_{h=1}^{H} \mathbf{S}_h \cdot (\mathbf{X} \cdot \mathbf{W}_{V,h}^T \cdot \mathbf{W}_{O,h}^T) = \sum_{h=1}^{H} \mathbf{S}_h \cdot \mathbf{X} \cdot (\mathbf{W}_{V,h}^T \cdot \mathbf{W}_{O,h}^T). \end{aligned} \tag{5}$$

The transformation in Equation 5 allows us to compute the matrix multiplication between $\mathbf{W_V}$ and $\mathbf{W_O}$ beforehand. Further, because $\mathbf{W_V}$ and $\mathbf{W_O}$ are known before the inference computation, we can pre-compute each head of $\mathbf{W}_{V,h}^T \cdot \mathbf{W}_{O,h}^T$.

Figure 3(b) shows how the pre-computed linear transformation operator works for the same example in Figure 3(a). We allow each head of $\mathbf{W_V}$ and $\mathbf{W_O}$ in ❼' to multiply and arrive at $(\mathbf{W}_{V,1}^T \cdot \mathbf{W}_{O,1}^T) \| (\mathbf{W}_{V,2}^T \cdot \mathbf{W}_{O,2}^T)$. Then, $\mathbf{X}$ multiplies with this matrix in step ❶'. Finally, step ❻' in Figure 3(b) to derive the output. In addition to the aforementioned derivation, our evaluation of real-world dataset also exhibits that this pre-computed linear transformation operator yields the same results as the original design [51].

### 3.2 Sequence Length Aware Optimization For On-the-fly Attention Operator

As the sequence length, i.e., the # of columns in one head of $\mathbf{Q}_1 \cdot \mathbf{K}_1^\mathrm{T} \| \mathbf{Q}_2 \cdot \mathbf{K}_2^\mathrm{T}$ in Figure 3(a) increases, E.T. could potentially suffer from two performance issues: (i) The intermediate result residing in shared memory becomes larger and may exceed the shared memory capacity. (ii) We need to load the entire **K** to compute each row of $\mathbf{QK}^\mathrm{T}$ at step ❸ in Figure 3(a). Similarly for step ❻. This could result in overwhelming memory access traffic.

For the first performance issue, the total needed shared memory space for a tile row has a relation with sequence length as below:

$$\underbrace{\text{tileHeight} \cdot \frac{\mathrm{d_{model}}}{\mathrm{H}}}_{\text{Tile row i of } \mathbf{Q} \text{ in Figure 3(a)}} + \underbrace{\text{tileHeight} \cdot \text{seqLen}}_{\text{Tile row i of } \mathbf{S} \text{ in Figure 3(a)}}, \quad (6)$$

where the length of a row in $\mathbf{Q}_1 \cdot \mathbf{K}_1^\mathrm{T}$ is the sequence length, i.e., seqLen, tileHeight = 16, $\mathrm{d_{model}}$ is the dimension of the word embedding, and H denotes the number of heads in the model which ranges from 2 to 128 according to [56]. If we use $\mathrm{BERT_{LARGE}}$, which features 16 heads, $\mathrm{d_{model}}$=1,024, and 384 as the sequence length, the total needed shared memory is 7KB. Since a commodity GPU like V100S usually has 96 KB shared memory per processor, our on-the-fly attention could serve a relatively long sequence length.

For the second performance concern, step ❸ of Figure 3(a), where every row of **Q** needs to access the entire head of **K** during the on-the-fly computation to avoid writing and reading one head of $\mathbf{Q}_1 \cdot \mathbf{K}_1^\mathrm{T} \| \mathbf{Q}_2 \cdot \mathbf{K}_2^\mathrm{T}$ matrix, respectively, to and from global memory. Our on-the-fly attention dissects the GEMM into the row vector of **Q** multiplying a head in $\mathbf{K}^\mathrm{T}$ so that each CTA performs the inner-product between the row vector and every column vector in the head. Thus finishing the entire resultant matrix needs **Q** to be accessed once while $\mathbf{K}^\mathrm{T}$ to be accessed multiple times. Similarly for step ❻, where we need to access the entire head of **V** for each row of **S**. As a result, on-the-fly attention would be beneficial only when **K** and **V** are relatively small which means the GPU bandwidth can afford repeatedly loading them for various rows of **Q**.

Otherwise, we adopt the outer-product-based GEMM to reduce the memory traffic when computing ❸ and ❻. Using ❸ in Figure 3(a) as an example, we can load the first column **Q**, and the first row of $\mathbf{K}^\mathrm{T}$ to compute the partial results for the entire $\mathbf{Q}_1\mathbf{K}_1^\mathrm{T}$. Afterwards, we do that for the follow-up columns of $\mathbf{Q}_1$, and rows of $\mathbf{K}_1^\mathrm{T}$. Once each column of $\mathbf{Q}_1$ has multiplied with the corresponding row of $\mathbf{K}_1$, we arrive at the $\mathbf{Q}_1 \cdot \mathbf{K}_1^\mathrm{T}$. This design only loads $\mathbf{Q}_1$ and $\mathbf{K}_1$ once. However, we need to schedule the entire GPU to perform this computation, writing the results into global memory, performing global synchronization, then loading one row of $\mathbf{Q}_1 \cdot \mathbf{K}_1^\mathrm{T}$ in shared memory for the follow-up masking, softmax, and eventual multiplication with **V**, similarly for $\mathbf{Q}_2 \cdot \mathbf{K}_2^\mathrm{T}$.

We find sequence length is the key factor that decides whether we should adopt outer-product-based GEMM, hence break ❸ and ❻ from our on-the-fly attention. Therefore, an adaptive solution that switches between on-the-fly and breaking ❸ and ❻ from on-the-fly is needed. In the evaluation section, we will present a thorough study about this design exploration.

### 3.3 Reordered Self-Attention Architecture For Pure FP16-based Attention Computation

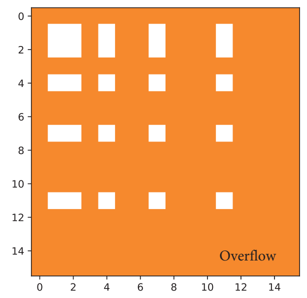


**Figure 4: The heatmap of overflow problem faced by attention mechanism when we use tensor core to compute $\mathbf{Q}_1 \cdot \mathbf{K}_1^\mathrm{T} \| \mathbf{Q}_2 \cdot \mathbf{K}_2^\mathrm{T}$, where the orange shadow colors are the overflow entries. Here, we use Transformer on WiKiText-2 with sequence length = 16 and dimension of word embedding as 256.**

We observe that the result of multiplying one tile of row *i* in **Q** with one tile in **K** goes beyond the value range of FP16. Figure 4 exhibits the overflow problem faced by pure FP16 based $\mathbf{Q}_1 \cdot \mathbf{K}_1^\mathrm{T} \| \mathbf{Q}_2 \cdot \mathbf{K}_2^\mathrm{T}$. Clearly, one can observe that the majority of the entries are shadow ones which mean they face overflow problems.

In this context, one needs to use a mixed precision-based tensor operation. Mixed precision implies that each row *i* of $\mathbf{Q}_1 \cdot \mathbf{K}_1^\mathrm{T} \| \mathbf{Q}_2 \cdot \mathbf{K}_2^\mathrm{T}$ has to be stored in FP32, and converted back to FP16 for the subsequent tensor core operations. Mixed precision introduces two aspects of overhead when compared to pure FP16-based attention: (i) double the shared memory consumption when storing row *i* of $\mathbf{Q}_1 \cdot \mathbf{K}_1^\mathrm{T} \| \mathbf{Q}_2 \cdot \mathbf{K}_2^\mathrm{T}$ ; (ii) converting FP32 back to FP16 for subsequent masking and softmax computations.

To achieve pure FP16-based attention computation, we move the scaling operation, i.e., ❷ in Figure 3(a) and 3(b), before the matrix multiplication operation ❸. Note, as shown in Figure 1, scaling operation is performed after $\mathbf{Q}_1 \cdot \mathbf{K}_1^\mathrm{T} \| \mathbf{Q}_2 \cdot \mathbf{K}_2^\mathrm{T}$ is obtained in the state of the art projects, such as PyTorch [42] and TensorRT [40]. since each row of $\mathbf{Q}_1 \cdot \mathbf{K}_1^\mathrm{T} \| \mathbf{Q}_2 \cdot \mathbf{K}_2^\mathrm{T}$ will be converted back to FP16 for subsequent operations, this implies that our reordering guarantees pure FP16 will not experience overflow problem during attention computation. It is also important to note that changing the location of scaling operation and optimizing softmax accuracy only means avoiding the overflow errors, which suggests that our reordering yields the same results for the design without such an adjustment.

## 4 ATTENTION-AWARE MODEL PRUNING

### 4.1 Transforming Existing Pruning Designs into Tensor Core Friendly Formats

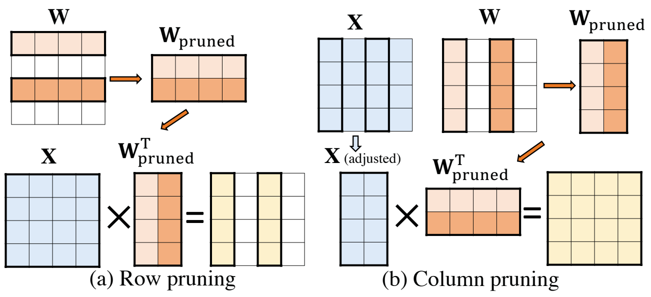


**Figure 5: Examples of a linear transformation with row/column pruned weight matrix. *X* is the input embedding matrix.**

**Condensing row/column pruning for the tensor core.** After row or column pruning [21] of the weight matrix $\mathbf{W}$, we propose to remove the pruned rows or columns, and condense the existing nonzero rows or columns into a new dense weight matrix $\mathbf{W}_{\text{pruned}}$. Here, $\mathbf{W}$ could represent $\mathbf{W_Q}$, $\mathbf{W_K}$, $\mathbf{W_V}$, and $\mathbf{W_O}$. Subsequently, we can resort to a tensor core-based highly optimized GEMM routine for rapid matrix multiplication. We further find that row and column pruning generate different weight matrices that could affect the subsequent attention computation. This leads to our attention-aware pruning (Section 4.3).

**Row pruning**. Figure 5(a) exemplifies the row pruning design. Assuming the second and last rows are pruned in $\mathbf{W}$. We condense the pruned $\mathbf{W}$ into a 2 by 4 dense $\mathbf{W}_{\text{pruned}}$ matrix. Finally, we multiply the input $\mathbf{X}$ with the pruned and transposed weight matrix $\mathbf{W}^{\text{T}}_{\text{pruned}}$ to arrive at the resultant matrix with two nonzero columns. Here, since both $\mathbf{X}$ and $\mathbf{W}^{\text{T}}_{\text{pruned}}$ are in dense format, one can use tensor core to accelerate the GEMM. **Column pruning** adopts a similar idea to enable tensor core-based GEMM but the procedure is slightly different. As shown in Figure 5(b), the weight matrix is column pruning, then transposed and eventually multiplied with the input matrix $\mathbf{X}$. Because only the first and third columns have nonzero entries in $\mathbf{W}_{\text{pruned}}$, only the first and third columns of $\mathbf{X}$ will have nonzero entries to multiply with. Therefore, one can use $\mathbf{X}_{\text{adjusted}}$ to multiply with $\mathbf{W}^{\text{T}}_{\text{pruned}}$.

**Irregular pruning**, which does not yield promising time consumption saving, is included here for completeness. Particularly, irregular pruning produces a weight matrix with nonzero entries scattered at random locations in the weight matrix. We adopt a hierarchical sparse format from a recent study [59] to implement tensor core-based irregular pruned linear transformation. This format contains two levels of sparsity: (i) a bitmap-based format to store each tensor tile in the weight matrix that contains at least one nonzero. (ii) a Block Compressed Sparse Row or Column (BCSR or BCSC) format to store various nonzero tensor tiles.

## 4.2 Tensor Tile Pruning for Transformer Models

Despite row/column/irregular pruned weight matrix can be transformed into formats leveraging tensor cores, one needs to pay nontrivial overheads on pre-processing the inputs, and post-processing the resultant matrix. This section introduces a tensor tile-based pruning algorithm, which either excludes the entire tensor tile of weights or keeps the entire tensor tile intact within weight matrices. Apparently, this tensor tile-based pruning introduces significantly better storage and control-flow logic compared to the aforementioned approaches in Section 4.1. Below, we formulate this pruning design into an algorithm that is similar to reweighted $l_1$ method [4].

Consider an $N$-layer Transformer, where the collection of weights and bias in the $k$th layer are denoted by $\mathbf{W}^k$ and $\mathbf{b}^k$, respectively. We denote the tile row as $r$, tile column as $c$. An original weight matrix $\mathbf{W} \in \mathbb{R}^{m\times n}$ is divided into $p \times q$ tiles, where $p = \frac{m}{r}$, and $q = \frac{n}{c}$. In weight pruning, our objective is to prune the weights. Therefore, we minimize the loss function subject to constraints of different weight distribution in each layer. Formally, this tensor tile-based weight pruning can be written as

$$\begin{aligned} &\min \quad f\big(\{\mathbf{W}^k\}_{k=1}^N, \{\mathbf{b}^k\}_{k=1}^N)\big\}\big), \\ &\text{subject to} \quad \#\text{ of non-zero tiles in } \mathbf{W}^k \text{ is less than } l^k, \end{aligned} \tag{7}$$

where $l^k$ is the desired numbers of non-zero tiles. The loss function is denoted by $f\big(\{\mathbf{W}^k\}_{k=1}^N, \{\mathbf{b}^k\}_{k=1}^N\big)$. To mitigate the accuracy loss caused by directly pruning the weights, we relax the hard constraints by adding regularization terms in the objective function. When we use group Lasso, the regularization term is $\sum_{k=1}^N \sum_{i=1}^p \sum_{k=1}^q \beta_{ij}^k \|\mathbf{W}_{ij}^k\|_2$, where $p$ and $q$ are the number of (tile-wise) rows and columns in the $k$th layer, respectively. Together, we rewrite the weight pruning problem from Equation 7 as:

$$\min \quad f\big(\{\mathbf{W}^k\}_{k=1}^N, \{\mathbf{b}^k\}_{k=1}^N\big) + \lambda \sum_{k=1}^{N} \sum_{i=1}^{p} \sum_{j=1}^{q} \beta_{ij}^k \|\mathbf{W}_{ij}^k\|_2, \tag{8}$$

where $\lambda$ is the weight penalty factor for regularization strength.

We illustrate the training process using an $8 \times 4$ weight matrix (two heads; each with $4 \times 4$) as shown in Figure 6. The training has the following steps. (i) Initialization. We start with a pre-trained model. (ii) We check if the current epoch falls into the pre-defined milestones (i.e., the $m_1, m_2, \cdots, m_s^{\text{th}}$ epoch). If yes, we select the tile size $r\times c$ as $2\times 2$ and divide the weight matrix $m\times n$ ($8\times 4$) into $p\times q$ ($4 \times 2$) sub-matrices. We compute the $l_2$-norm of each tile $\|\mathbf{W}_{ij}^k\|_2$ and update the tile penalty factor $\beta_{ij}^k$ using $\beta_{ij}^k = 1/(\|\mathbf{W}_{ij}^{k-1}\|_2 + \epsilon)$, where $\epsilon$ is a small value preventing division by zero. (iii) We update the model loss as indicated by Equation 8. Note that in this step we treat $\lambda$ and $\beta_{ij}^k$ as constant. (iv) We train the Transformer model and update the parameters. We stop increasing $\lambda$ when the reweighted training accuracy drops slightly. As a result, a well-trained set of parameters will be generated. (v) We perform weight pruning based on $l_2$ norm. ❶ We first divide the weight matrix $8\times 4$ into 8 $2\times 2$ sub-matrices. ❷ We compute $l_2$ norm of the sub-matrices to produce a $4 \times 2$ norm matrix. We will select the values with the largest group $l_2$ norm in a tile. If the value is less than pre-set percentile (say 50%), we set the value to 0. ❸ We generate a pruning mask matrix ($8 \times 4$, all 0s, and 1s) for the $p \times q$ tiles. ❹ We multiply the mask matrix with the original weight matrix element-wisely to obtain the tensor tile pruned weight matrix. Finally (vi), we retrain the non-zero entries for several epochs (e.g., 4), to recover the accuracy.

## 4.3 Novel Attention-aware Adaptive Pruning

Since irregular pruning introduces tremendous overhead, only row, column, and tensor tile-based pruning algorithms can bring performance gains. However, the row pruning of either matrix $\mathbf{Q}$ or $\mathbf{K}$ degrades the prediction accuracy severely. The root cause is that the attention neural network could be interpreted as a retrieval process, where the row vectors in $\mathbf{Q}$ and $\mathbf{K}$ are considered as queries and keys [51]. The dot-product of query and key is computing their distance in the feature space. Therefore, removing an entire row means reducing the size of input data. This significantly affects the captured information from the model. As a result, we only have column and tile pruning for $\mathbf{W_Q}$ and $\mathbf{W_K}$. For $\mathbf{W_O}$, row, column, and tile pruning are possible candidates.

Now, we consider $\mathbf{W_Q}$ and $\mathbf{W_K}$. On the one hand, because column pruning either of them will lead to a dense matrix after $\mathbf{X}$

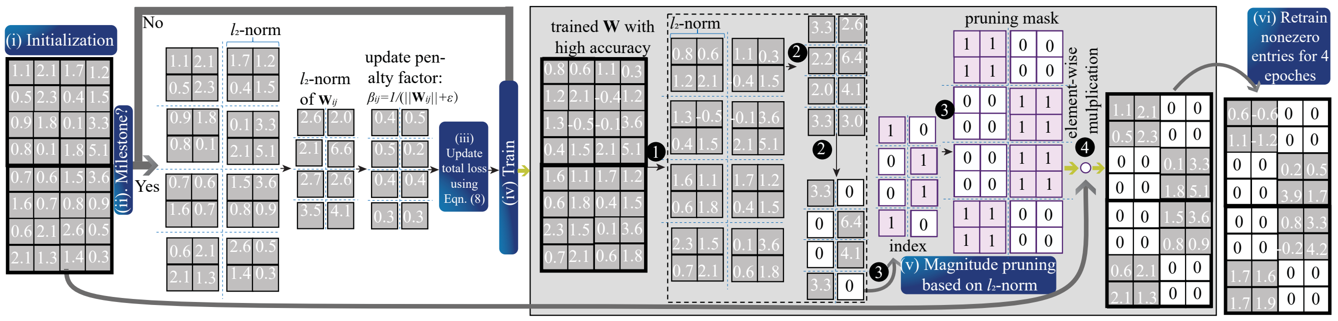

(i) Initialization
No
(ii). Milestone?
Yes
$l_2$-norm
$l_2$-norm of $\mathbf{W}_{ij}$
update penalty factor: $\beta_{ij}=1/(||\mathbf{W}_{ij}||+\epsilon)$
(iii) Update total loss using Eqn. (8)
(iv) Train
trained W with high accuracy
pruning mask
index
element-wise mulplication
(v) Magnitude pruning based on $l_2$-norm
(vi) Retrain nonezero entries for 4 epochs

Figure 6: Re-weighted training, pruning, and masked retraining; we use an $8 \times 4$ weight matrix (two head; each with $4 \times 4$) as an example.

multiplies with the transpose of either $\mathbf{W_Q}$ or $\mathbf{W_K}$. Therefore, one cannot enjoy pruned resultant matrix for future operations. On the other hand, tensor tile pruned matrix can avoid both pre- and post-processing overhead that is suffered by column pruning. As a result, we use tensor tile-based pruning for $\mathbf{W_Q}$ and $\mathbf{W_K}$.

For $\mathbf{W_V}$ and $\mathbf{W_O}$, the preferred pruning patterns are highly dependent upon whether we adopt pre-computed linear transformation or not. If we do not adopt pre-computed linear transformation, which is shown in Figure 3(a), we prefer column pruned $\mathbf{W_V}$, and tensor tile pruned $\mathbf{W_O}$. Otherwise, since we pre-compute the multiplication between $\mathbf{W_V}$ and $\mathbf{W_O}$, we care more about the pruning pattern in the resultant matrix ($\|_{h=1}^{H} \mathbf{W}_{\mathbf{V},h}^{T} \cdot \mathbf{W}_{\mathbf{O},h}^{T}$). As shown in Figure 3(b), we adopt row pruning for $\mathbf{W_O}$ while leave $\mathbf{W_V}$ as dense matrix for two reasons. First, $\mathbf{W_O}$ is row pruned because the resultant matrix $\mathbf{X} \cdot (\|_{h=1}^{H} \mathbf{W}_{\mathbf{V},h}^{T} \cdot \mathbf{W}_{\mathbf{O},h}^{T})$ will remain column pruned that can benefit step ❻ in Figure 3(b). Second, $\mathbf{W_V}$ is not pruned because pruning $\mathbf{W_V}$ will not only lead to no sparsity changes in $\mathbf{X} \cdot (\|_{h=1}^{H} \mathbf{W}_{\mathbf{V},h}^{T} \cdot \mathbf{W}_{\mathbf{O},h}^{T})$ but also prevent us from pruning more entries in the other weight matrices. We abbreviate the method as *attention-aware pruning* in the following context.

## 5 EXPERIMENTS

### 5.1 Experimental Setup

**Datasets.** For the Transformer model, we conduct experiments on the Wikitext-2 dataset [30]. For BERT$_{\text{BASE}}$ [11] and DistilBERT [48] models, we conduct experiments on GLUE benchmark [52], a comprehensive collection of natural language understanding tasks covering three NLP categories, i.e., inference tasks (MNLI [55], Quora Question Pairs (QQP) [62], QNLI [52] (a set of over 100,000+ question-answer pairs from SQuAD [45]), single-sentence (SST-2 [49]), paraphrase similarity matching (STS-B [5], Microsoft Research Paraphrase Corpus (MRPC [12]) and WNLI [26]).

**Baseline models.** Our baseline models are unpruned Transformer, BERT$_{\text{BASE}}$, and DistilBERT. We list the reported prediction accuracy of BERT$_{\text{BASE}}$ and DistilBERT from the original papers in the first row as shown in Table 1. For BERT, we use the official BERT$_{\text{BASE}}$ [11], uncased model as our pre-trained model. There are 12 encoder layers (L =12; embedding dimension $d_{\text{model}}$ = 768; self-attention heads H = 12), with total number of parameters 110 Million. For DistilBERT, there are 6 encoder layers (L =6; embedding dimension $d_{\text{model}}$ = 768; self-attention heads H = 12). For Transformer, there are 2 encoder layers (L =2; embedding dimension $d_{\text{model}}$ = 800; self-attention heads $H$ = 4).

**Evaluation metrics.** We measure the latency of inference for different models and sparsities. It takes word embeddings as the input and generates the output for task-specific post-process. For pruning algorithms, on WikiText-2, we use the accuracy of next word prediction. On all GLUE benchmarks, we report the metrics following the conventions in [52], i.e., accuracy scores are reported for MNLI, SST-2, QNLI and WNLI; F1 scores are reported for QQP, MRPC; Spearman correlations are reported for STS-B.

**Implementation details**. We implement E.T. from scratch instead of using TensorRT [40]. Although TensorRT provides the API for developers to add customized kernels as plugins, we find doing that would disrupt TensorRT from optimizing the entire implementation. Further, the optimizer of TensorRT works on the operator level as opposed to the source code level. It can only fuse the built-in operators by searching from the pre-defined rules and implementations. Besides, we have to pay the overhead of wrapping our customized code into TensorRT, as well as maintaining the compatibility between our kernel and the TensorRT ones.

Regarding pruning, for Transformer, we first train a model from scratch for 50 epochs, and the resulting trained model is used as our pre-trained model. We load the pre-trained model, run 50 epochs reweighted-based training, subsequently another 50 epochs retraining after applying pruning on the previously trained models. For BERT, we use the official BERT$_{\text{BASE}}$, uncased model as our pre-trained model and we set $\lambda$ as 0.0001. While a distilled model from the BERT, uncased checkpoint, is used as the pre-trained model for DistilBERT [48]. We set $\lambda$ as 0.0001 for MNLI and 0.0003 on other tasks for DistilBERT. We train the models for the downstream GLUE tasks with their corresponding datasets. As we feed the data, the entire pre-trained model and the additional untrained classification layer are trained on our specific task. For fine-tuning, we run 4 epochs with a learning rate from the range between 0.00003 and 0.00005 (learning rate which gives best performance is selected for each task) and batch size of 32.

**Evaluation platforms.** Training and Evaluation are performed on Python 3.6.10 and CUDA 11.1 on V100S GPU and Intel(R) Xeon(R) Gold 6244 @ 3.60GHz CPU. The model is trained with PyTorch 1.4.0, and the source code of our implementation is compiled with NVIDIA nvcc 11.1 and GCC 7.5.0 with the optimization flag of O3.

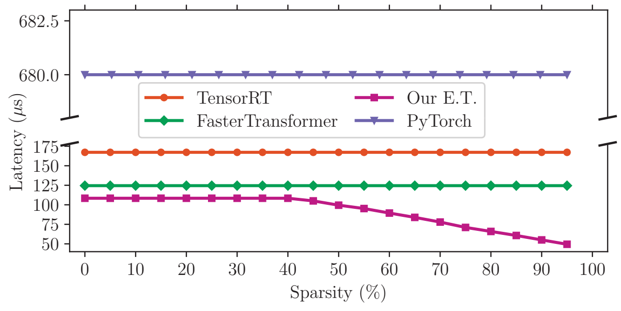


Figure 7: The latency of one encoder layer in $BERT_{BASE}$ model with the input sequence length = 128.

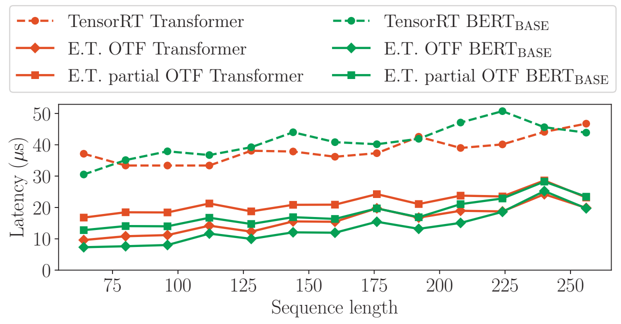


**Figure 8: The performance of different attention implementations, where "OTF" stands for on-the-fly, and "partial OTF" means we break ❷ - ❻ of Figure 3(a) into two kernels, i.e., ❷ - ❸ as one kernel and ❹ - ❻ as another kernel.**

## 5.2 Turnaround Time Study

*5.2.1 E.T. vs. state-of-the-art.* Figure 7 shows the latency of our implementation of an encoder vs the state-of-the-art including TensorRT [40], FasterTransformer [39] and PyTorch [42]. For TensorRT, we use the implementation of the encoder in the BERT demo of TensorRT [40]. For FasterTransformer, we use the sample code of C++ API provided with the latest version 3.1. Particularly, we compare the latency of one encoder layer with $d_{model}$ of 768 and 12 heads, which is the configuration of $BERT_{BASE}$ and DistilBERT. Since E.T. can automatically search through various linear transformation implementations and choose the optimal one (similar to FasterTransformer), E.T. finds and uses the best CUBLAS GEMM routine, i.e., algorithm CUBLAS_GEMM_ALGO5_TENSOR_OP (on our server) [38] when the sparsity is below 40% while attention-aware pruning afterwards. Overall, we observe that E.T. outperforms PyTorch, TensorRT, and FasterTransformer across all sparsity cases. Notably, as the prune ratio increases, the maximum speedup climbs to 13.7×, 3.4× and 2.5× over PyTorch, TensorRT, and FasterTransformer, respectively.

*5.2.2 On-the-fly attention operator vs TensorRT attention.* We compare the performance of attention computation, i.e., steps ❷ - ❻ in Figure 3(a), between our implementations and the corresponding BERT plugin in TensorRT [40] in Figure 8. Note, we attempt to compare against FasterTransformer regarding on-the-fly attention operator but found breaking the encapsulation of FasterTransformer, i.e., inserting cudaDeviceSynchronize() to FasterTransformer kernels for timing, would end up with worse turnaround time than TensorRT. This leads us to use TensorRT for this comparison.

As shown in Figure 8, either our on-the-fly attention or partial on-the-fly attention operator would best TensorRT across all cases. Particularly, for the sequence length between 64 to 256, our average speedup on Transformer is 2.5×, and 3.3× on $BERT_{BASE}$. For the full on-the-fly vs partial on-the-fly, when the sequence length is short, the full on-the-fly attention has 1.5× speedup in $BERT_{BASE}$ and 1.4× speedup in Transformer on average. But when the sequence length goes beyond 224, our partial on-the-fly attention starts to outperform. Therefore, E.T. will adapt the partial on-the-fly attention when sequence length is larger than 224.

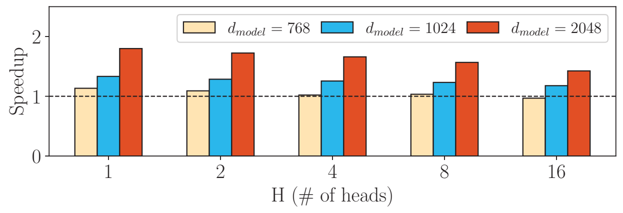
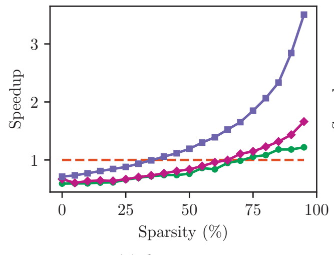


**Figure 9: The speedup of with pre-computed linear transformation over without pre-computed linear transformation for one encoder.**

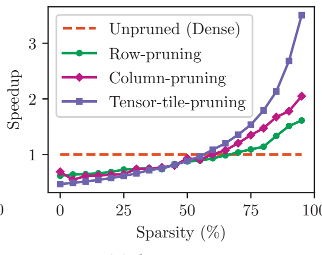
*5.2.3 Performance impacts of pre-computed linear transformation.* Figure 9 shows the performance of an encoder changing with the number of heads with sequence length as 128 and $d_{model}$ = 768, 1024, and 2048. Here, we use DistilBERT on the MRPC dataset, where the prune ratio is 50% w/o pre-computed and 80% with pre-computed linear transformation. The computation saving together with a higher prune ratio in the pre-computed linear transformation yields better performances virtually across all cases. On average, the speedup is 1.1×, 1.3×, and 1.6× respectively for $d_{model}$ = 768, 1024 and 2048, respectively. Note, a larger $d_{model}$ leads to more speedup because the computation saving is positively proportional to the model size. This trend suggests that our pre-computed linear transformation would offer more benefits when the model becomes larger, which is also the trend in recent transformer-based models.

*5.2.4 Comparison of various pruning algorithms.* Figure 10 shows the performance of different pruning algorithms in the linear transformation with $d_{model} = 768$ and $d_{model} = 1,024$ and the input sequence length as 128. We use the fastest CUBLAS GEMM configuration on our server, i.e., CUBLAS_GEMM_ALGO5_TENSOR_OP [38] as the unpruned baseline. When sparsity reaches 95%, our tensor tile pruned linear transformation enjoys 3.5×, 3.2× speedups, respectively for $d_{model} = 768$ and $d_{model} = 1,024$. The row and column pruned linear transformation has 1.6× and 1.7× maximum speedups when $d_{model} = 1,024$, respectively, while 1.2× and 1.6× maximum speedups when $d_{model} = 768$, respectively. We observe that the pruned linear transformer performance is determined by its sparsity and pruning method. As the results shown in Figure 10, when sparsity is the same, tensor tile pruning has better performance than column pruning.

*5.2.5 Hardware profiling.* Figure 11 shows the profiling result via nvprof with 128 as sequence length and $BERT_{BASE}$ configuration. Figures 11(a) and 11(b) show the memory traffic of accessing global

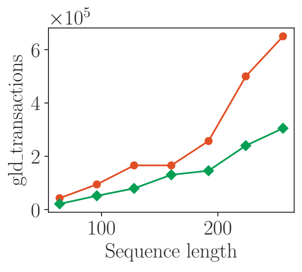


(a) $d_{model}$=768

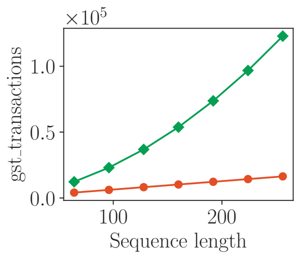


(b) $d_{model}$=1,024

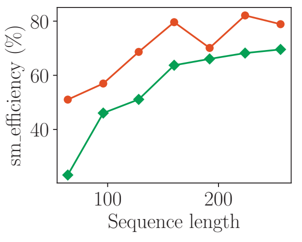
**Figure 10: The performance of different pruning algorithms of the linear layers in the encoder.**



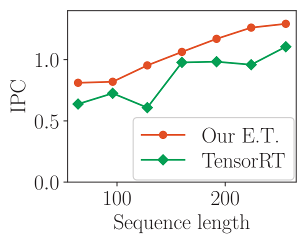
**(a) gld_transaction**



**(b) gst_transaction**



**(c) SM_efficiency**



**(d) IPC**

**Figure 11: The hardware profiling results of our on-the-fly attention operator vs. the TensorRT counterpart.**

memory using nvprof, where *gld_transactions* measures the number of global memory loading transactions and *gst_transactions* measures the number of global memory storing transactions. Our implementation loads 1.8×, on average, more memory compared to TensorRT. The good news is that our saving on storing the intermediate results to global memory is strikingly more significant, i.e., 5× on average.

Other than the contribution of reducing global memory traffic, our CTA-based workload decomposition and kernel fusing expose more parallelism so the utilization of GPU is higher. Figure 11(c) shows the average ratio of time when there is at least one warp residing on the multiprocessors measured by *sm_efficiency*. Figure 11(d) shows the instruction-per-cycle (IPC). On average, we enjoy 30% and 22% boost over baseline on *sm_efficiency* and *IPC*, respectively. The improved resource utilization indicates that the benefit of on-the-fly attention (i.e., reducing store transactions) outweighs the overhead (i.e., more load transactions). The intrinsic reason is that extra load does not saturate the memory bandwidth of GPU while storing data after each kernel execution hurts the hardware utilization because it is on the critical path. On-the-fly attention loads more data to reduce the latency of each inference execution which determines the performance in this scenario.

**Table 1: Comparison of prediction accuracy and inference time of different models on GLUE benchmark. We test the irregular pruning method [23] on $BERT_{BASE}$ and DistilBERT report the accuracy.**

| **$BERT_{BASE}$** | MNLI | QQP | QNLI | SST-2 | STS-B | MRPC | WNLI | **AVG.** |
|---|---|---|---|---|---|---|---|---|
| Baseline [11] | 84.6 | 91.2 | 90.5 | 93.5 | 85.8 | 88.9 | 56.3 | **84.4** |
| $BERT_{BASE}$ (ours) | 84.3 | 91.4 | 91.6 | 92.8 | 89.1 | 89.4 | 56.3 | **85.0** |
| Irregular [23]$^1$ | 81.3 | 85.7 | 87.9 | 89.4 | 87.1 | 84.8 | 56.3 | **81.8** |
| Pruning ratio (%) | 70% | 90% | 70% | 70% | 60% | 70% | 90% | **74.3%** |
| Latency (*ms*) | 51.7 | 17.4 | 47.2 | 47.2 | 78.1 | 47.2 | 17.4 | **43.8** |
| Column (ours) | 80.3 | 86 | 81.4 | 85.9 | 85.3 | 86.8 | 56.3 | **80.3** |
| Pruning ratio (%) | 30% | 50% | 40% | 30% | 20% | 10% | 90% | **38.6%** |
| Latency (*ms*) | 2.10 | 1.97 | 2.04 | 2.09 | 2.28 | 2.50 | 1.57 | **2.08** |
| Tensor tile (ours) | 82.9 | 86.2 | 83.3 | 84.1 | 85.3 | 86.9 | 56.3 | **80.7** |
| Pruning ratio (%) | 30% | 50% | 40% | 50% | 30% | 20% | 90% | **44.3%** |
| Latency (*ms*) | 1.43 | 1.20 | 1.44 | 1.3 | 1.45 | 1.56 | 0.69 | **1.30** |
| Attention-aware (ours) | 81.8 | 86.4 | 84.6 | 84.2 | 85.2 | 86.7 | 56.3 | **80.7** |
| Pruning ratio (%) | 30% | 80% | 40% | 70% | 30% | 20% | 90% | **51.4%** |
| Latency (*ms*) | 1.38 | 0.78 | 1.25 | 0.90 | 1.38 | 1.46 | 0.67 | **1.12** |
| **DistilBERT** | MNLI | QQP | QNLI | SST-2 | STS-B | MRPC | WNLI | **AVG.** |
| Baseline [48] | 82.2 | 88.5 | 89.2 | 91.3 | 86.9 | 87.5 | 56.3 | **83.1** |
| DistilBERT (ours) | 81.9 | 90.2 | 89 | 90.7 | 86.5 | 89.5 | 56.3 | **83.4** |
| Irregular [23]$^1$ | 80.3 | 83.9 | 84.2 | 85.4 | 84.1 | 81.3 | 56.3 | **79.4** |
| Pruning ratio (%) | 40% | 80% | 80% | 80% | 60% | 70% | 90% | **71.4%** |
| Latency (*ms*) | 44.4 | 16.1 | 16.2 | 16.2 | 38.4 | 23.5 | 8.7 | **23.4** |
| Column (ours) | 76.9 | 83.9 | 84.1 | 83.4 | 78.1 | 80.5 | 56.3 | **77.6** |
| Pruning ratio (%) | 40% | 40% | 30% | 50% | 20% | 40% | 90% | **44.3%** |
| Latency (*ms*) | 1.03 | 1.03 | 1.06 | 1.04 | 1.10 | 1.12 | 0.79 | **1.02** |
| Tensor tile (ours) | 77.3 | 84.7 | 83.7 | 82.6 | 84.5 | 80.6 | 56.3 | **78.5** |
| Pruning ratio (%) | 40 % | 40% | 30% | 60% | 20% | 50% | 90% | **47.1%** |
| Latency (*ms*) | 0.69 | 0.66 | 0.72 | 0.60 | 0.77 | 0.60 | 0.35 | **0.63** |
| Attention-aware (ours) | 78.4 | 85.1 | 83.6 | 82.6 | 84.3 | 81.2 | 56.3 | **78.8** |
| Pruning ratio (%) | 40% | 40% | 30% | 90% | 20% | 90% | 90% | **57.1%** |
| Latency (*ms*) | 0.62 | 0.62 | 0.69 | 0.33 | 0.74 | 0.33 | 0.33 | **0.53** |

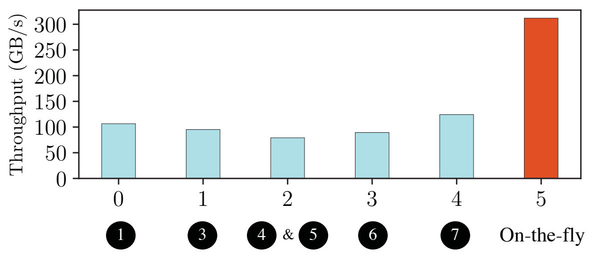


**Figure 12: The achieved memory throughput of various steps in self-attention by TensorRT. Note, steps ❶-❼ are the steps from Figure 3(a). ❷, a scaling operator, is not reported.**

*5.2.6 Memory throughput of TensorRT operators.* Figure 12 shows the achieved memory throughput of various steps in TensorRT. The average achieved memory throughput is 98 GB/s, which is only 8.6% of the peak memory bandwidth of V100S at 1,134 GB/s. In contrast, our on-the-fly attention achieves significantly higher memory throughput, i.e., 311 GB/s or 27.5% of the peak memory bandwidth. Note, we report memory throughput for GEMM, softmax and masking operators because these operators are memory bound operators. Particularly, according to [36], when arithmetic intensity is lower than 138, the operator is memory bound. For operators ❶ - ❼, the highest arithmetic intensity is 128 of ❶.

## 5.3 Evaluation of Pruning Algorithm

We evaluate and compare 4 different pruning methods of BERT and DistilBERT on GLUE benchmarks: (i) attention-aware pruning: row pruned for $\mathbf{W_V}$ on all encoder layers and tensor tile pruned for other weights; (ii) irregular pruning for all weights; (iii) column pruning for all weights, and (iv) tensor tile pruning for all weights.

*5.3.1 BERT$_{\text{BASE}}$.* Experimental results for BERT are shown in Table 1. There are two baselines, i.e., one from [11] and one our fine-tuned model. We retain 95% prediction accuracy/score on average on attention-aware pruning, tensor tile pruning, and irregular pruning compared to the baseline model BERT$_{\text{BASE}}$ (ours). Our irregular pruning achieves higher prediction accuracy/score [7] while maintaining higher pruning ratio on average. On the WNLI task, there is no accuracy loss even when the pruning ratio reaches 90% on our proposed and other pruning methods. On MNLI, STS-B, MRPC tasks, the accuracy degradation of attention-aware pruning is within 4.5%. Irregular pruning has a much higher latency (39.1× on average) even with a higher pruning ratio) than attention-aware pruning, with similar accuracy, therefore is not hardware-friendly.

Attention-aware pruning outperforms both column pruning and tensor tile pruning. It has a 0.56% increase on the average score than the column pruning while gaining a 24.9% more average pruning ratio. It also has a 13.8% improvement in prune ratio than tensor tile pruning, while achieving almost the same accuracy on average. E.T. allows user-controlled pruning ratio and accuracy loss as shown in Figure 14. Overall, there is a trade-off between pruning ratio and accuracy. On average, 5% loss of accuracy is lower or similar to current state-of-the-arts on BERT compression on GLUE benchmark [15, 41, 43, 58]. Our reported average model accuracy after pruning (96%) is higher or the same compared with these works. For instance, work [43] achieves the only 90% of full model accuracy. In [41], the compressed models only retain 81.3%, 77.3%, 94.6% of full model accuracy using different techniques. [58] achieves 89.8%, and 93.6% of full model accuracy. [15, 58] retain 95.5% of full model accuracy.

*5.3.2 DistilBERT.* Two baselines are also adopted for DistilBERT. The first one is [48] and the other one is our fine-tuned model. We retain around 94.5% prediction accuracy/score on average on attention-aware pruning, tensor tile pruning, and irregular pruning compared to the baseline model DistilBERT (ours). Same as BERT$_{\text{BASE}}$, there is no accuracy loss even when the pruning ratio reaches 90% for all pruning methods on the WNLI task. On SST-2 and MRPC, our attention-aware pruning achieves a higher pruning ratio than irregular pruning. Especially on MRPC, attention-aware pruning has a higher pruning ratio (90% vs. 70%) under the same f1 score. The latency advantage is also significant (44.2× on average) as shown in Table 1.

Attention-aware pruning outperforms both column pruning and tensor tile pruning. It has a 1.5% increase on the average score compared to column pruning, while the average pruning ratio is 22.5% higher. It also has a 17.5% higher average score than tensor tile pruning. On MNLI, STS-B tasks, more than 95.7% of the original score is retained for attention-aware pruning and we achieve a 94.4% average score compared to our fine-tuned baseline.

*5.3.3 Impacts of pruning on turnaround time.* In Table 1, attention-aware pruning has lower latency among the four pruning methods presented. In BERT$_{\text{BASE}}$, attention-aware pruning has the maximum speedup over our other pruning methods in the dataset QNLI, where the speedup over column pruning is 2.73× and 1.53× over

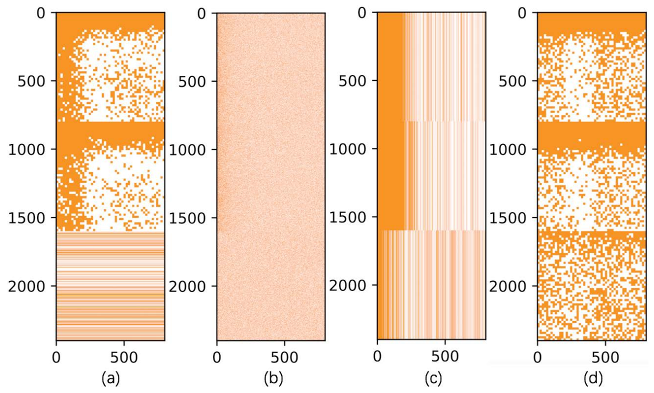
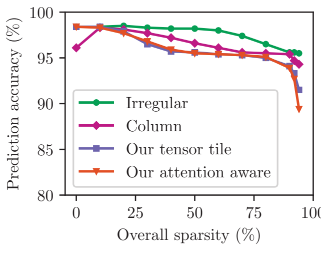


**Figure 13: The masks of in_proj_weight of Transformer after applying 4 different pruning methods (a) Attention-aware pruning: $W_V$ row pruning and others tensor tile pruning (b) Irregular pruning (c) Column pruning and (d) Tensor tile pruning. The size of in_proj_weight is 2,400×800, which is divided into three 800×800 matrices, $W_Q$, $W_K$, $W_V$ from top to bottom.**

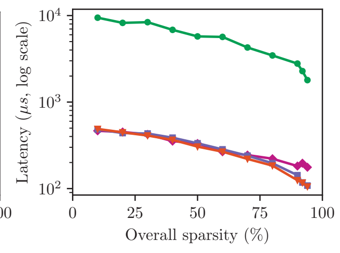
tensor tile pruning. We notice that attention-aware pruning can further increase sparsity and allow self-attention to benefit from sparsity as well. Therefore, attention-aware pruning has a better performance compared to tensor tile pruning, and column pruning when their sparsity is the same. For example, in dataset QNLI, our column pruning, tensor-tile pruning and attention-aware pruning all have a pruning ratio of 40% while attention-aware pruning has 1.59× speedup over column pruning and 1.15× speedup compared to tensor tile pruning. It has an average speedup of 1.15× over tensor tile pruning and 1.84× over column pruning. We observe that attention-aware pruning has a similar performance advantage in DistilBERT and Transformer as well. It has average speedup of 1.18× over tensor tile pruning and 1.98× over column pruning in DistilBERT. Overall, attention-aware pruning achieves the best accuracy/score than other pruning methods. It has the best turnaround time with the help of the utilization of sparsity in self-attention and tensor tile pruning for other linear layers.

*5.3.4 Transformer.* For each pruning method, we set a group of pruning ratios and report the corresponding prediction accuracy in Figure 14(a). Overall, there is a very small accuracy loss when the pruning ratio is below 85% for all pruning methods. Attention-aware pruning achieves similar accuracy compared to column pruning and tensor tile pruning. Our experiments in Figure 14(b) show that irregular is not hardware friendly since it leads to a higher latency (19×) than other pruning methods, with similar accuracy. For better visualization about how the weight matrix is pruned, Figure 13 shows an example of masks of in_proj_weight of Transformer under the aforementioned 4 different pruning methods, with pruning ratio of 50%, i.e., (i) $\mathbf{W_V}$ row pruning and others tensor tile pruning; (ii) irregular pruning; (iii) column pruning, and (iv) tensor tile pruning.

In Figure 14, attention-aware pruning has 1.19× speedup and 1.05× speedup compared to column pruning and tensor tile pruning respectively on average. In summary, our approach can achieve shorter turnaround time (maintaining tensor core friendly) for transformer-based models while maintaining the high prediction accuracy and high pruning ratio.



**(a) Prediction accuracy**



**(b) Latency**

**Figure 14: Test accuracy and Latency of the pre-trained model and different pruning ratios using Transformer.**

## 6 RELATED WORK

Recent years have seen a rise of interest in deep learning acceleration in our community. Although a majority of the well-known deep learning systems, such as TensorFlow [1] and PyTorch [42] support inference, training is their major focus. For inference, there exists a collection of specialized efforts, such as, TensorRT [40], Clipper [10], BatchMaker [18], TVM [8], and TensorFlow XLA [19]. In this section, we describe the efforts that are closely related to this work mainly from two aspects, that is, GPU-accelerated inference and model compression for transformer models.

**GPU-accelerated inference engine** includes efforts on convolutional neural network [47], recurrent neural network [24], and attention neural network [16, 21, 39, 40]. Among them, E.T. is closely related to Guo et al. [21] and TurboTransformer [16]. (i) Guo et al. [21] combines row/column pruning to accelerate matrix multiplications for inference. However, their design requires inputting the dense matrix (after row/column pruning) and the pruning mask instead of directly removing the pruned rows/columns. Consequently, [21] cannot introduce pruned resultant matrix for the follow-up computations. In contrast, E.T. revamps row/column pruning, introduces a tensor tile pruning, and, most importantly, attention-aware pruning designs. As shown in Table 1, with our attention-aware pruning, E.T. achieves 1.6×, and 2× speedup over column-based pruning (which is used in [21]). (ii) TurboTransformer [16] designs an inference system that improves the throughput for various queries. TurboTransformer processes sequences of different lengths via memory scheduling to avoid batch padding, which is absent from some fixed-size inference engines like TensorRT [40]. [16] also implements optimized and fused CUDA kernels for higher throughput. In comparison, E.T. focuses on the latency of a single inference with novel architectures and pruning designs. Therefore, E.T. could serve as the backend for TurboTransformer. We also notice a recent effort [17] that aims to accelerate attention computation. However, this work focuses upon *sparse* self-attention while E.T. accelerates conventional self-attention.

**Transformer model pruning.** Popular Transformer-based deep learning models often face the large model size and high computational costs challenges which motivate the need for weight pruning, and quantization [60]. Below, we focus on pruning. Irregular pruning [20], which heuristically prunes the redundant weights on arbitrary locations, often results in *irregular, sparse weight matrices*, and causes extra memory storage. Structured pruning [29] was proposed to structurally remove entire filters, channels, or filter shapes from the weight matrix. By taking advantage of the regular shapes of the pruned weight matrices, structured pruning avoids introducing extra indices to indicate the pruned locations and becomes more hardware-friendly. However, due to the coarse pruning granularity used in structured pruning, a considerable accuracy drop is observed under a high pruning rate. To strike a better balance, other pruning methods, such as row/column pruning [21] and hierarchical decomposition [17], focus on breaking the large matrix into multiple smaller sub-matrices for parallel execution, which maintains a regular pattern at the sub-matrix level for efficient execution but allows for irregular, arbitrary pruning at the global scale to maintain the high accuracy. Other techniques [48] such as parameter-sharing [25] and distillation have also been explored. While pruning is extensively explored by existing work, E.T. is the first to introduce tensor core supported pruning algorithms for transformer, as well as attention-aware pruning designs.

**E.T. tensor tile pruning vs. existing pruning methods.** There mainly exist four popular directions for pruning, e.g., magnitude-based, low-rank decomposition, Neural Architecture Search (NAS) and loss-based pruning. *(i) Magnitude-based pruning*, which is used by early work [23], often results in irregular weight distribution and is even difficult to be accelerated on current parallel architectures as reported in [54]. Not to mention our tensor core hardware. *(ii) Low-rank decomposition* [32] decomposed weights into smaller matrices to reduce the number of parameters. We conduct experiments using Singular Value Decomposition (SVD). On Transformer, our study shows that the low-rank-based method has worse performance than all the four methods in Figure 14(a). *(iii) Loss-based pruning*, such as structured pruning [54], of uses group Lasso as the relaxation of the hard constraint. Another loss-based pruning uses dynamic regularization, for instance, Alternating Direction Method of Multiplier (ADMM) to solve $l_0$ constraint problem [61]. 2× and 4× more pruning ratios are achieved compared to magnitude-based pruning and SVD-based pruning on AlexNet on ImageNet, respectively [61]. However, the resultant sparse matrices are irregularly distributed. And they are difficult to be accelerated on GPUs as reported in Table 1. *(iv) Neural architecture search (NAS)*-based [27] pruning replaces hand-crafted network architecture by an automatic learning process to search/prune the best network. NAS is orthogonal to our design as we could apply our attention-aware pruning on a NAS pruned Transformer model.

Our attention-aware pruning is tailored for Transformer acceleration on tensor cores. We extend reweighted optimization method, which is a loss-based design, for each tile to achieve a high compression rate under the same accuracy requirement than using a fixed penalty parameter in the group Lasso method. The resultant sparse matrices are highly compatible with tensor cores. More specifically, tensor tile pruning not only orchestrates the tensor core architecture for higher performance but also avoids both pre- and post- processing overhead that is suffered by column pruning. Our attention-aware pruning results in a pruned resultant matrix that can further speed up step ❻ in Figure 3(a) and 3(b) when deriving output. Therefore, our pruned transformer-based models achieve a high pruning ratio and superior performance over the mainstream projects on various NLP tasks.

## 7 DISCUSSION

**E.T. on other hardware platforms.** There are other emerging hardware platforms that aim at accelerating machine learning inference. AMD introduces an accelerator similar to Tensor Core called Matrix Core in the recent architecture [2]. Our hardware-friendly pruning could work on various fixed-size hardware accelerators by adjusting the pruning hyper-parameters.

On-the-fly attention is optimized for GPU but it has the potential to extend to other hardware. The idea is to avoid accessing slow memory and keep most of the operations on fast memory. For example, configurable spatial accelerators, such as Field-Programmable Gate Array (FPGA), are also equipped with fast on-chip Block RAM (BRAM) [57]. In fact, FPGA might gain more benefits with on-the-fly attention. Particularly, FPGA often equips larger on-chip memory when compared to GPU, so the batched multiplication of $\mathbf{Q}$ and $\mathbf{K}$ could fully reside on-chip to increase memory efficiency. Besides, more flexible scheduling and pipeline configurations could further reduce the breakdown among operators. The Softmax of a row in $\mathbf{Q}{\cdot}\mathbf{K}$ could be decomposed into two stages: sum and element-wise operation. With a coherent cache design of FPGA, the sum can be fused with the previous multiplication while the element-wise operation can be fused with the following $\mathbf{S}{\cdot}\mathbf{V}$ multiplication.

**E.T. for training.** As future work, we plan to integrate our novel tailored self-attention operators to training. Particularly, first, on-the-fly attention operator can directly substitute steps ❷ - ❻ of Figure 3(a). And this replacement will not affect the backward propagation of training. Second, pre-computed attention will slightly change the workflow of training. That is, the new architecture will not have $\mathbf{W_V}$ and $\mathbf{W_O}$ matrices anymore. It will directly use a new matrix, i.e., the $\|_{\mathrm{h=1}}^{\mathrm{H}} \mathbf{W}_{\mathbf{V},\mathrm{h}}^{\mathrm{T}} \cdot \mathbf{W}_{\mathbf{O},\mathrm{h}}^{\mathrm{T}}$ matrix. By writing the computation in the new flow, the backward propagation phase will use autograd to automatically update this new matrix as opposed to prior ones, i.e., $\mathbf{W_V}$ and $\mathbf{W_O}$.

## 8 CONCLUSION

This paper introduces E.T. with three main contributions: (i) We design a novel self-attention architecture with tailored on-the-fly attention and pre-computed linear transformation operators, sequence length-aware optimization, and scaling operator reordering designs. (ii) We revamp the existing pruning algorithm, as well as introducing a tensor tile-based pruning algorithm for transformer models. And E.T. goes further by introducing attention-aware pruning. Not limited there, (iii) we implement three transformer models, i.e., $\mathrm{BERT_{BASE}}$, DistilBERT, and Transformer on E.T. across a variety of benchmarks to show the impacts of our optimizations on accuracy and turnaround time. Taken together, E.T. outperforms the state of the art efforts, e.g., TensorRT and FasterTransformer.

## ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their helpful suggestions. This work was in part supported by the NSF CRII Award No. 2000722 and CAREER Award No. 2046102. Any opinions, findings and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

## REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, USA, 265–283.

[2] AMD. 2021. INTRODUCING AMD CDNA ARCHITECTURE. https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., Virtual, 1877–1901.

[4] Emmanuel J Candes, Michael B Wakin, and Stephen P Boyd. 2008. Enhancing sparsity by reweighted l1 minimization. *Journal of Fourier analysis and applications* 14, 5-6 (2008), 877–905.

[5] Daniel Cer, Mona Diab, Eneko Agirre, Iñigo Lopez-Gazpio, and Lucia Specia. 2017. SemEval-2017 Task 1: Semantic Textual Similarity Multilingual and Crosslingual Focused Evaluation. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*. The Association for Computer Linguistics, Vancouver, Canada, 1–14.

[6] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. 2020. Generative pretraining from pixels. In *International Conference on Machine Learning*. PMLR, Vienna, Austria, 1691–1703.

[7] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. 2020. The lottery ticket hypothesis for pre-trained bert networks. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., Vancouver, Canada.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, USA, 579–594.

[9] Jianpeng Cheng, Li Dong, and Mirella Lapata. 2016. Long Short-Term Memory-Networks for Machine Reading. (2016), 551–561.

[10] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A low-latency online prediction serving system. In *14th Symposium on Networked Systems Design and Implementation*. USENIX Association, Boston, MA, 613–627.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186.

[12] Bill Dolan and Chris Brockett. 2005. Automatically Constructing a Corpus of Sentential Paraphrases. In *Third International Workshop on Paraphrasing* (third international workshop on paraphrasing (IWP2005) ed.). Asia Federation of Natural Language Processing.

[13] Linhao Dong, Shuang Xu, and Bo Xu. 2018. Speech-transformer: a no-recurrence sequence-to-sequence model for speech recognition. In *International Conference on Acoustics, Speech and Signal Processing*. IEEE, Calgary, Alberta, Canada, 5884–5888.

[14] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. (2020). arXiv:arXiv preprint arXiv:2010.11929

[15] Angela Fan, Edouard Grave, and Armand Joulin. 2019. Reducing transformer depth on demand with structured dropout. (2019). arXiv:arXiv:1909.11556

[16] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, USA, 389–402.

[17] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, Virtual, 1–14.

[18] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, New York, NY, USA, 1–15.

[19] Google. 2021. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla

[20] Mitchell Gordon, Kevin Duh, and Nicholas Andrews. 2020. Compressing BERT: Studying the Effects of Weight Pruning on Transfer Learning. In *Proceedings of the 5th Workshop on Representation Learning for NLP*. Association for Computational Linguistics, Virtual, 143–155.

[21] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse DNN models without hardware-support via tile-wise sparsity. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Virtual, 1–15.

[22] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, Zhaohui Yang, Yiman Zhang, and Dacheng Tao. 2020. A Survey on Visual Transformer. (2020). arXiv:arXiv:2012.12556

[23] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*. MIT Press, Cambridge, MA, USA, 1135–1143.

[24] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. 2019. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, New York, NY, USA, 1–16.

[25] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. (2019). arXiv:arXiv:1909.11942

[26] Hector Levesque, Ernest Davis, and Leora Morgenstern. 2012. The winograd schema challenge. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*. AAAI Press, Rome, Italy.

[27] Xin Li, Yiming Zhou, Zheng Pan, and Jiashi Feng. 2019. Partial Order Pruning: For Best Speed/Accuracy Trade-Off in Neural Architecture Search. In *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Long Beach, CA, USA, 9137–9145.

[28] Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Lisbon, Portugal, 1412–1421.

[29] JS McCarley, Rishav Chakravarti, and Avirup Sil. 2019. Structured pruning of a bert-based question answering model. (2019). arXiv:arXiv:1910.06360

[30] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer Sentinel Mixture Models. (2017). arXiv:arXiv:1609.07843

[31] Sharan Narang, Eric Undersander, and Gregory Diamos. 2017. Block-sparse recurrent neural networks. *arXiv preprint arXiv:1711.02782* (2017).

[32] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. 2015. Tensorizing neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 1*. MIT Press, Cambridge, MA, USA, 442–450.

[33] NVIDIA. 2007. cuBLAS. https://developer.nvidia.com/cublas.

[34] NVIDIA. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[35] NVIDIA. 2020. CUTLASS. https://github.com/NVIDIA/cutlass.

[36] NVIDIA. 2020. Matrix Multiplication Background User Guide. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html#math-mem.

[37] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[38] NVIDIA. 2021. cuBLAS: cublasgemmalgo_t. https://docs.nvidia.com/cuda/cublas/index.html#cublasgemmalgo_t.

[39] NVIDIA. 2021. FasterTransformer. https://github.com/NVIDIA/DeepLearningExamples/tree/master/FasterTransformer.

[40] NVIDIA. 2021. TensorRT. https://developer.nvidia.com/tensorrt.

[41] Peyman Passban, Yimeng Wu, Mehdi Rezagholizadeh, and Qun Liu. 2020. ALP-KD: Attention-Based Layer Projection for Knowledge Distillation. In *Proceedings of the Conference on Artificial Intelligence*. AAAI Press, Virtual, 13657–13665.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. Curran Associates, Inc., Vancouver, Canada, 8026–8037.

[43] Sai Prasanna, Anna Rogers, and Anna Rumshisky. 2020. When BERT Plays the Lottery, All Tickets Are Winning. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Virtual, 3208–3229.

[44] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *International Symposium on High Performance Computer Architecture*. IEEE, San Diego, USA, 58–70.

[45] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Austin, Texas, 2383–2392.

[46] Qing Rao and Jelena Frtunikj. 2018. Deep learning for self-driving cars: Chances and challenges. In *Proceedings of the 1st International Workshop on Software Engineering for AI in Autonomous Systems*. IEEE, Gothenburg, Sweden, 35–38.

[47] Masuma Akter Rumi, Xiaolong Ma, Yanzhi Wang, and Peng Jiang. 2020. Accelerating Sparse CNN Inference on GPUs with Performance-Aware Weight Pruning. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. ACM, New York, NY, USA, 267–278.

[48] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. (2019). arXiv:arXiv:1910.01108

[49] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Seattle, Washington, USA, 1631–1642.

[50] Betty van Aken, Benjamin Winter, Alexander Löser, and Felix A Gers. 2019. How does bert answer questions? a layer-wise analysis of transformer representations. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. ACM, New York, NY, USA, 1823–1832.

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. Curran Associates Inc., Red Hook, NY, USA, 5998–6008.

[52] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2018. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Association for Computational Linguistics, Brussels, Belgium, 353–355.

[53] Ziheng Wang. 2020. SparseRT: Accelerating Unstructured Sparsity on GPUs for Deep Learning Inference. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. Association for Computing Machinery, New York, NY, USA, 31–42.

[54] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, 2074–2082.

[55] Adina Williams, Nikita Nangia, and Samuel Bowman. 2018. A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Association for Computational Linguistics, New Orleans, Louisiana, 1112–1122.

[56] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. arXiv:cs.CL/1910.03771

[57] Xilinx. 2017. SDAccel Environment Profiling and Optimization Guide. https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/jbt1504034294480.html.

[58] Canwen Xu, Wangchunshu Zhou, Tao Ge, Furu Wei, and Ming Zhou. 2020. BERT-of-Theseus: Compressing BERT by Progressive Module Replacing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Virtual, 7859–7869.

[59] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. 2020. Accelerating sparse matrix–matrix multiplication with GPU Tensor Cores. *Computers & Electrical Engineering* 88 (2020), 106848.

[60] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. 2020. Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference. In *53rd Annual International Symposium on Microarchitecture*. IEEE, Athens, Greece, 811–824.

[61] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. 2018. A systematic DNN weight pruning framework using alternating direction method of multipliers. In *Proceedings of the European Conference on Computer Vision*. Springer, Munich, Germany, 184–199.

[62] Xiaodong Zhang, Xu Sun, and Houfeng Wang. 2018. Duplicate question identification by integrating framenet with neural networks. In *Proceedings of the Conference on Artificial Intelligence*, Vol. 32. AAAI Press, New Orleans, Louisiana, USA.

[63] Xingxing Zhang, Furu Wei, and Ming Zhou. 2019. HIBERT: Document Level Pre-training of Hierarchical Bidirectional Transformers for Document Summarization. In *Proceedings of the 57th Annual Meeting*. Association for Computational Linguistics, Florence, Italy, 5059–5069.

# Appendix: Artifact Description/Artifact Evaluation

## SUMMARY OF THE EXPERIMENTS REPORTED

E.T. accelerates the inference of Transformer-family models by introducing a novel self-attention architecture and an attention-aware pruning design, which takes advantage of GPU hardware and Transformer-specific optimization.

We ran the experiment on one V100S GPU and Intel(R) Xeon(R) Gold 6244 @ 3.60GHz CPU. The training is conducted by the PyTorch library based on the pre-trained models. The evaluation of turnaround time is implemented on CUDA C++. The source code is compiled with NVCC 11.1. The prediction evaluation is conducted on General Language Understanding Evaluation (GLUE) benchmark. Our inference baselines are the example code from Faster Transformer (v3.1) and the BERT demo from TensorRT (v7.3). We ran the baselines on our machine and followed the instruction provided.

*Author-Created or Modified Artifacts:*

```
Persistent ID: https://github.com/cctry/SCpaper-2021⌋
↪  /tree/aedab163f44bff8dfad3745d4f57972cb7640cda
Artifact name: E.T.
```

## BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

*Relevant hardware details:* V100S GPU, Intel(R) Xeon(R) Gold 6244 @ 3.60GHz CPU

*Operating systems and versions:* Ubuntu 18.04

*Compilers and versions:* NVCC 11.1, GCC 7.5.0

*Applications and versions:* PyTorch 1.7.0

*Libraries and versions:* CUDA 11.1

*Key algorithms:* AdamW Optimizer, Self-attention Network

*Input datasets and versions:* Pre-trained models: HuggingFace's Transformers; Dataset: GLUE benchmark