# DFSSATTEN: DYNAMIC FINE-GRAINED STRUCTURED SPARSE ATTENTION MECHANISM

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Transformers are becoming mainstream solutions for various tasks like NLP and Computer vision. Despite their success, the quadratic complexity of their attention mechanism hinders them from applying to latency-sensitive tasks. Tremendous efforts have been made to alleviate this problem, and many of them successfully reduce the asymptotic complexity to linear. Nevertheless, few of them achieve practical speedup over the original full attention, especially under the moderate sequence length. In this paper, we present DFSSATTEN, an attention mechanism that dynamically prunes the full attention weight matrix to the 50% fine-grained structured sparse pattern used by the sparse tensor core on NVIDIA A100 GPU. We provide both theoretical and empirical evidence that demonstrate DFSSATTEN is a good approximation of the full attention mechanism and can achieve speedups in wall-clock time under arbitrary sequence length. We evaluate our method on tasks from various domains under different sequence lengths from 256 to 4096. DFSSATTEN achieves $1.27 \sim 1.89\times$ speedups over the full-attention mechanism with no accuracy loss on A100 GPU.

## 1 INTRODUCTION

Transformers (Vaswani et al., 2017) have achieved competitive performance across various domains like NLP (Ott et al., 2018) and Computer Vision (Dosovitskiy et al., 2021). The key feature that sets them apart from traditional neural network architectures is the attention mechanism (Vaswani et al., 2017), which allows the transformers to gather information from the embeddings of elements in the input sequence in an adaptive and learnable manner.

Nevertheless, the high computation cost and memory footprint brought by the attention mechanism make it difficult to apply transformers to latency-sensitive tasks. Many efficient attention mechanisms(Tay et al., 2020b; Zaheer et al., 2020; Beltagy et al., 2020; Tay et al., 2020a; Roy et al., 2021; Kitaev et al., 2020) have been proposed to address this issue. However, most of them drastically modify the original full attention mechanism and introduce a handful of hyper-parameters to tune. Therefore, they require tremendous engineering effort to deploy and optimize. Besides, they usually need to be trained from scratch instead of exploiting pretrained models like BERT (Devlin et al., 2019). Some of them rely on fixed sparse patterns or extremely high sparsity to achieve wall-clock time speedup. Therefore, these methods usually require thousands of pretraining or fine-tuning steps on specific tasks and toilsome tuning of several hyper-parameters to reach good accuracy. Last but not least, previous methods usually introduce additional operators like top-k, sort that cause large overheads and offset their benefits at moderate sequence length.

In this paper, we present DFSSATTEN, a simple and effective sparse attention mechanism that address the limitations mentioned above. DFSSATTEN *dynamically* prunes the full attention score matrix using 50% fine-grained structured sparse patterns (NVIDIA, 2020). This pattern is GPU friendly and can leverage the new sparse tensor core on NVIDIA A100 GPU (Mishra et al., 2021). Our DFSSATTEN offers several advantages over existing studies:

- It requires minimal changes to the original full-attention with no hyper-parameters to tune. This makes it a drop-in replacement of the full attention that only requires to change a few lines of code. Moreover, it can directly exploit existing pretrained models like BERT (Devlin et al., 2019) and RoBERTa (Liu et al., 2020).

- We dynamically prune the attention scores based on their magnitude under only 50% sparsity. This allows the pruned attention matrix to reserve the important entries, achieving on par model accuracy with full attention even without fine-tuning.
- Our method introduces zero overhead on existing GPU hardware. As a result, we are able to achieve wall-clock time speedup and memory footprint reduction over the full attention in arbitrary sequence length.

To conclude, our main **contributions** are summarized below:

- We propose DFSSATTEN, a dynamic sparse attention mechanism that is a drop-in proxy of the full attention mechanism. Its effectiveness is justified by both empirical and theoretical evidence.
- We present a dedicated CUDA kernel design to completely remove the pruning overhead. The pruning is implemented as an epilogue of the dense matrix multiplication which produces the attention score matrix.
- We evaluate DFSSATTEN on tasks cross various domains and sequence lengths. It achieves $1.27 \sim 1.89\times$ speedup over the full attention with no accuracy loss.

## 2 BACKGROUND AND MOTIVATION

We first introduce the preliminaries, notations, and background of our paper.

### 2.1 FULL ATTENTION MECHANISM

Given an input sequence $\boldsymbol{X} = (\boldsymbol{x}_1,..,\boldsymbol{x}_n) \in \mathbb{R}^{n\times d}$, the full attention mechanism can be defined as

$$\boldsymbol{O} = Softmax(\boldsymbol{Q}\boldsymbol{K}^T/\sqrt{d})\boldsymbol{V}, \tag{1}$$

where $\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_q$, $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_k$, and $\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_v$ are query, key, and value matrices. $\boldsymbol{Q}\boldsymbol{K}^T$ forms a full-quadratic adjacency matrix, whose edge weights are the dot-product similarity between all the elements in the sequence. This adjacency matrix is standardized with $1/\sqrt{d}$ to keep the unit second moment and then normalized with softmax. At last, the row feature vectors in $\boldsymbol{V}$ are aggregated according to the normalized adjacency matrix by multiplying them together. In the rest of this paper, we denote $\boldsymbol{A} = Softmax(\boldsymbol{Q}\boldsymbol{K}^T/\sqrt{d})$ for simplicity. We refer $\boldsymbol{Q}\boldsymbol{K}^T$ as the attention score matrix and $\boldsymbol{A}$ as the attention weight matrix.

### 2.2 EFFICIENT ATTENTION MECHANISM

The high computation cost and memory footprint in the full attention mechanism come from $\boldsymbol{A}$, whose size grows quadratically with the sequence length $n$. To address this issue, various efficient attention mechanisms have been proposed (Tay et al., 2020b).

**Fixed Sparse Patterns**. Zaheer et al. (2020); Beltagy et al. (2020) apply a set of fixed sparse attention patterns on $\boldsymbol{A}$, like global attention and sliding window attention. These patterns are constructed from empirical observations and designed GPU-friendly to achieve wall-time speedup. However, as these patterns are designed empirically and fixed during inference, there is no guarantee that they can always capture the important entries in $\boldsymbol{A}$ or transfer easily across different tasks.

**Dynamic Sparse Patterns**. Ham et al. (2021) dynamically generate fine-grained sparse attention patterns on $\boldsymbol{A}$ with low-cost binary hashing. However, this technique requires specialized hardware to achieve speedup, so it is not available on general-purpose hardware like GPU. Tay et al. (2020a); Roy et al. (2021); Kitaev et al. (2020) apply various clustering methods and only compute the attention within each cluster. Although computing full attention in each cluster is more friendly to GPU compared with fine-grained sparsity, the clustering methods contain several GPU-unfriendly operators like top-k and sorting that offsets their benefits under moderate sequence length.

**Low Rank / Kernel**. Wang et al. (2020) project $\boldsymbol{A}$ from $n \times n$ to $n \times k$ with linear projection. Choromanski et al. (2021) introduce the FAVOR+ which approximates the softmax with kernel method. This allows them to change the computation order and reduce the asymptotic complexity

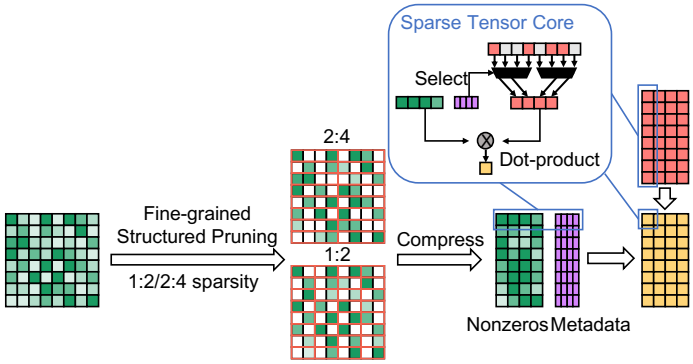


Figure 1: A100 GPU Fine-Grained Structured Sparsity Pruning. (NVIDIA, 2020)

to linear. However, the low-rank projection and kernel construction also introduce considerable overhead. This makes these methods only effective under long sequence length.

Besides, the previous studies drastically change the attention mechanisms, tens of thousands pre-training or finetuning steps are required to reach a comparable performance with the origin full attention mechanism. So they require tremendous engineering effort to deploy.

### 2.3 Fine-grained Structured Sparsity in NVIDIA A100 GPU

NVIDIA introduces the fine-grained structured sparsity in the A100 GPU. As shown in Figure 1, the dense input matrix is pruned with fine-grained structured pruning. If the data type is float, 1:2 sparsity is used which selects the larger one in two consecutive entries. If the data type is bfloat16 or float16, the 2:4 sparsity is used which selects two larger ones among four consecutive elements. After the pruning, the result is compressed to nonzeros and metadata. The nonzeros contain the value of reserved data that is 50% smaller than the original one. The metadata records the index of the nonzeros in the origins matrix. It takes 4 bit metadata to record the decision of each 1:2 or 2:4 selection. Therefore, the metadata is only 1/16 of the original dense matrix in terms of bits. This compressed sparse matrix can be multiplied with a dense matrix under the support of the sparse tensor core to achieve significant speedup.

This fine-grained structured sparsity has been applied to the static weight matrices in various neural network models including transformer (Mishra et al., 2021). It can effectively accelerate the feed-forward part of the transformer up to $1.9\times$. However, to the best of our knowledge, no previous studies use it in the attention mechanism where the attention weight matrix is *dynamically* generated for each sequence. One plausible explanation is that during pruning, GPU must read the whole dense matrix to be pruned from the memory. Then, after selecting the elements to be reserved under the 1:2 and 2:4 pattern, it must also generate the metadata encoded in a special format such that the metadata can be used efficiently later. All these overheads will offset the benefit brought by the pruning if we do it on the fly.

## 3 Dynamic Fine-grained Structured Sparse Attention Mechanism

In this section, we first give an overview of our DFSSATTEN method. Then, we discuss the design considerations of exploring sparsity in attention and the choice of sparse granularity in our method for GPU-friendly implementation and effectiveness. Finally, we briefly introduce our GPU kernel design to remove pruning overhead.

Our proposed DFSSATTEN mechanism is simple and effective, as illustrated in Figure 2. Compared with the full-quadratic attention mechanism, our method dynamically prunes attention scores without incurring storage or computation overhead, while maintaining the effectiveness of attention. More importantly, our method can achieve practical speedups of attention on existing GPU hardware with customized CUDA kernels. Listing 1 shows all the modifications to be made to use DFSSATTEN.

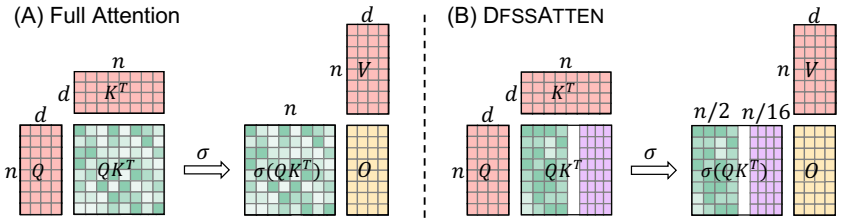


Figure 2: Overview of our Dynamic Fine-grained Structured Sparse Attention Mechanism. $\sigma$ represents *softmax*, $\frac{1}{\sqrt{d}}$ is omitted for simplicity.

```
# Full attention mechanism
def full_attention(q,k,v):
    attn_weight = torch.bmm(q, k.transpose(1, 2))
    attn_weight = torch.nn.functional.softmax(attn_weight, -1)
    return torch.bmm(attn_weight, v)

# DFSS attention mechanism
import dspattn
def dfss_attention(q,k,v):
    attn_weight, metadata = dspattn.bmm(q, k)
    attn_weight = torch.nn.functional.softmax(attn_weight, -1)
    return dspattn.spmm(attn_weight, metadata, v)
```

Listing 1: Example of using DFSSATTEN. The "dspattn" is the package we developed.

### 3.1 DESIGN CONSIDERATIONS FOR EXPLOITING ATTENTION SPARSITY

As illustrated in Figure 3, the attention mechanism can be considered as three stages: $\boldsymbol{QK}^T$, $Softmax$, and $\boldsymbol{AV}$. To design a sparse attention mechanism, the first decision to make is where should we induce pruning to sparsify the attention.

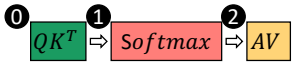


Figure 3: Attention Stages.

If we start from ⓪, all the three stages will be benefited from the sparsity given effective implementation on GPU: the dense matrix multiplication between $\boldsymbol{Q}$ and $\boldsymbol{K}$ will be replaced with the sampled dense-dense matrix multiplication (SDDMM) which only computes the entries identified by the sparse pattern. The $Softmax$ only operates on the nonzero values in each row. The original dense matrix multiplication between $\boldsymbol{A}$ and $\boldsymbol{V}$ will be replaced with a sparse matrix-matrix multiplication (SpMM) which multiplies a sparse matrix with a dense matrix. However, as it is not possible to exactly know which entry in $\boldsymbol{QK}^T$ has higher magnitude before computing $\boldsymbol{QK}^T$, starting from ⓪ usually requires some additional components to predict the location of important entries.

Starting from ❶ requires us to compute a dense matrix multiplication between $\boldsymbol{Q}$ and $\boldsymbol{K}$. The benefit is that we can explicitly select important entries from $\boldsymbol{QK}^T$ without prediction. As the softmax is a monotonically increasing function, starting from ❷ does not offer any benefits over ❶ but throws away the opportunity to accelerate $Softmax$.

In this paper, we choose to start from ❶ based on two considerations. First, replacing the dense matrix multiplication with SDDMM at $\boldsymbol{QK}^T$ offers limited speedup even at high sparsity. Chen et al. (2021b) show that it is difficult for SDDMM to achieve speedup over its dense counterpart under 80% sparsity even with some structured design. Second, starting from ❶ allows us to keep our design simple such that it does not introduce additional overhead or hyper-parameters to tune.

### 3.2 ON THE GRANULARITY OF SPARSE ATTENTION PATTERNS

The second decision to make is what sparse pattern to use as it will tremendously affect the latency of SpMM as well as the overhead to encode the sparse $\boldsymbol{QK}^T$. Existing studies exploit various sparse encoding schemes. For instance, the compressed sparse row (CSR) is popular for encoding fine-grained sparsity. However, CSR-based SpMM requires over 95% sparsity to be on par with its dense counterpart (Chen et al., 2021b). Block sparsity is also widely used as it can bring considerable wall-time speedup at moderate sparsity given large block size. However, it cannot capture some

fine-grained attention patterns. Moreover, these patterns require data comparisons within the entire row, which is difficult to execute in parallel and unfriendly to GPUs.

We find the fine-grained structured sparsity mentioned in Section 2.3 is a good choice as long as we address the pruning overhead. On one hand, the 1:2 and 2:4 selections are performed locally and are easy to execute in parallel. On the other hand, the size of compressed nonzeros is half of the original dense attention matrix, so the softmax only needs half of the computations. Powered by the NVIDIA Sparse Tensor Core, the SpMM between the compressed $\boldsymbol{A}$ and $\boldsymbol{V}$ can also achieve $1.7\times$ speedup.

### 3.3 Empirical Results of DfssAtten Mechanism

Empirically, we find this pattern can well approximate the full attention mechanism. We first finetune a BERT-large model on SQuAD v1.1 under full attention. Then, we directly replace the full attention with the 1:2 and 2:4 attention without additional finetuning. The F1-scores are summarized in Table 1 under $Cl = 95\%$. It is obvious that the accuracy loss is only around one sigma even without finetuning.

Table 1: F1 Score w/o Finetune on SQuAD v1.1

| **Full** | **1:2** | **2:4** |
|---|---|---|
| $93.17 \pm 0.27$ | $92.86 \pm 0.22$ | $93.00 \pm 0.16$ |

### 3.4 Removing Pruning Overhead

As mentioned in Section 2.3, the major challenge that hinders us from using the fine-grained structured sparse attention is the pruning overhead. We observe that when computing $\boldsymbol{QK}^T$, the results are first accumulated in GPU registers and written to memory when all the computations are done. Therefore, we can implement the pruning as an epilogue of the matrix multiplication: after the accumulation is finished, we compare the data stored in the registers, select the larger ones and generate the metadata. Then, we only write the reserved non-zeros and metadata to memory. This design brings two benefits. First, it completely removes the overhead caused by reading the matrix to be pruned from memory, so it has zero overhead. Second, the memory footprint caused by the attention weight matrix is reduced from $n^2 \times 32\text{-}bit$ to $\frac{n^2}{2} \times 32\text{-}bit + \frac{n^2}{16} \times 32\text{-}bit$ as the original $n \times n$ full attention weight matrix is not written to memory. The more detailed description of the CUDA kernel design including how to encode the metadata on the fly is summarized in Appendix A.1.

## 4 Theoretical Results

In this section, we provide more theoretical and empirical evidence that justify our DfssAttn as a good proxy of the full attention mechanism. The strategy is to first derive the theoretical value of 1) quality of the approximation with different sparse patterns 2) speedup can be achieved under certain sparsity. Then, we compare the quality of different methods under the same speedup.

### 4.1 Attention Lottery Ticket

We borrow the lottery ticket hypothesis (Frankle & Carbin, 2019) and extend it to the attention mechanism. The last step $\boldsymbol{AV}$ in the attention mechanism can be viewed as the aggregation in the graph neural network. Following the Generalized Attention Mechanism (Zaheer et al., 2020), we describe it with a weighted directed graph $\mathcal{G} = (\boldsymbol{A}, \boldsymbol{X})$. $\boldsymbol{A}$ is the adjacent matrix and $\boldsymbol{A}_{u,v} > 0$ indicates that element $\boldsymbol{x}_u$ attends to $\boldsymbol{x}_v$. Inspired by the *Graph Lottery Tickets* (Chen et al., 2021a), we propose the *Attention Lottery Ticket* as follows.

**Attention Lottery Ticket (ALT)**. Given a fully connected d graph $\mathcal{G} = \{\boldsymbol{A}, \boldsymbol{X}\}$ constructed from the full quadratic attention mechanism (Vaswani et al., 2017), the associated sub-graph can be defined as $\mathcal{G}_s = \{\boldsymbol{m} \odot \boldsymbol{A}, \boldsymbol{X}\}$, where $\boldsymbol{m}$ is a binary mask. If a $\mathcal{G}_s$ has the performance matching or surpassing the original full quadratic attention mechanism, then we define the sparse attention mechanism with $\mathcal{G}_s$ as an *attention lottery ticket*.

Zaheer et al. (2020) have proved the existence of lottery tickets by showing 1) sparse attention mechanisms are universal approximators of sequence to sequence functions when being used as

encoder 2) sparse encoder-decoder transformers are Turing Complete. So the remaining problem is how to identify the winning tickets $\mathcal{G}_s$ at runtime.

### 4.2 Quality of the Lottery Ticket

A popular strategy that empirically works well is selecting the top-k neighborhood in $\mathcal{G}$ based on the magnitude of edge weight. We refer it as *Top-k Sparsity*. Intuitively, this strategy is based on the hypothesis that the edges with larger edge weight are more important. It has been widely adapted in existing studies (Frankle & Carbin, 2019; Chen et al., 2021a; Ham et al., 2021; Wang et al., 2021) and demonstrated its ability to preserve model accuracy at a high sparsity ratio. Following this trend of work, we define the Quality of Attention Lottery Ticket as follows:

**Definition 4.1.** ***($L^p$-Quality of Attention Lottery Ticket)*** *The quality of attention lottery ticket $\mathcal{G}_s = \{\boldsymbol{m} \odot \boldsymbol{A}, \boldsymbol{X}\}$ under density $s = \frac{1}{n^2}\sum_{j=1}^{n}\sum_{i=1}^{n} \boldsymbol{m}_{j,i}$ is defined as*

$$\mathcal{Q}^p = \frac{1}{n}\sum_{j=1}^{n} \frac{\sum_{i=1}^{n} (\boldsymbol{m} \odot \boldsymbol{A})_{j,i}^p}{\sum_{i=1}^{n} \boldsymbol{A}_{j,i}^p}. \tag{2}$$

The above definition computes the expectation of normalized $L^p$ norm in each row of the attention score matrix. The $p$ is a task-dependent factor that indicates how the accuracy depends on the edges with higher magnitude. In this paper, we compare the $L^p$-Quality of tickets yield by three types of sparse patterns: Top-K, fixed, and our dynamic 1:2 and 2:4 sparse pattern. Particularly, we have the proposition below:

**Proposition 4.1.** *Under the assumption that the entries in $\boldsymbol{Q}\boldsymbol{K}^T/\sqrt{d}$ follow i.i.d. $\mathcal{N}(\mu, \sigma)$, we have*

$$\mathcal{Q}^p_{topk}|_s \approx \frac{1 + erf\left(\frac{p\sigma}{\sqrt{2}} - erfinv(1-2s)\right)}{2}, \quad \mathcal{Q}^p_{fix}|_s = s, \quad \mathcal{Q}^p_{2:4} \geq \mathcal{Q}^p_{1:2} = \frac{1 + erf\left(\frac{p\sigma}{2}\right)}{2} \tag{3}$$

*(Proof: Appendix A.2)*

It is obvious that the $\mathcal{Q}^p_{topk}$ achieves the upper bound of $\boldsymbol{Q}^p$ under $s$. Besides, the $p\sigma$ is always positive. Therefore, we also have $\mathcal{Q}^p_{2:4} \geq \mathcal{Q}^p_{1:2} > \mathcal{Q}^p_{fix}|_{s=0.5} = 1/2$.

### 4.3 Efficiency of the Lottery Ticket

A lottery ticket with high quality does not necessarily mean that it is also efficient to execute for wall-clock time speedup. In this section, we analyze the efficiency of the three sparse patterns.

**Top-K Sparsity**. Zhao et al. (2019) explicitly select k neighbors in each row of $\boldsymbol{A}$ based on their magnitude. However, as shown in their Table 4, the explicit sparse transformer has lower inference throughput despite $k \ll n$. On one hand, the top-k operator is difficult to parallel and introduces high overhead. On the other hand, even if an oracle top-k sparsity mask $\boldsymbol{m}$ were provided with zero overhead, it would still be difficult for the explicit Top-K sparse attention to beat its dense counterpart. We provide a theoretical upper bound for density $s$ in Proposition 4.2.

**Proposition 4.2.** *Given embedding size $d$ and the maximum tiling size supported by GPU $T$, the upper bound of the speedup achieved by Top-K Sparsity under density $s$ is (Proof: Appendix A.3)*

$$Speedup < \frac{4d + 3T}{2d + T + (d + 2T + dT)s}. \tag{4}$$

As typical values for the dimension $d$ and tiling size $T$ are $d = 64, T = 128$, we need $s < 4.5\%$ for $Speedup > 1$. Notably, this is not a strict upper bound as we did not take the overhead of identifying top-k entries into consideration. Therefore, the strict upper bounder should be even smaller.

**Fixed Sparsity**. As the fixed sparse pattern are designed or learned before inference, they can be designed to be GPU-friendly and have the same tiling size with the dense matrix multiplication. Therefore, we can derive the upper bound of the speedup under density $s$ with the same strategy in Proposition 4.2:

$$Speedup \leq \frac{n^2\left(\frac{2d}{T}+1\right)+2n^2+nd\left(\frac{2n}{T}+1\right)}{sn^2\left(\frac{2d}{T}+1\right)+2n^2s+nd\left(\frac{(1+s)n}{T}+1\right)} \overset{n\gg d}{\approx} \frac{4d+3T}{(1+3s)d+3sT}. \tag{5}$$

**Dynamic 1:2 / 2:4 Sparsity**. Similarly, we can derive the theoretical speedup with 1:2 and 2:4 sparsity as follows

$$Speedup = \frac{n^2\left(\frac{2d}{T}+1\right)+2n^2+nd\left(\frac{2n}{T}+1\right)}{n^2\left(\frac{2d}{T}+\frac{1}{2}+\frac{1}{16}\right)+n^2+nd\left(\frac{n}{T}+\frac{n}{2T}+\frac{n}{16T}+1\right)} \overset{n\gg d}{\approx} \frac{64d+48T}{57d+25T}. \tag{6}$$

### 4.4 Quality of the Lottery Tickets under the same Efficiency

With the theoretical conclusions above, we compare the quality of the lottery ticket under our dynamic 1:2 and 2:4 sparsity with the other two methods under the same efficiency.

**Comparison with Top-K Sparsity**. The Top-K sparsity achieves the same efficiency with ours at

$$s < \frac{(4d+3T)(57d+25T)}{(64d+48T)(d+2T+dT)} - \frac{2d+T}{(d+2T+dT)}. \tag{7}$$

With typical values $T = 128, d = 64$, we have $s \approx 0.02$. We can substitute it to Proposition 4.1 and get $\mathcal{Q}^p_{topk} < \mathcal{Q}^p_{1:2}$ when $p\sigma < 7$. On the other hand, when $p\sigma > 7$, although the Top-K sparsity produces tickets with higher quality, $\mathcal{Q}^p_{1:2}|_{p\sigma=7} \approx 0.9999996$ is already very close to 1.

**Comparison with Fixed Sparsity**. The fixed sparsity achieves the same efficiency with ours when

$$s \approx \frac{(4d+3T)(64d+48T)}{(57d+25T)(3d+3T)} - \frac{d}{3d+3T}. \tag{8}$$

With typical values $T = 128, d = 64$, we have $s \approx 0.63$. On the other hand, we have theoretical value of $\sigma \approx 1$ and $p \geq 1$ based on the observation that the edges with higher magnitude are more influential. Therefore, we have $p\sigma \geq 1$ and $\mathcal{Q}^p_{1:2} \geq 0.76 > 0.63 = \mathcal{Q}^p_{fix}|_{s=0.63}$.

To conclude, compared with both top-k sparsity and fixed sparsity, our method can always yield lottery tickets with higher quality under the same efficiency. To support this conclusion, we further provide some empirical studies in Appendix A.4. Besides, we found both theoretically and empirically that our method is a good complementary to the kernel-based transformers like Performer (Choromanski et al., 2021). We add more discussions about it in Appendix A.5.

## 5 Evaluation

In this section, we first evaluate the accuracy of our dynamic fine-grained structured sparse attention mechanism on tasks across different domains. Then, we profile our methods on NVIDIA A100 GPU under different sequence lengths from 256 to 4096 to show that we can achieve practical speedup in arbitrary sequence length.

### 5.1 Model Accuracy

To show that our method is effective in comprehensive scenarios, we first evaluate the model accuracy on tasks in different domains and sequence length. For models under "bfloat16" data type, we first finetune them from the pretrained model under "float" data type as "float" provides more precise gradient that helps convergence. After the finetuning, we directly cast all the parameters in the model to "bfloat16" and test it on the test dataset. For Question Answering and Masked Language Modeling tasks, we report the results averaged over 8 runs under different random seeds.

**Question Answering**. We evaluate the BERT-large on SQuAD v1.1 under sequence length 384. We use the "bert-large-uncased-whole-word-masking" in Huggingface (Wolf et al., 2020) as the pretrained model and finetune it with the default configuration in Huggingface [1]. The F1-scores of "1:2(float)" and "2:4(bfloat16)" without finetuning are obtained by directly using the checkpoints from "Transformer(float)".

[1] https://github.com/huggingface/transformers/tree/master/examples/pytorch/question-answering

Table 2: F1 score on BERT-large SQuAD v1.1 (Cl=95%)

| **Model** | **w/o finetune** | **w/ finetune** |
|---|---|---|
| Transformer (float) | - | $93.17 \pm 0.27$ |
| Transformer (bfloat16) | - | $93.18 \pm 0.27$ |
| DFSSATTEN 1:2 (float) | $92.86 \pm 0.22$ | $93.07 \pm 0.17$ |
| DFSSATTEN 2:4 (bfloat16) | $93.00 \pm 0.16$ | $\mathbf{93.28 \pm 0.29}$ |

As shown in Table 2, with finetuning, our 1:2 sparsity has only 0.1 F1 score loss that is smaller than the standard deviation. Our 2:4 sparsity even achieves a little bit of performance improvement over the dense baseline. One plausible explanation is that while the 2:4 sparsity can keep most of the important edges, it also occasionally drops a small fraction of important edges which acts like the attention dropout technique (Zehui et al., 2019). Besides, directly applying our methods to the dense transformer without finetuning also achieves comparable results, it justifies that our method can well approximate the dense attention mechanism.

**Masked Language Modeling**. We also evaluate our models on the masked modeling tasks on Wikitext-2 and Wikitext-103 under sequence length 512. Similar to the question-answering tasks, we choose the "roberta-large" as the pretrained model and finetune it under the default configuration in Huggingface [2]. The results are summarized in Table 3. Similarly, the perplexities achieved by our methods are on par with the dense transformer.

Table 3: Perplexity on roBERTa-large (Cl=95%)

| Model | Wikitext-2 | | Wikitext-103 | |
|---|---|---|---|---|
| | **w/o finetune** | **w/ finetune** | **w/o finetune** | **w/ finetune** |
| Transformer (float) | - | $\mathbf{2.83 \pm 0.09}$ | - | $2.62 \pm 0.04$ |
| Transformer (bfloat16) | - | $2.85 \pm 0.07$ | - | $2.63 \pm 0.05$ |
| DFSSATTEN 1:2 (float) | $2.88 \pm 0.06$ | $2.88 \pm 0.07$ | $2.64 \pm 0.06$ | $2.64 \pm 0.06$ |
| DFSSATTEN 2:4 (bfloat16) | $2.88 \pm 0.07$ | $2.84 \pm 0.04$ | $2.63 \pm 0.03$ | $\mathbf{2.61 \pm 0.04}$ |

**Long Range Arena**. For sequence length longer than 512, we incorporate three tasks from the Long Range Arena (Tay et al., 2021), including Text Classification, Document Retrieval, and Image Classification under sequence lengths 2048, 4096, and 1024, respectively. For a fair comparison with other efficient transformers, the model is trained from scratch under the default configurations. The results are summarized in Table 4. Our method achieves comparable accuracy on all the three benchmarks for long sequence.

Table 4: Accuracy of different transformer models on LRA benchmark. We follow the training instructions from Tay et al. (2021) to reuse the results from this paper.

| Model | Text (n=2048) | Retrieval (n=4000) | Image (n=1024) | Avg |
|---|---|---|---|---|
| Transformer (float) | 65.05 | 61.72 | 42.15 | 56.31 |
| Transformer (bfloat16) | 65.03 | 61.73 | 42.17 | 56.31 |
| Local Attention | 52.98 | 53.39 | 41.46 | 50.89 |
| Sparse Trans. | 63.58 | 59.59 | **44.24** | 55.80 |
| Longformer | 62.85 | 56.89 | 42.22 | 53.99 |
| Linformer | 53.94 | 52.27 | 38.56 | 48.26 |
| Reformer | 56.10 | 53.40 | 38.07 | 49.19 |
| Sinkhorn Trans. | 61.20 | 53.83 | 41.23 | 52.09 |
| Synthesizer | 61.68 | 54.67 | 41.61 | 52.65 |
| BigBird | 64.02 | 59.29 | 40.83 | 54.71 |
| Linear Trans. | **65.90** | 53.09 | 42.34 | 53.78 |
| Performer | 65.40 | 53.82 | 42.77 | 54.00 |
| **DFSSATTEN 1:2 (float)** | 64.95 | 61.83 | 42.02 | 56.27 |
| **DFSSATTEN 2:4 (bfloat16)** | 64.91 | **62.26** | 42.31 | **56.49** |

### 5.2 SPEEDUP

In this section, we demonstrate the speedup achieved by our method across different sequence lengths. For a fair comparison, we only show the speedup achieved on the attention mechanism declared in equation 1. This is because the end-to-end speedup can be affected by various other

[2]https://github.com/huggingface/transformers/tree/master/examples/pytorch/language-modeling

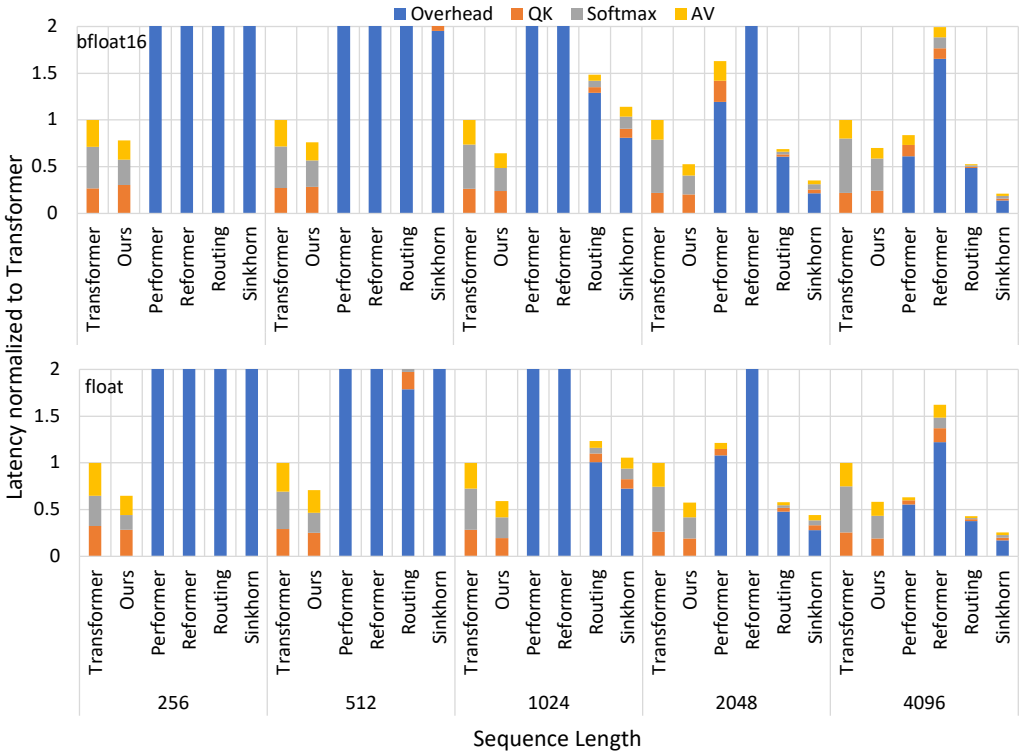


Figure 4: Latency breakdown of different attention mechanism. For each configuration, we normalize the latency to Transformer with full attention mechanism and cut off the axis at 2 for clarity.

factors including the quantization and pruning strategies applied to the other parts of the transformer models, the embedding sizes, and the efficiency of the code implementation (e.g. some operators could be fused for lower latency). For models in previous studies, We also apply the PyTorch JIT script when possible in case that their implementations are not efficient. The configuration we use is as follows: Each layer contains 4 heads, the feature dimension per head is 64. The batch size is set to be large enough to keep the GPU busy. We summarize the profiling results in Figure 4. We normalize the latency to the Transformer with full attention mechanism under each configuration. We also cut the y axis off at 2 for clarity, because some methods designed for long sequence could be more than $20\times$ slower than the dense transformer at moderate sequence length.

First of all, our method achieves $1.27\sim1.89$x speedup over the transformer with full attention. It is the only method that brings consistent speedup across different sequence lengths, while other methods from previous papers suffer from high overhead at moderate and short sequence lengths. Second, under the float data type, our method achieves speedup in all the three stages with zero overhead. This accords with our arguments in Section 3. Under the data type bfloat16, the $\boldsymbol{QK}^T$ in our method is a little bit slower than the dense baseline. The reason is that selecting 2 larger ones from 4 elements requires more comparisons, which results in more warp divergence.

## 6 Conclusion and Discussion

In this paper, we present DFSSATTEN, a dynamic fine-grained structured sparse attention mechanism that dynamically prunes the $\boldsymbol{QK}^T$ on the fly to 1:2 and 2:4 structured sparsity. As it only requires 50% sparsity, it can achieve no accuracy loss compared with the full attention mechanism across tasks in various domains and sequence lengths. Besides, it only requires modifying a few lines of code, which makes it a drop-in replacement of the full-attention mechanism. Moreover, powered by our customized CUDA kernels and the new sparse tensor core on Ampere GPU, we achieve $1.27\sim1.89\times$ speedup over the full attention in arbitrary sequence length. All of these pieces of evidence demonstrate that our method can be a good replacement for the full attention mechanism. Our method is also orthogonal to many existing efficient attention mechanisms and can potentially be applied jointly for further speedup. We present two examples in Appendix A.6.

## REFERENCES


Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Tianlong Chen, Yongduo Sui, Xuxi Chen, Aston Zhang, and Zhangyang Wang. A unified lottery ticket hypothesis for graph neural networks. In *International Conference on Machine Learning*, pp. 1695–1706. PMLR, 2021a.

Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. 2021b.

Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Ua6zuk0WRH.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, pp. 4171–4186, 2019. URL https://aclweb.org/anthology/papers/N/N19/N19-1423/.

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. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.

Andrew Kerr et al. Cutlass. https://github.com/NVIDIA/cutlass, 2021.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=rJl-b3RcF7.

Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W Lee. Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 692–705. IEEE, 2021.

Andrew Kerr. Gtc 2020: Developing cuda kernels to push tensor cores to the absolute limit on nvidia a100. https://developer.nvidia.com/gtc/2020/video/s21745, 2020.

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkgNKkHtvB.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Ro{bert}a: A robustly optimized {bert} pretraining approach, 2020. URL https://openreview.net/forum?id=SyxS0T4tvS.

Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

NVIDIA. Nvidia a100 tensor core gpu architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. Scaling neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pp. 1–9, 2018.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

Yi Tay, Dara Bahri, Liu Yang, Donald Metzler, and Da-Cheng Juan. Sparse sinkhorn attention. In *International Conference on Machine Learning*, pp. 9438–9447. PMLR, 2020a.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020b.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena : A benchmark for efficient transformers. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=qVyeW-grC2k.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 97–110. IEEE, 2021.

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

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. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. In *NeurIPS*, 2020.

Lin Zehui, Pengfei Liu, Luyao Huang, Junkun Chen, Xipeng Qiu, and Xuanjing Huang. Dropattention: A regularization method for fully-connected self-attention networks. *arXiv preprint arXiv:1907.11065*, 2019.

Guangxiang Zhao, Junyang Lin, Zhiyuan Zhang, Xuancheng Ren, Qi Su, and Xu Sun. Explicit sparse transformer: Concentrated attention through explicit selection. *arXiv preprint arXiv:1912.11637*, 2019.

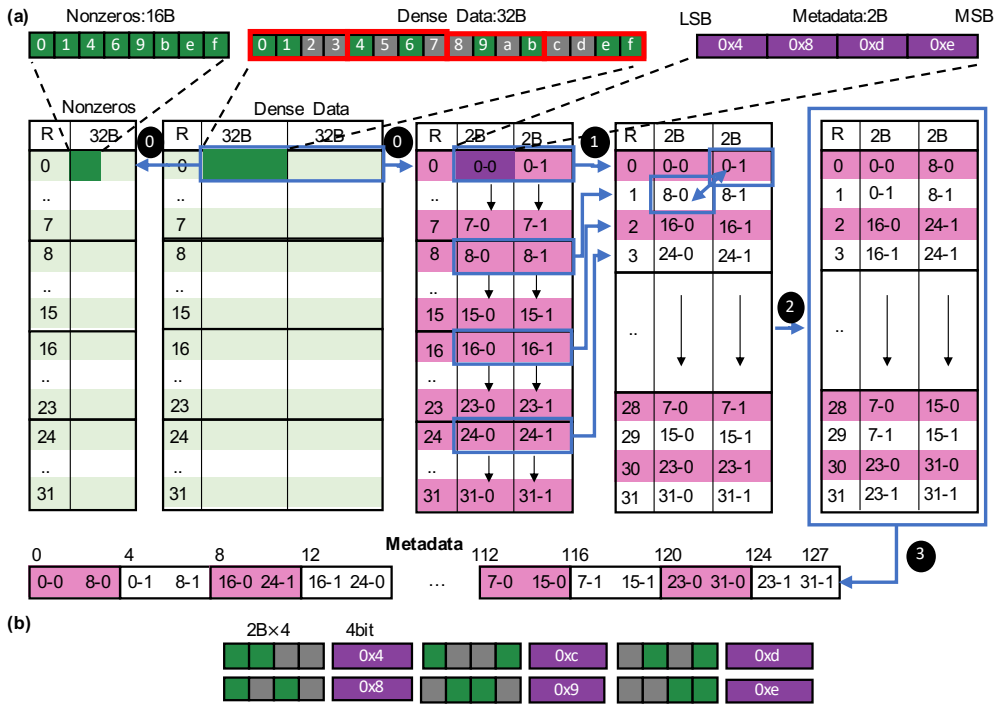


Figure 5: Prune dense data and generate nonzeros, metadata.

# A Appendix

## A.1 Kernel Design Details

In this section, we first demonstrate how a dense matrix is pruned and compressed under the 50% structured sparsity on Ampere GPU in Section A.1.1. Then, we detail the design and implementation of the SDDMM, Softmax, and SpMM kernels in Section A.1.2 and A.1.3.

### A.1.1 Structured Pruning of Dense Matrix

We first illustrate how to dynamically prune a dense matrix with 50% structured sparsity. Under data type float, we select the larger one in every two consecutive elements. If the data type is bfloat16, we select two larger ones in every four consecutive elements. We compress the pruned dense matrix to nonzeros and metadata following CUTLASS et al (2021) as there are two benefits. First, it can be directly used by high-performance SpMM kernels in CUTLASS. Second, as we will show in Section A.1.2, it can be dynamically generated from the SDDMM kernel with neglectable overhead.

As shown in Figure 5, the basic tile size to prune is $32 \times 64$-byte, this corresponds to a $32 \times 32$ block under bfloat16 or $32 \times 16$ block under float. There are four major steps: ⓪ Prune 50% of each consecutive 8B data, generate nonzeros and metadata; ❶ Interleave the metadata rows by 8; ❷ Switch the metadata along sub-diagonal. ❸ Write metadata and nonzeros to global memory.

In detail, 2 out of 4 2-byte data are select based on their magnitude and a unique 4-bit metadata is assigned to each combination in ⓪. The correspondence between selection pattern and metadata is enumerated in Figure 5 (b). Notably, with float32 data type, each 32-bit data occupies two consecutive 2-byte slots. Therefore, it only supports the patterns under *0x4* and *0xe*. After generating the 4-bit metadata, consecutive four of them are concatenated to a 2B metadata block. Then, the rows of metadata are interleaved by 8 in ❶ following

$$dst_row = \lfloor row/32 \rfloor \times 32 + (row\%8) \times 4 + \lfloor (row\%32)/8 \rfloor. \tag{9}$$

In ❷, the metadata blocks at upper right and lower left of each $2 \times 2$ grid are switched. At last, in ❸, the metadata produced by ❷ is written into global memory following the interleaved column-

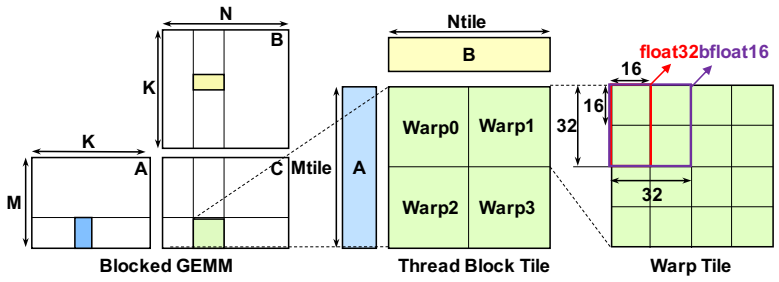


Figure 6: Dense Matrix Multiplication (GEMM) Tiling Design.

major format under stride 4-byte. This can be realized by interpreting two consecutive metadata as an int object and then write it to DRAM in column-major. The nonzeros are simply writen to global memory under row-major.

### A.1.2 SDDMM Kernel Design

Our strategy for dynamically pruning the attention score matrix has two steps. First, perform a conventional dense GEMM. Second, prune the GEMM output with procedures described in Section A.1.1. However, if the second step is implemented as a separate GPU kernel, we need to write the dense attention score matrix to DRAM and read it back. This not only introduces high overhead, but also prevents us from reducing global memory footprint. To address this issue, we implement the pruning step as an epilogue attached to the conventional GEMM kernel: the results of the dense GEMM are stored in the registers, the epilogue processes the results and then writes nonzeros and metadata to global memory.

**Dense GEMM**. The GEMM step is no different from conventional GEMM kernels, and all the existing optimizations can be used. The tiling is shown in Figure 6: each thread block processes a $Mtile \times Ntiles$ output tile, which is further partitioned to several warp tiles. Each warp tile is composed of a grid of $16 \times 16$ blocks that matches the tensor core output size. In each thread block, all the threads jointly load $Mtile \times Ktile$ and $Ktile \times Ntile$ input tiles from matrix A and B into the shared memory. We use the new synchronize copy feature on Ampere architecture to fully utilize the memory bandwidth and reduce register usage. To fully annihilate shared memory bank conflict, we use the XOR layout. Once the load is completed, the warps fetch their source operands from shared memory with *ldmatrix* and perform a $(16 \times 32B) \cdot (32B \times 16)$ warp matrix multiply accumulate (*wmma*) with tensor core. Notably, float data will be converted to *tensorfloat-32* before *wmma*. To reduce accumulation error, we accumulate the partial sum as float regardless of the source operand data type. Besides, software 2-stage pipeline is used to overlap memory access and computation with double buffering (et al, 2021). Although deeper software pipeline can be built on Ampere, we find 2 stages is enough as the inner-product dimension $K$ is usually very small (e.g. 64). More detailed explanation of the above techniques can be found in this GTC 2020 talk (Kerr, 2020).

**Pruning the GEMM result**. In the pruning step, the warp tile is partitioned to a grid of $32 \times 64B$ blocks that are processed by the warp one at a time.

Under data type float, the register layout of the $32 \times 16$ block is illustrated in Figure 7 (a). It consists of two $16 \times 16$ *wmma* blocks, so each thread has sixteen 32-bit registers to hold the results. The registers are annotated with "T*thread_id*{register_id}". As the adjacent two data are held by the same thread, we can simply compare them and the larger one is retained.

Under data type bfloat16, we need to select 2 larger ones from adjacent 4 entries. However, under the naive mapping shown in Figure 8 (a), these 4 entries are held by 2 thread. Therefore, we need additional warp shuffle to first pass these 4 entries to the same thread, then compare them and obtain the 2 larger ones. This will introduce additional overhead. To solve this problem, we propose to interleave the columns when loading matrix B to shared memory by simply manipulating the pointer to the global memory at the beginning. The resulted mapping to the registers is shown in Figure 8 (b) which is equivalent with Figure 7 (a) bfloat16. After the interleaving, consecutive four

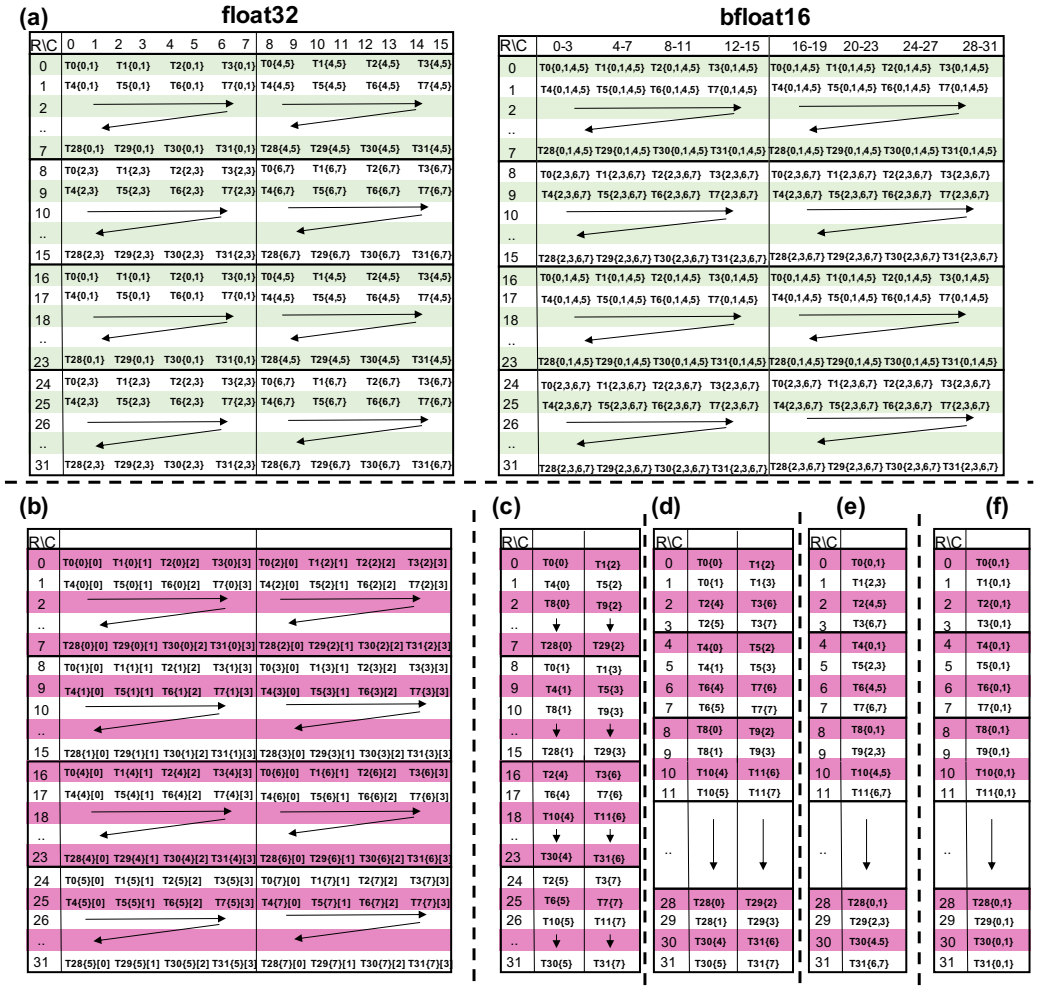


Figure 7: Mapping between the registers and data, metadata.

data are naturally held by the same thread, and we select 2 larger ones from them. To reduce branch divergence, the selection is done by comparing the sum of any two data.

**Generate Metadata and Nonzeros**. For both float and bfloat16 data type, each comparison produces a 4-bit metadata. Next, following the procedures described in Section A.1.1, we need to concatenate consecutive four metadata to a 16-bit metadata block. This is done in two steps. First, put the 4-bit metadata to the correct position of a int16 register with bit shift. Second, share these int16 registers cross threads with warp shuffle, and concatenate them with bitwise OR. As consecutive four metadata are held by thread *4t* to *4t+3*, we put the 4-bit metadata of thread *4t+k* to $[k\times4{:}k\times4+3]$ bits in the *int16* object in the first step. The detailed layout is shown in Figure 8 (b), where we denote each 4-bit metadata as "T*thread_id*{*register_id*}[*bit_id*]". The result of the second step is shown in Figure 7 (c). Figure 7 (d) and (e) illustrate the result after ❶ and ❷ in Figure 5. Notably, these two step only change the logic mapping of the metadata and the register allocation is not affected. So no code is required for these two steps. At last, we need to write the metadata and nonzeros to global memory following ❸ in Figure 5. As shown in Figure 7 (e), each row is held by consecutive two *int16* registers of the same thread, so we can simply reinterpret it as an *=int32* object and write the metadata to global memory in column major. For the nonzeros, we simply coalesce them in the shared memory and then write to global memory in row-major.

**Batched Kernel**. The self-attention layer in transformer usually has multiple independent attention heads. Instead of launching one CUDA kernel for each attention head, using a batched kernel that

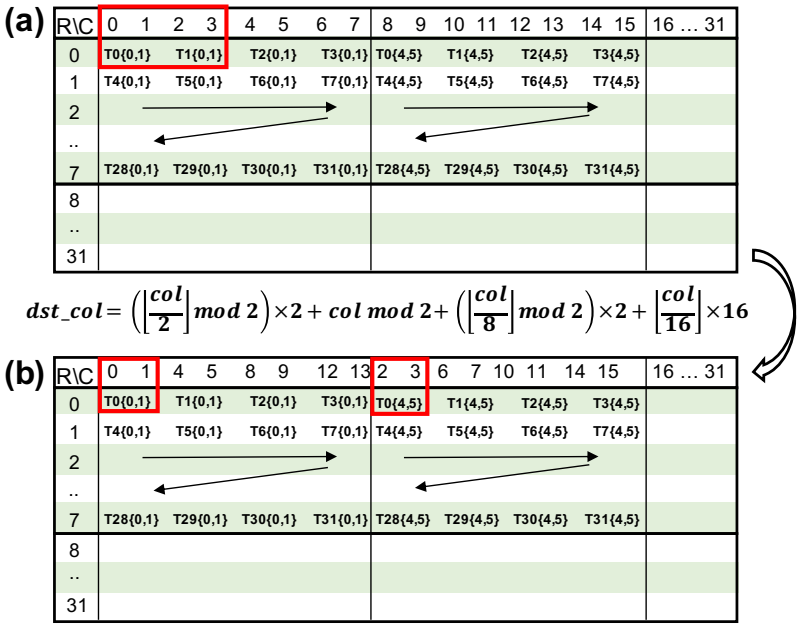


Figure 8: Interleave the columns for matrix B to reduce cross-lane data sharing during pruning for bfloat16.

processes all the heads can better utilize the GPU resources and reduce kernel launching overhead. We support the batched computation by using the *blockIdx.z* to index the heads in the batch and update the pointers to the input and output based on the index.

**Blocked-ELL Sparsity**. Under long sequence length, higher sparsity is desired to reduce computation cost and memory footprint. Our kernel support hybrid blocked-ELL sparsity Zaheer et al. (2020) and 50% structured sparsity. To support this feature, we set the block size in blocked-ELL to the thread block tile size of the GEMM. Therefore, we can simply skip those pruned blocks during the execution.

### A.1.3 Softmax and SpMM Kernel

In this section, we detail the implementation of the softmax and SpMM kernels.

**Softmax Kernel**. To improve numerical stability, the softmax on GPU is computed with

$$softmax(\boldsymbol{x})_i = \frac{e^{x_i - max(\boldsymbol{x})}}{\sum_j e^{x_j - max(\boldsymbol{x})}}. \tag{10}$$

Therefore, each element in $\boldsymbol{x}$ has to be loaded for three times. 1) compute $c = max(\boldsymbol{x})$; 2) compute $s = \sum_j e^{x_j - c}$; 3) compute $e^{x_i - c}/s$. Instead of loading $x_i$ from global memory in each time, we cache it in the register when the whole row fits in the register file capacity. Besides, the ordinary softmax kernel in libraries like PyTorch can also be used.

**SpMM Kernel**. As we encode the nonzeros and metadata following the CUTLASSet al (2021), we directly construct the SpMM kernels from the CUTLASS APIs. To support the hybrid blocked-ELL and structured 50% sparsity, we modify the *PredictedTileAccessIterator* class in CUTLASS to skip the tiles masked out by the blocked-ELL sparsity.

## A.2 Proof of Proposition 4.1

*Proof.* Under the assumption that the entries in $\boldsymbol{QK}^T/\sqrt{d}$ follow i.i.d. $\mathcal{N}(\mu, \sigma)$, we denote $x_{i,j} = e^{\mu + \sigma z_{i,j}}$, where $z \sim i.i.d.\ \mathcal{N}(0, 1)$. Then we can substitute it into the definition of the softmax and

get

$$\boldsymbol{A}_{u,v} = \frac{x_{u,v}}{\sum_{i=1}^{n} x_{u,i}}. \tag{11}$$

We substitute the above equation into the definition of $L^p$-Quality and get

$$\mathcal{Q}^p = \frac{1}{n}\sum_{j=1}^{n} \frac{\sum_{i=1}^{n} (\boldsymbol{m} \odot \boldsymbol{A})_{j,i}^p}{\sum_{i=1}^{n} \boldsymbol{A}_{j,i}^p} = \frac{1}{n}\sum_{j=1}^{n} \frac{\frac{1}{n}\sum_{i=1}^{n} m_{j,i} x_{j,i}^p}{\frac{1}{n}\sum_{i=1}^{n} x_{j,i}^p} \tag{12}$$

With $n \to \infty$, the denominator can be approximated with

$$\frac{1}{n}\sum_{i=1}^{n} x_{j,i}^p \approx \int_{-\infty}^{\infty} \frac{e^{p\mu+p\sigma z}}{\sqrt{2\pi}} exp\left(-\frac{z^2}{2}\right) dz = exp\left(p\mu + \frac{p^2\sigma^2}{2}\right) \tag{13}$$

**Top-K Sparsity**. When the sequence is long enough such that we have $n \to \infty$, the numerator can be approximated with

$$\begin{aligned} \frac{1}{n}\sum_{i=1}^{n} m_{j,i} x_{j,i}^p &\approx \int_{\sqrt{2}erfinv(1-2s)}^{\infty} \frac{e^{p\mu+p\sigma z}}{\sqrt{2\pi}} exp\left(-\frac{z^2}{2}\right) dz \\ &= exp\left(p\mu + \frac{p^2\sigma^2}{2}\right) \frac{1 + erf\left(\frac{p\sigma}{\sqrt{2}} - erfinv(1-2s)\right)}{2} \end{aligned} \tag{14}$$

Therefore, the $L^p$-Quality of Top-K sparsity is

$$\mathcal{Q}_{topk}^p \approx \frac{1 + erf\left(\frac{p\sigma}{\sqrt{2}} - erfinv(1-2s)\right)}{2}. \tag{15}$$

**Fixed Sparsity**. Without any assumption on the distribution of important edges in $\boldsymbol{A}$, applying a fixed pattern is equivalent with uniformly sampling with probability $s$ and we have

$$\frac{1}{n}\sum_{i=1}^{n} m_{j,i} x_{j,i}^p \approx exp\left(p\mu + \frac{p^2\sigma^2}{2}\right) s. \tag{16}$$

Therefore, the $L^p$-Quality of Top-K sparsity is

$$\mathcal{Q}_{fix}^p \approx s. \tag{17}$$

**2-to-1 Sparsity**: This sparsity pattern select the larger one in every two elements. We denote adjacent two elements with

$$X = e^{\mu+\sigma Z_1}, Y = e^{\mu+\sigma Z_2}; Z_1, Z_2 \sim \mathcal{N}(0,1), \tag{18}$$

$Z_1$ and $Z_2$ are independent. Then we have

$$\begin{aligned} &\frac{1}{n}\sum_{i=1}^{n} m_{j,i} x_{j,i}^p \approx \frac{1}{2}\mathbb{E}\left[max(X^p, Y^p)\right] = \\ &\frac{1}{2}\left[\iint_{z_1 \geq z_2} e^{p\mu+p\sigma z_1} \frac{1}{2\pi} exp\left(-\frac{z_1^2+z_2^2}{2}\right) dz_1 dz_2 + \iint_{z_1 < z_2} e^{p\mu+p\sigma z_2} \frac{1}{2\pi} exp\left(-\frac{z_1^2+z_2^2}{2}\right) dz_1 dz_2\right] \end{aligned} \tag{19}$$

We denote

$$x = \frac{z_1 - z_2}{\sqrt{2}}, \quad y = \frac{z_1 + z_2}{\sqrt{2}}, \tag{20}$$

then we have

$$\begin{aligned} &\iint_{z_1 \geq z_2} e^{p\mu+p\sigma z_1} \frac{1}{2\pi} exp\left(-\frac{z_1^2+z_2^2}{2}\right) dz_1 dz_2 \\ &= \int_{-\infty}^{\infty}\int_{0}^{\infty} e^{p\mu+\frac{p\sigma}{\sqrt{2}}(x+y)} \frac{1}{2\pi} exp\left(-\frac{x^2+y^2}{2}\right) dxdy \\ &= \int_{-\infty}^{\infty}\int_{0}^{\infty} e^{p\mu+\frac{p^2\sigma^2}{2}} \frac{1}{2\pi} exp\left[-\frac{1}{2}\left(x - \frac{p\sigma}{\sqrt{2}}\right)^2 - \frac{1}{2}\left(y - \frac{p\sigma}{\sqrt{2}}\right)^2\right] dxdy \\ &= e^{p\mu+\frac{p^2\sigma^2}{2}} \int_{0}^{\infty} \frac{1}{\sqrt{2\pi}} exp\left[-\frac{1}{2}\left(x - \frac{\sigma}{\sqrt{2}}\right)^2\right] dx = \frac{e^{p\mu+\frac{p^2\sigma^2}{2}}}{2}\left[1 + erf\left(\frac{p\sigma}{2}\right)\right]. \end{aligned} \tag{21}$$

With the conclusion above, we have

$$\frac{1}{n}\sum_{i=1}^{n} m_{j,i}x_{j,i}^{p} = exp\left(p\mu + \frac{p^2\sigma^2}{2}\right)\frac{1+erf\left(\frac{p\sigma}{2}\right)}{2}. \tag{22}$$

The $L^P$-Quality of 1:2 sparsity can be computed with

$$\mathcal{Q}_{1:2}^{p} = \frac{1+erf\left(\frac{p\sigma}{2}\right)}{2}. \tag{23}$$

**2:4 Sparsity**: This sparsity pattern select the largest two elements in consecutive four elements. While it is more challenging to find an explicit expression for $\mathcal{Q}_{4-to-2}^{p}$, a trivial lower bound can be found with

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n} m_{j,i}x_{j,i}^{p} &\approx \frac{1}{4}\mathbb{E}\left[max(X^p+Y^p, X^p+U^p, X^p+Vp, Y^p+U^p, Y^p+V^p, U^p+V^p)\right] \\ &\geq \frac{1}{4}\left(\mathbb{E}[max(X^p,Y^p)] + \mathbb{E}[max(U^p,V^p)]\right) = \frac{1}{2}\mathbb{E}[max(X^p,Y^p)],\end{aligned} \tag{24}$$

where we have

$$X = e^{\mu+\sigma Z_1}, Y = e^{\mu+\sigma Z_2}, U = e^{\mu+\sigma Z_3}, V = e^{\mu+\sigma Z_4}; Z_1, Z_2, Z_3, Z_4 \sim \mathcal{N}(0,1), \tag{25}$$

$Z_1, ..., Z_4$ are independent. Therefore, the lower-bound of $\mathcal{Q}_{2:4}^{p}$ is

$$\mathcal{Q}_{2:4}^{p} \geq \frac{1+erf\left(\frac{p\sigma}{2}\right)}{2}. \tag{26}$$

□

### A.3 Proof of Proposition 4.2

*Proof.* First of all, thanks to the Tensor Core in latest GPUs, the latency of matrix multiplication operations, both sparse and dense, are bounded by the memory access. Therefore, instead of counting the number of MACs (multiply-accumulate operations), the amount of memory access is a better metric to estimate the latency.

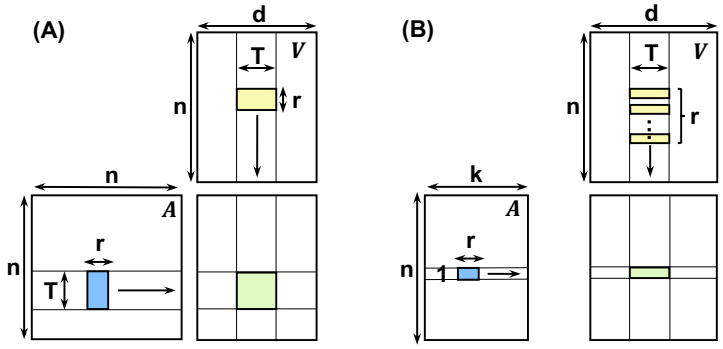


Figure 9: Tiling Matrix-Matrix Multiply

Tiling is a basic optimization applied to optimize matrix matrix multiply on GPU. As shown in Figure 9 (A), the original $n \times n$ output is partitioned to independent blocks with size $T \times T$. When computing each block, operands with size $T \times r$ and $r \times T$ are loaded from $\boldsymbol{A}$ and $\boldsymbol{V}^T$ to the fast memory, respectively. Then, these two operand are multiplied and accumulated to the partial sum stored in the registers. After applying the top-k, as shown in Figure 9 (B), the k elements in each row of $\boldsymbol{A}$ correspond to different rows in $\boldsymbol{A}$. Therefore, we can only partition the output to independent

vectors with size $1 \times T$. During the computation, operands with size $1 \times r$ and $r \times T$ are loaded from $\boldsymbol{A}$ and $\boldsymbol{V}^T$ to the fast memory, respectively. Then, the loaded operands are multiplied and accumulated to the partial sum stored in the registers.

With the tiling strategy mentioned above, we can summarize the amount of memory access in different attentions in the table below.

Table 5: Amount of Memory Access in Different Operations in Attention. $s = k/n$: density of the sparse attention; $T$: tiling size.

| | $\boldsymbol{QK}^T$ | Softmax | $\boldsymbol{AV}$ |
|---|---|---|---|
| Full Attention | $n^2\left(\frac{2d}{T}+1\right)$ | $2n^2$ | $nd\left(\frac{2n}{T}+1\right)$ |
| Explicit Top-k Attention | $n^2\left(\frac{2d}{T}+1\right)$ | $2n^2s$ | $nd\left(sn+\frac{sn}{T}+1\right)$ |

For $\boldsymbol{QK}^T$, as we need to compute all of it before getting the top-k elements, it is a dense matrix matrix multiplication for both full and explicit top-k attention. The Softmax needs to read the $n \times n$ $\boldsymbol{QK}^T$ in, normalizes it, and write the result $\boldsymbol{A}$ back. As the intermediate values can be stored in registers, we only need to count reading $\boldsymbol{QK}^T$ in and writing $\boldsymbol{A}$ out. Therefore, its memory access is $2n^2$ for full attention and $2n^2s$ for explicit top-k attention. For $\boldsymbol{AV}$ in full attention, the output size is $nd$. As each output element is generated from the inner product between two vectors with length $n$, the total data read equals $nd \times 2n$. However, with the tiling in Figure 9 (A), each operand is reused for $T$ times. Therefore, the total memory access for $\boldsymbol{AV}$ in full attention is $nd(\frac{2n}{T}+1)$. For $\boldsymbol{AV}$ in explicit top-k attention, as shown in Figure 9 (B), each left-hand-side data is reused for $T$ times while each right-hand-side data is used only for once. Therefore, its memory access equals to $nd(\frac{sn}{T}+sn+1)$.

The theoretical speedup can be computed with

$$Speedup \leq \frac{n^2\left(\frac{2d}{T}+1\right)+2n^2+nd\left(\frac{2n}{T}+1\right)}{n^2\left(\frac{2d}{T}+1\right)+2n^2s+nd\left(sn+\frac{sn}{T}+1\right)} \overset{n \gg d}{\approx} \frac{4d+3T}{2d+T+(d+2T+dT)s} \tag{27}$$

□

### A.4 Quality of the Lottery Tickets under the same Efficiency

In this section, we provide more empirical evidences to support our conclusions. In Figure 10, we compute the theoretical value (solid line) and empirical value (box plot) of $\mathcal{Q}^p$ over attention matrix $\boldsymbol{A}$ in BERT-Large on SQuAD v1.1. As $p$ is a task-dependent value that is hard to obtain, we instead sweep through several typical values.

Compared with the top-k sparsity, when $p < 7$, our 1:2 and 2:4 sparsity always achieve better performance than the top-k sparsity when $s < 0.05$. Besides, when $p = 7$, the $\mathcal{Q}^p_{1:2}$ and $\mathcal{Q}^p_{2:4}$ are very close to 1. These observations accord with our conclusion that our 1:2 and 2:4 sparsity can obtain tickets with better quality than Top-K sparsity at the same efficiency.

Compared with the fixed sparsity, our $\mathcal{Q}^p_{1:2}$ and $\mathcal{Q}^p_{2:4}$ are also similar or better than $\mathcal{Q}^p_{fix}$ across different $p$s. This supports our conclusion that our method achieves better performance than the fixed sparsity patterns under the same efficiency.

### A.5 Comparison with Performer

In this section, we add more discussions on how our method compared with kernel based transformer, i.e. Performer (Choromanski et al., 2021). As our Definition 4.1 is designed to characterize how well the sparse pattern could reserve the important edges in $\boldsymbol{A}$, so it is not suitable for kernel-based attention mechanisms that do not involve sparsity. For example, an approximation of $\boldsymbol{A}$ with high positive approximation error can have $\mathcal{Q}^P \geq 1$ under Definition 4.1. Therefore, we instead compare the mean squared error (MSE) following Choromanski et al. (2021). Given the query and two adjacent key vectors $\boldsymbol{q}$, $\boldsymbol{k}$, and $\boldsymbol{k}' \in \mathcal{N}(0, \boldsymbol{I}_d)$, we denote the softmax kernel between them as $SM(\boldsymbol{q}, \boldsymbol{k}) = exp(\boldsymbol{q}^T\boldsymbol{k}/\sqrt{d})$. And the softmax approximated by our dynamic 1:2 sparsity

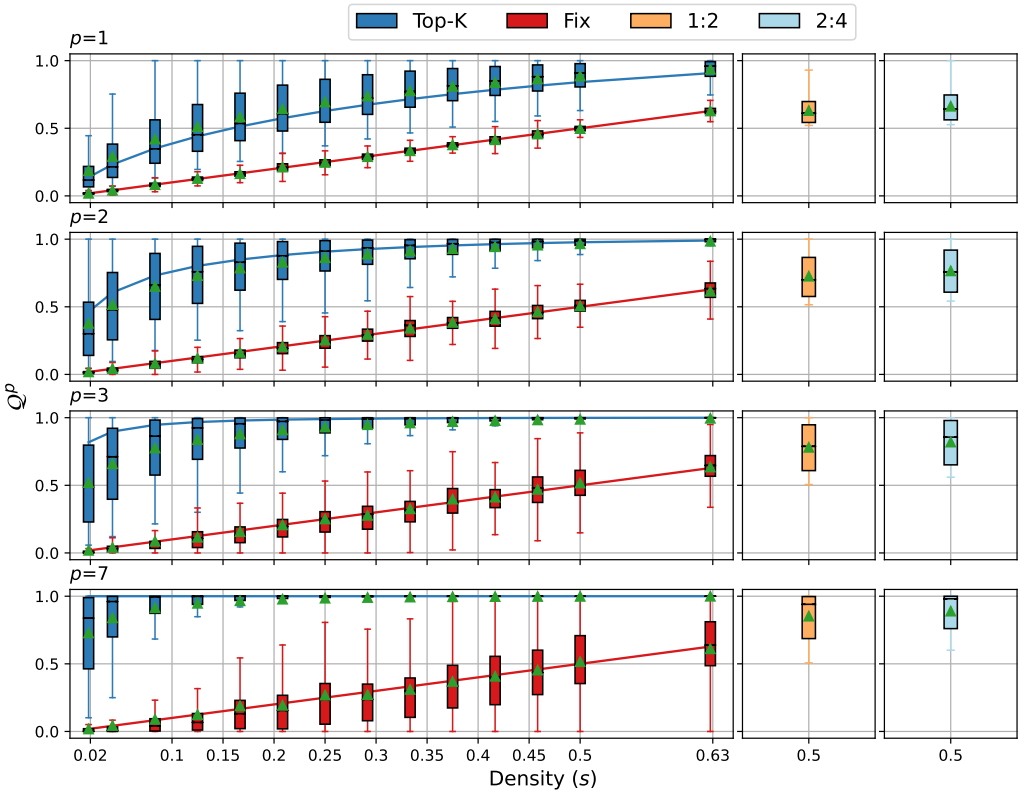


Figure 10: $\mathcal{Q}^p$ under different density $s$ and sparsity strategies. Box plot: Empirical results from BERT-large on SQuAD v1.1; Solid line: Theoretical results from Proposition 4.1.

$\widehat{SM_{1:2}}(\boldsymbol{q},\boldsymbol{k})$ is defined as

$$\widehat{SM_{1:2}}(\boldsymbol{q},\boldsymbol{k}) = \begin{cases} exp\left(\frac{\boldsymbol{q}^T\boldsymbol{k}}{\sqrt{d}}\right) & if \;\; \boldsymbol{q}^T\boldsymbol{k} > \boldsymbol{q}^T\boldsymbol{k}' \\ 0 & else \end{cases}. \tag{28}$$

Then, we can compute its MSE as follows

$$MSE(\widehat{SM_{1:2}}(\boldsymbol{q},\boldsymbol{k})) = \int_{\boldsymbol{q}^T\boldsymbol{k}<\boldsymbol{q}^T\boldsymbol{k}'} exp\left(\frac{2\boldsymbol{q}^T\boldsymbol{k}}{\sqrt{d}}\right) 2\pi^{-d/2} exp\left(-\frac{||\boldsymbol{k}'||_2^2}{2}\right) d\boldsymbol{k}'. \tag{29}$$

Because $\boldsymbol{q}^T\boldsymbol{k}' = \sum_{i=1}^d \boldsymbol{q}_i\boldsymbol{k}'_i$ is the weighted sum of i.i.d variables following $\mathcal{N}(0,1)$, we have $x = \boldsymbol{q}^T\boldsymbol{k}' \sim \mathcal{N}(0, ||\boldsymbol{q}||_2^2)$. We can substitute it into equation 29 and get

$$\begin{aligned} MSE(\widehat{SM_{1:2}}(\boldsymbol{q},\boldsymbol{k})) &= exp\left(\frac{2\boldsymbol{q}^T\boldsymbol{k}}{\sqrt{d}}\right) \int_{x>\boldsymbol{q}^T\boldsymbol{k}} \frac{1}{\sqrt{2\pi||\boldsymbol{q}||_2^2}} exp\left(-\frac{x^2}{2||\boldsymbol{q}||_2^2}\right) dx \\ &= exp\left(\frac{2\boldsymbol{q}^T\boldsymbol{k}}{\sqrt{d}}\right) \frac{1-erf\left(\frac{\boldsymbol{q}^T\boldsymbol{k}}{||\boldsymbol{q}||_2\sqrt{2}}\right)}{2} = SM^2(\boldsymbol{q},\boldsymbol{k}) \frac{1-erf\left(\frac{\sqrt{d}}{||\boldsymbol{q}||_2\sqrt{2}} ln\left(SM(\boldsymbol{q},\boldsymbol{k})\right)\right)}{2}. \end{aligned} \tag{30}$$

With Lemma 2 and Theorem 2 in Choromanski et al. (2021), the MSE of their positive softmax kernel with orthogonal random features has an upper bound as follows

$$\begin{aligned} MSE\left(\widehat{SM_m^{ort+}}(\boldsymbol{q},\boldsymbol{k}))\right) &\leq \frac{1}{m} exp\left(\frac{2\boldsymbol{q}^T\boldsymbol{k}}{\sqrt{d}}\right) \left[exp\left(\frac{||\boldsymbol{q}+\boldsymbol{k}||^2}{\sqrt{d}}\right) - 1 - \left(1-\frac{1}{m}\right)\frac{2}{d+2}\right] \\ &= \frac{1}{m} SM^2(\boldsymbol{q},\boldsymbol{k}) \left[exp\left(\frac{||\boldsymbol{q}||_2^2+||\boldsymbol{k}||_2^2}{\sqrt{d}}\right) SM^2(\boldsymbol{q},\boldsymbol{k}) - 1 - \left(1-\frac{1}{m}\right)\frac{2}{d+2}\right]. \end{aligned} \tag{31}$$

First of all, when $SM(\boldsymbol{q},\boldsymbol{k}) \to 0$, both $MSE(\widehat{SM}_{1:2}(\boldsymbol{q},\boldsymbol{k}))$ and $MSE\left(\widehat{SM}_m^{ort+}(\boldsymbol{q},\boldsymbol{k}))\right)$ converge to 0. However, for large $SM(\boldsymbol{q},\boldsymbol{k})$s that are potentially be critical for the model accuracy, the $exp\left(\frac{||\boldsymbol{q}||_2^2+||\boldsymbol{k}||_2^2}{\sqrt{d}}\right)SM^2(\boldsymbol{q},\boldsymbol{k})$ term in the positive softmax kernel in Performer could greatly increases the MSE. Oppositely, the $1-erf\left(\frac{\sqrt{d}}{||\boldsymbol{q}||_2\sqrt{2}}ln\left(SM(\boldsymbol{q},\boldsymbol{k})\right)\right)$ term in our method reduces the MSE. To conclude, while both the positive softmax kernel and ours has low MSE error when approximating small edge weights, our method can better approximate the edges with high magnitude.

From the empirical perspective, as shown in Table 2 and 3, our method can achieve good accuracy even without finetuning. Whereas the Performer still requires tens of thousands steps of finetuning (e.g. Figure 5 in Choromanski et al. (2021)). Table 4 also reveals that Performer has poor accuracy on certain tasks like byte-level document retrieval, while ours consistently achieve accuracy on par with the dense transformer. All this observations suggest that our method can better approximate the full attention mechanism than Performer.

In terms of wall-clock time speedup, Figure 4 illustrates that the Performer can only achieve good speedup at long sequence length. The similar phenomenon is also observed in multiple online forums [3]. Certainly, the PyTorch JIT script does not yield the optimal implementation of the computation graph, but it reveals that tremendous engineering efforts are required for Performer to achieve good speedup under moderate sequence length.

Following Section 4.3, we also compare the theoretical speedup achieved by ours and the Performer.

$$
\begin{aligned}
&\boldsymbol{T}^{(\mathbf{1})}_{n\times m} = \frac{\boldsymbol{Q}_{n\times d}}{\sqrt[4]{d}}\boldsymbol{P}_{d\times m},\ \ \boldsymbol{T}^{(\mathbf{2})}_{n\times 1} = \frac{1}{2\sqrt{d}}\sum_{i=1}^{d}\left[\boldsymbol{Q}_{n\times d}\odot\boldsymbol{Q}_{n\times d}\right]_{:,i}\\
&\boldsymbol{T}^{(\mathbf{3})}_{n\times 1} = max_i[\boldsymbol{T}^{(\mathbf{1})}_{n\times m}]_{:i},\ \ \phi(\boldsymbol{Q}_{n\times m}) = \frac{1}{\sqrt{m}}exp\left(\boldsymbol{T}^{(\mathbf{1})}_{n\times m}-\boldsymbol{T}^{(\mathbf{2})}_{n\times 1}-\boldsymbol{T}^{(\mathbf{3})}_{n\times 1}+\epsilon\right)\\
&\boldsymbol{T}^{(\mathbf{4})}_{n\times m} = \frac{\boldsymbol{K}_{n\times d}}{\sqrt[4]{d}}\boldsymbol{P}_{d\times m},\ \ \boldsymbol{T}^{(\mathbf{5})}_{n\times 1} = \frac{1}{2\sqrt{d}}\sum_{i=1}^{d}\left[\boldsymbol{K}_{n\times d}\odot\boldsymbol{K}_{n\times d}\right]_{:,i}\\
&\boldsymbol{T}^{(\mathbf{6})}_{n\times 1} = max_i[\boldsymbol{T}^{(\mathbf{4})}_{n\times m}]_{:i},\ \ \phi(\boldsymbol{K}_{n\times m}) = \frac{1}{\sqrt{m}}exp\left(\boldsymbol{T}^{(\mathbf{4})}_{n\times m}-\boldsymbol{T}^{(\mathbf{5})}_{n\times 1}-\boldsymbol{T}^{(\mathbf{6})}_{n\times 1}+\epsilon\right)\\
&\boldsymbol{T}^{(\mathbf{7})}_{m\times 1} = \sum_{i=1}^{n}[\phi(\boldsymbol{K})_{n\times m}]_{i,:},\ \ \boldsymbol{T}^{(\mathbf{8})}_{n\times 1} = 1/\left(\phi(\boldsymbol{Q})_{n\times m}\times\boldsymbol{T}^{(\mathbf{7})}_{m\times 1}\right)\\
&\boldsymbol{T}^{(\mathbf{9})}_{m\times d} = \phi(\boldsymbol{K})^T_{n\times m}\times\boldsymbol{V}_{n\times d},\ \ \boldsymbol{T}^{(\mathbf{10})}_{n\times d} = \phi(\boldsymbol{Q})_{n\times m}\times\boldsymbol{T}^{(\mathbf{9})}_{m\times d}\odot\boldsymbol{T}^{(\mathbf{8})}_{n\times 1}.
\end{aligned}
\tag{32}
$$

The computation steps of Performer are listed in equation 32 where each equation denotes a sub computation graph that can potentially be fused. Notably, this is more complex than the original mathematical expression to handle the numerical instability of $exp$. The total memory access can be computed with

$$
\begin{aligned}
Speedup=&\{2\left[nm\left(\frac{2d}{T}+1\right)+n(d+1)+n(m+1)+n(m+3)\right]+m(n+1)+n(\frac{m}{T}+m+1)\\
&+md\left(\frac{2n}{T}+1\right)+nd\left(\frac{2m}{T}+1\right)+n\}/\left[n^2\left(\frac{2d}{T}+1\right)+2n^2+nd\left(\frac{2n}{T}+1\right)\right].
\end{aligned}
\tag{33}
$$

We have $m = dln(d)$ following Theorem 4 in Choromanski et al. (2021). We can substitute $m = 266$, $d = 64$, and $T = 128$ into equation 33 and get $Speedup > 1$ when $n > 672$. On the other hand, the performer achieves the same speedup with ours with $n > 1002$.

To conclude, our method is a good complementary to performer. With delicately optimized computation graph, performer can achieve good speedup and relatively good accuracy under long sequence scenario. In contrary, our method has better speedup and accuracy under moderate and short

[3] https://github.com/huggingface/transformers/issues/7675

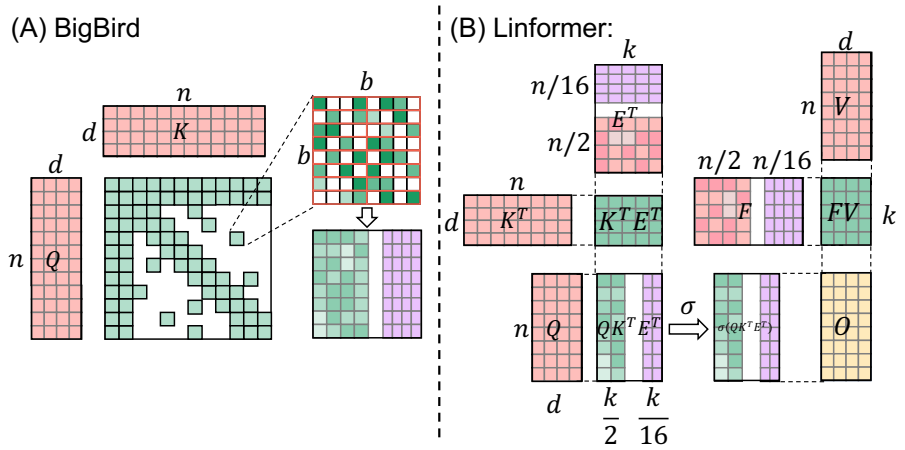


Figure 11: Combination of our method with BigBird (Zaheer et al., 2020) and Linformer (Wang et al., 2020)

sequence length. Besides, our method delivers lower approximation error on important edges so it is more friendly to finetuning.

### A.6 Combination with the Existing Efficient Transformers

Existing efficient transformers usually sparsify the full attention mechanism to densely connected clusters (Tay et al., 2020a; Roy et al., 2021; Kitaev et al., 2020; Zaheer et al., 2020) or approximate it with low-rank projection (Wang et al., 2020). As our method is a good approximation of the full attention mechanism and brings wall time speedup at arbtrary sequence length, it can potentially be combined with the existing efficient transformers.

For instance, as shown in Figure 11 (A), Zaheer et al. (2020) use block sparsity with block size 64 and compute a full attention within each block. We can apply the 1:2 or 2:4 sparsity within each block to bring further speedup.

Figure 11 (B) gives another example on how to combine our method with Linformer (Wang et al., 2020). Linformer uses low-rank approximation on the attention mechanism as follows:

$$\boldsymbol{O} = softmax\left(\frac{\boldsymbol{Q}\left(\boldsymbol{E}\boldsymbol{K}\right)^T}{\sqrt{d}}\right)\boldsymbol{F}\boldsymbol{V}, \tag{34}$$

where $\boldsymbol{E}, \boldsymbol{F} \in \mathbb{R}^{n\times k}$ are linear projection matrices and $k \ll n$. We can first prune $\boldsymbol{E}$ and $\boldsymbol{F}$ along with other weight matrices to have 1:2 or 2:4 sparsity offline following Mishra et al. (2021). Then we compute $\boldsymbol{E}\boldsymbol{K}$ and $\boldsymbol{F}\boldsymbol{V}$ with Sparse Matrix-Matrix multiplication. Next, we multiply $\boldsymbol{Q}$ and $(\boldsymbol{E}\boldsymbol{K})^T$ and the result is pruned to 50% structured fine-grained sparsity on the fly. After applying softmax to the nonzeros, we multiply it with $\boldsymbol{F}\boldsymbol{V}$.

As these methods require training from scratch for each task, we leave them for future work.