공학박사 학위논문

Accelerating Transformer-Based Model Inference using Efficient Matrix Multiplications on GPUs

GPU에서 효율적인 행렬 곱셈을 사용하여 Transformer 기반 모델 추론 가속화 제안

2023년 8월

서울대학교 융합과학기술대학원
융합과학부 지능형융합시스템전공
이 해 룡

Accelerating Transformer-Based Model Inference using Efficient Matrix Multiplications on GPUs

지도교수 안 정 호

이 논문을 공학박사 학위논문으로 제출함

2023년 07월

서울대학교 융합과학기술대학원

융합과학부 지능형융합시스템전공

이 해 룡

이해룡의 공학박사 학위 논문을 인준함

2023년 07월

Abstract

Accelerating Transformer-Based Model Inference using Efficient Matrix Multiplications on GPUs

Hailong Li
Intelligence Systems
Department of Transdisciplinary Studies
The Graduate School
Seoul National University

Transformer-based models have become the backbone of many state-of-the-art natural language processing (NLP) and computer vision tasks. As existing powerful models become large, enabling the models to learn and represent complex data relationships. Additionally, increasing the input sequence can be an effective way to improve performance for challenging real-world tasks. However, high inference cost hinders the use of powerful transformers because of large memory footprint, quadratic complexity with input sequence length in attention layers, and inefficient kernel operations.

In this thesis, we propose Transformer optimization methods to reduce inference costs in various scenarios, depending on the model size, input sequence length, and batch size. First, we propose Multigrain, an optimization method for scenarios where the input length (L_{in}) is significantly greater than the hidden dimension (D_h). Existing sparse attention techniques can effectively reduce computation and memory footprints in long input sequences; however, they are inefficiently processed on GPUs and still account for the majority of the execution time. Multigrain takes into account the sparse patterns of sparse attention, processing the coarse-grained part with a coarse-grained kernel using high-performance tensor cores and the fine-grained part with a fine-grained kernel using CUDA cores, respectively. As a result, Multigrain achieves a 2.07× end-to-end speedup over DeepSpeed when running Longformer inference.

Second, we propose a tiled singular value decomposition (TSVD) method to reduce inference costs in scenarios where L_{in} is similar to or smaller than D_h. TSVD is a technique that divides a matrix into tiles, performs singular value decomposition (SVD) on each tile, and compresses the matrix using low-rank approximation. By performing matrix multiplication, the fundamental operation of attention layers and feed-forward layers in Transformer models, using low-rank approximation-based TSVD-matmul, memory footprint and computation can be reduced, significantly lowering inference costs. Consequently, when compressing matrices by 2 to 8×, TSVD-based matrix multiplication is 1.02 to 2.26× faster than the uncompressed matrix multiplication. However, when applying TSVD to models, the execution time is reduced, but there is a trade-off in decreased accuracy.

To address this issue, we propose TSVD-common, a parameter-efficient fine-tuning method based on TSVD. TSVD-common shares one of the submatrices decomposed by SVD in each tile across all tiles and fine-tunes only the common submatrix during training. As a result, TSVD-common improves accuracy by approximately 2% even when compressing the GPT-2 model by 2 or 4× in E2E NLG tasks, compared to full fine-tuning without compression.

keywords: Transformer acceleration, Model optimization, Sparse attention, Parameter efficient fine-tuning, Model compression

student number: 2017-37405

Contents

List of Tables

List of Figures

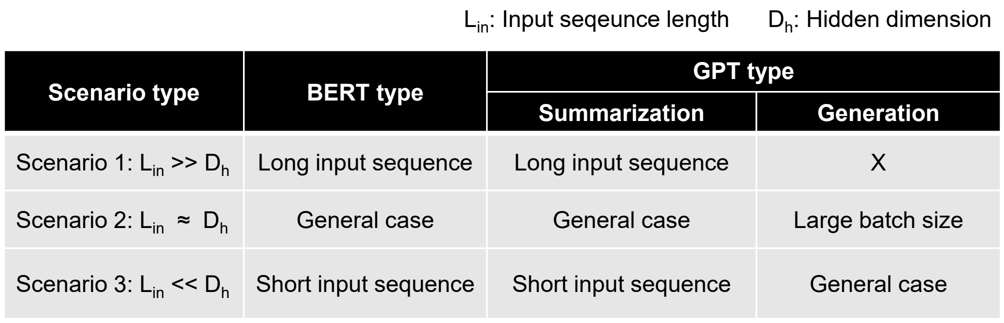
Chapter 1

Introduction

The advent of Transformer-based models has catalyzed a revolution in the realms of artificial intelligence and machine learning, particularly in natural language processing (NLP) [5,15,31,52,61,65] and computer vision [17,35]. These models, due to the power of their Transformer-based structure, can understand and represent complex data relationships by capturing long-range dependencies within the data. Furthermore, expanding the size of these models and the length of their input sequences has shown to significantly enhance their performance, particularly in the execution of complex, real-world tasks [4,5,28,50,67]. These advancements have unlocked the potential for more accurate language translations, more effective sentiment analysis, and more nuanced image recognition, among other applications.

However, deploying Transformer-based models presents certain difficulties. As model sizes increase and input sequences lengthen, inference costs rise proportionately. This is due to a number of factors, including an increased memory footprint, a quadratic complexity associated with input sequence length in attention layers, and the inefficiencies in kernel operations [50]. These challenges act as substantial hurdles, hindering the broader usage of powerful Transformer

This chapter is based on [32].
"A Slice and Dice Approach to Accelerate Compound Sparse Attention on GPU" ©2022 by Hailong Li, Jaewan Choi, and Jung Ho Ahn is licensed under CC BY 4.0. https://doi.org/10.1109/IISWC55918.2022.00019.


L_{in}: Input seqeunce length D_h: Hidden dimension

Scenario type	BERT type	GPT type	
		Summarization	Generation
Scenario 1: $L_{in} >> D_h$	Long input sequence	Long input sequence	X
Scenario 2: $L_{in} \approx D_h$	General case	General case	Large batch size
Scenario 3: $L_{in} << D_h$	Short input sequence	Short input sequence	General case

Figure 1.1: Various Transformer inference scenarios categorized based on factors such as the model size, input sequence length, and batch size.

models in numerous applications where they could be beneficial.

In this dissertation, we propose Transformer optimization methods to reduce inference costs on graphics processing units (GPUs), which are known for their high-performance capabilities in handling parallel operations. We propose two main strategies for various inference scenarios (see Fig. 1.1), and that these scenarios are categorized according to the model size, input sequence length, and batch size. The first, Multigrain, is designed to optimize a scenario where the input length (L_{in}) significantly exceeds the hidden dimension (D_h), taking advantage of the unique characteristics of sparse attention patterns. Our second proposal, the tiled singular value decomposition (TSVD) method, is devised for scenarios where L_{in} is similar to or smaller than D_h, and focuses on matrix compression to reduce inference costs. Recognizing the potential accuracy trade-off with the inference costs, we also introduce TSVD-common, a parameter efficient fine-tuning method that mitigates this issue.

1.1 Accelerating Sparse-Attention-Based Transformer Inference

In a Transformer-based model, the attention mechanism plays a crucial role in the model's success. The attention mechanism relates different positions of an input sequence to capture contextual information from the entire sequence, representing the long-distance dependency of the model.

Recently, processing long sequences has been gaining interest in NLP research [3, 56, 57]. For example, scientific literature has 1K − 10K words or even longer in a typical document, and digital humanity books can easily exceed 1K words. Therefore, existing models must deal with long sequences to understand long documents for the tasks such as document classification, multi-hop QA, and reading comprehension. However, early-stage Transformer-based models exhibit unsatisfying accuracies in long-sequence tasks compared to short-sequence tasks [55, 56]. The existing studies [13, 15] segment or shorten long sequences into short sequences during training, which can induce information loss due to the data loss [4]. To tackle such information loss, [4, 67] merge short, segmented sequences into one single input sequence to increase the contextual representation. As the sequence length increases, the accuracy of the model is improved.

Long-sequence processing has shortcomings in that its attention operations become expensive as the computation and memory footprint sizes are proportional to L_{in}^2 when the input length is L_{in}. For $L_{in} = 4096$, BERT-large [15] requires a memory size of 64GB, which is equipped only in the most expensive graphics processing units (GPUs). Considering that attention operations have quadratic space and time complexities, attention operations on long input sequences are limited by existing hardware resources.

Recently, sparse Transformer models based on sparse attention (SA) have been actively researched to address these issues. In particular, they perform SA with pre-defined sparse patterns using inductive biases, which reflect the additional assumption of language- and image-data features for accurate prediction. SA exhibits linear complexity, reducing computation and memory footprints. However, SA is processed inefficiently on existing GPUs during inference because the locality of the pre-defined, compound-sparse patterns, which combine multiple atomic sparse patterns,[1] are not considered for processing SA.

More models adopt a compound SA, with the compound-sparse-pattern-based SA as the main operation. However, such models only treat the compound SA using either the coarse-grained method based on a blocked sparse format (e.g., block coordinate format (BCOO) and block compressed row format (BSR)) or the fine-grained method based on an element-wise sparse format (e.g., compressed sparse row format (CSR), coordinate format (COO), and compressed sparse column format (CSC)). Only utilizing one of the two methods causes unnecessary computation and memory access and also causes poor data reuse during inference. Also, it is challenging to utilize the high-performance tensor cores in the fine-grained method [9, 34]. From Fig. 1.2, SA takes up significant execution time (73.8% and 52.9% in Longformer [4] and QDS-Transformer [27], respectively). Thus, the compound SA primarily contributes to the end-to-end execution time of the sparse Transformer model during inference.

[1]The atomic sparse pattern means a single sparse pattern in a compound-sparse pattern having various sparse patterns.

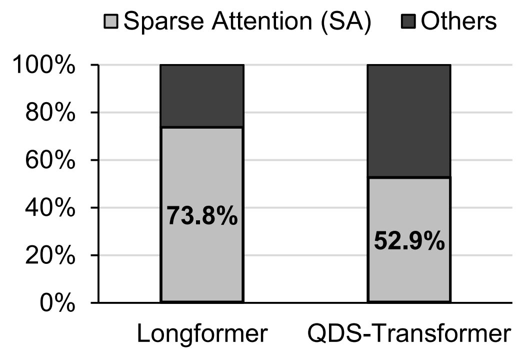


Figure 1.2: Execution time breakdown of Longformer, QDS-Transformer on an A100 GPU. Detailed model configurations and tasks are described in Section 3.2.1.

In this paper, we propose Multigrain, a compound processing method that accelerates the compound SA on GPUs. We categorize the atomic sparse patterns in the compound sparse patterns into coarse-grained and fine-grained parts by considering the locality of each pattern. Then, we accelerate the compound SA by performing coarse-grained and fine-grained parts with the corresponding kernels separately. To achieve such objectives, we design a customized coarse-grained kernel for the coarse-grained part that utilizes high-performance tensor cores with enhanced data reuse. We use a fine-grained kernel based on an optimized version of the Sputnik library[2] [20] for the fine-grained part to reduce unnecessary computation and memory access. We execute the coarse-grained and fine-grained kernels using multi-stream [36]; hence they can run concurrently in different SMs to utilize the hardware resources better.

Multigrain outperformed the latest sparse Transformer models such as Longformer and QDS-Transformer up to 2.08× and 1.68× on the latest GPUs (A100

[2]The optimized version of Sputnik used by our fine-grained method will be referred to as Sputnik in the following sections.

and RTX3090). We also evaluated our proposed method in the compound SA with various sparse patterns. We achieved $1.73\times - 2.34\times$, $5.06\times - 12.63\times$, and $1.79\times - 3.04\times$ speedup in the sampled dense-dense matrix multiplication (SD-DMM), sparse softmax (SpSoftmax), and sparse matrix-matrix multiplication (SpMM), respectively, over Triton [58], which only uses the coarse-grained method. Moreover, we achieved $1.34\times - 5.81\times$, $1.26\times - 2.82\times$, and $1.23\times - 5.24\times$ speedup compared to Sputnik, which only uses the fine-grained method.

1.2 Accelerating Transformer Inference using Low-Rank Approximation

Transformer-based models have become the backbone of many state-of-the-art natural language processing (NLP) tasks, including large language models such as GPT3 [5] and GPT4 [45]. These models employ a self-attention mechanism that allows them to capture long-range dependencies in text, leading to improved performance in various NLP tasks [15, 51, 61]. Matrix multiplication plays a crucial role in the Transformer models [10, 32, 33, 48]. It accounts for most of the execution time because key operations such as self-attention and feed-forward layers heavily rely on matrix multiplications.

Matrix multiplications often have different computational characteristics with respect to the input sequence length (L_{in}) and hidden dimension (D_h) in the Transformer-based models. When L_{in} is similar to or larger than D_h, the operation is compute-bound, which means that reducing the computation of the operation is crucial for decreasing the execution time. When $L_{in} \ll D_h$, the operation is memory-bound, which means that reducing the data size of the input/output matrices can lead to a decrease in execution time [29].

Singular value decomposition (SVD)-based matrix multiplication can accelerate matrix multiplication by reducing the complexity of computation and memory footprint through rank size reduction [47]. Such techniques can influence the result and the execution time of the matrix multiplication depending on the rank size. Compressing Transformer-based models using SVD approach can degrade the model quality, leading to a drop in accuracy [25] even after fine-tuning, due to the difficulty in preserving important information. Moreover, in small models (e.g., GPT2-M [51] and BERT-large [15]), the computing resources of the GPU are not fully utilized, leading to no speedup even at high degrees of compression. Therefore, we propose tiled SVD (TSVD, see Fig. 4.1) approach to mitigate the decrease in accuracy of the Transformer-based model while still achieving speedup.

1.3 Research Contributions

- We categorize the Transformer inferences into three scenarios based on model size, input sequence length, and batch size (see Fig. 1.1).
- We perform an analysis of the computational characteristics of the operations in the aforementioned inference scenarios.
- In cases where L_{in} is much greater than D_h, we propose a Multigrain solution to accelerate the sparse attention layer.
- When L_{in} is less than or similar to D_h, we propose a fast matrix multiplication method based on TSVD with low-rank approximation.

- We design a customized GPU kernel to accelerate the TSVD-based matrix multiplication's speed on a GPU.
- Additionally, we optimize the TSVD-based matrix multiplication using the stream-K method [46].
- We propose a parameter-efficient fine-tuning method, TSVD-common, to improve accuracy.

1.4 Outline

The organization of this dissertation is as follows. Chapter 2: Background provides an understanding of the Transformer model, Transformer inference, sparse attention-based Transformers, singular value decomposition, parameter efficient fine-tuning (PEFT), and graphics processing unit (GPU). Chapter 3: "Multigrain ($L_{in} \gg D_h$)" introduces the Multigrain mechanism, elaborates its GPU kernels, and evaluates its effects on Sparse Transformer and Compound Sparse Attention, followed by a review of related work. Chapter 4: "TSVD ($L_{in} \leq D_h$)" details the tiled singular value decomposition (TSVD), its GPU Kernel, its role in parameter efficient fine-tuning of low-rank approximation, and compares its performance with conventional SVD. Related work in this area is also reviewed. Chapter 5: Conclusion summarizes the research, revisits its contributions, and speculates on the future of the field.

Chapter 2

Background

2.1 Transformer Model

2.1.1 Transformer architecture

The advent of Transformer models brought a substantial shift in the natural language processing (NLP) and computer vision (CV) domains. Distinguished from their precursors, Transformer models operate without recurrence, enabling greater parallelization.

A vanilla Transformer [61] architecture is composed of an encoder-decoder structure where each part is a stack of identical blocks. Over time, Transformer variant models such as those in the BERT [4, 15, 30] and GPT [2, 5, 45, 59] families have demonstrated superior performance on a variety of tasks.

The BERT family of models, which are comprised exclusively of the encoder part of the vanilla Transformer model, are auto-encoding models that allow each token to attend to all other tokens in its context, regardless of their positions. This enables these models to better capture the contextual relationships among tokens. As a result, these models excel at natural language understand-

This chapter is based on [32].
"A Slice and Dice Approach to Accelerate Compound Sparse Attention on GPU" ©2022 by Hailong Li, Jaewan Choi, and Jung Ho Ahn is licensed under CC BY 4.0. https://doi.org/10.1109/IISWC55918.2022.00019.

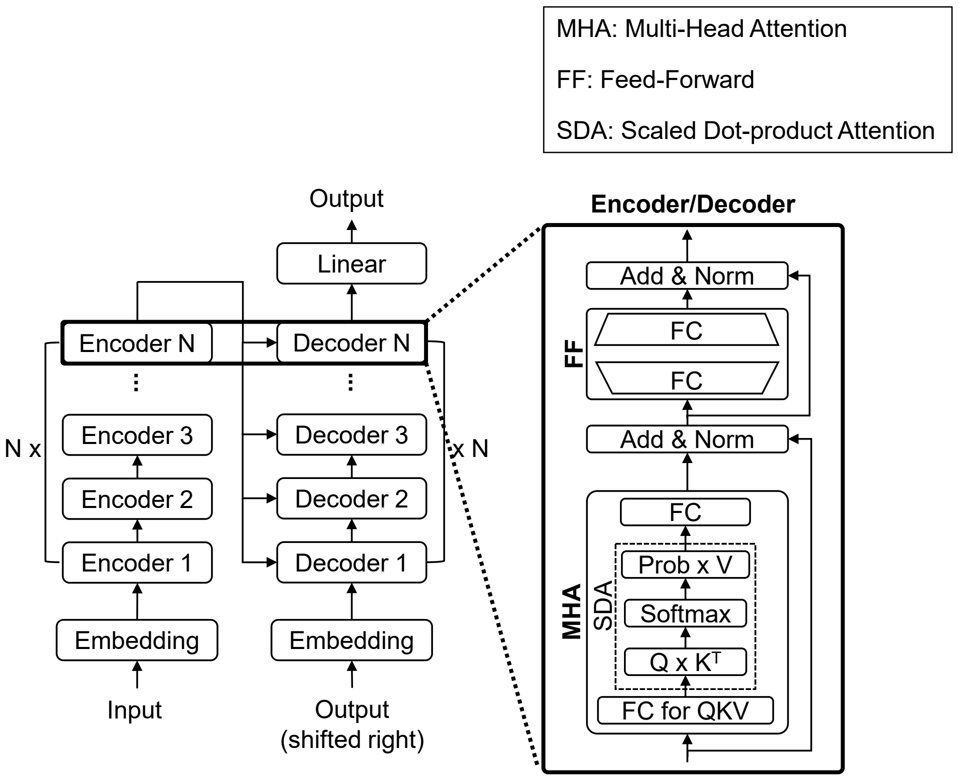


Figure 2.1: Transformer architecture

ing (NLU) tasks such as question answering, sentiment analysis, and reading comprehension.

On the other hand, the GPT family of models, based on the decoder part of the Transformer, are auto-regressive models that predict the next token according to previous tokens. These models exhibit superior generative abilities, making them well-suited for natural language generation (NLG) tasks such as text generation, machine translation, and summarization.

The basic block (e.g., encoder and decoder) in Transformer-based models primarily consists of a multi-head attention layer (MHA) and a feed-forward layer (FF) (see Fig. 2.1). Layer normalization and residual connections are utilized between the MHA and FF to stabilize learning and allow for deeper models.

The MHA allows the model to focus on different tokens in the input sequence, thereby enabling a better understanding of the context. In the first encoder/decoder, the MHA receives hidden states (L_{in}, D_h) generated by the embedding layer and produces an output matrix through the following procedure. In subsequent encoders/decoders, the MHA receives hidden states from the previous FF. Here, L_{in} denotes the input sequence length, while D_h represents the hidden vector size of a single token.

Firstly, the MHA utilizes the hidden state as input to prepare the queries (Q), keys (K), and values (V) through fully connected layer (FC). Then, Q, K, and V are split into multiple heads, allowing the attention process to run multiple times in parallel. Secondly, these are used in scaled dot-product attention (SDA) for each head. The SDA computes the dot product of Q and the transpose of K (K^T) to generate attention scores that represent the level of focus for each element in the sequence. The attention scores are then normalized using a softmax function to obtain attention probabilities ($Prob$). These attention probabilities are then used to compute a weighted sum of V. The output matrices from each head are subsequently concatenated. Thirdly, the concatenated results are passed through another FC. The output matrix of the MHA is then subjected to layer normalization, and the results are added to the input hidden states of the MHA (see Fig. 2.1).

The FF following the layer normalization consists of two FC with an activation layer (e.g., ReLU) in between. This structure enlarges the hidden dimension of the input hidden states, then reduces it back to the original dimension. As a result, the model can capture patterns that occur at any position in the sequence. The first FC enlarges the hidden dimension of the input hidden states from D_h

to $4 * D_h$, and the second FC reduces the dimension back to the original D_h.

Recently emerged Transformer–based models still maintain this architecture. However, as models grow larger and to address the limitations of the attention mechanism, various Transformer variants have appeared. In particular, there are many large models that increase the dimension D_h and the number of encoders/ decoders. These models show performance surpassing humans in various tasks. It leads to trigger a new boom in artificial intelligence. In the following section, we show the Transformer model trends.

2.1.2 Transformer model trends

The field of NLP has been witnessing significant advancements, primarily driven by the use of Transformer models. The most dominant strategy to improve performance in these models has been the increase in model size [28]. The trend of increasing model size is particularly evident in the development of large language models (LLMs) that followed GPT–2 [51]. These models are characterized by their ability to demonstrate few– or even zero–shot learning capabilities when pretrained on large datasets. For instance, GPT–J is an LLM with 6 billion parameters and trained on 400 billion tokens. It was followed by GPT–3 [5] from OpenAI, a family of decoder–only models, the largest of which has 175 billion parameters and is trained on 300 billion tokens. Recently, PaLM2 [2] was released from Google, the largest model in this family having 300 billion parameters and trained on 1 tillion tokens in various languages.

In tandem with the trend of increasing model size, there has been a notable focus on making attention mechanisms more efficient, especially for processing documents with longer sequence lengths. Sparse attention has emerged as a piv–

otal trend in Transformer models. This technique has been adopted by models such as Longformer [4] and BigBird [67], which have introduced explicit sparse bias in their self-attention mechanisms. This innovation has effectively reduced the computational complexity from $O(L_{in}^2)$ to $O(L_{in})$, making these models more efficient. For instance, Longformer from AllenAI employs a combination of local attention (attention only calculated from a fixed window size around each token) and global attention (only for specific task tokens like $[CLS]$ for classification) to create sparse attention scores instead of full attention scores. The BigBird [67] model, proposed by Google, employs a unique sparse attention mechanism that consists of three main components: local, global, and random attention. Local and global attention are similar to Longformer's, except that BigBird's are composed as blocks. In addition to local and global attention, the model also includes random attention. The model starts with a sliding window on the block, then a random subset of all connections is replaced with a random connection, while retaining the other local connections.

These sparse attention mechanisms reduce the computational and memory requirement from quadratic to linear in the sequence length, thereby enabling the handling of longer sequences.

2.2 Transformer Inference

The inference of BERT family models, as shown in the left of Fig. 2.2, processes multiple input tokens simultaneously. For example, when preprocessing the sentence "I like a cat" with a tokenizer and passing it through a embedding layer, it generates a set of embedding vectors, which is a (L_{in}, D_h) hidden states. Here,

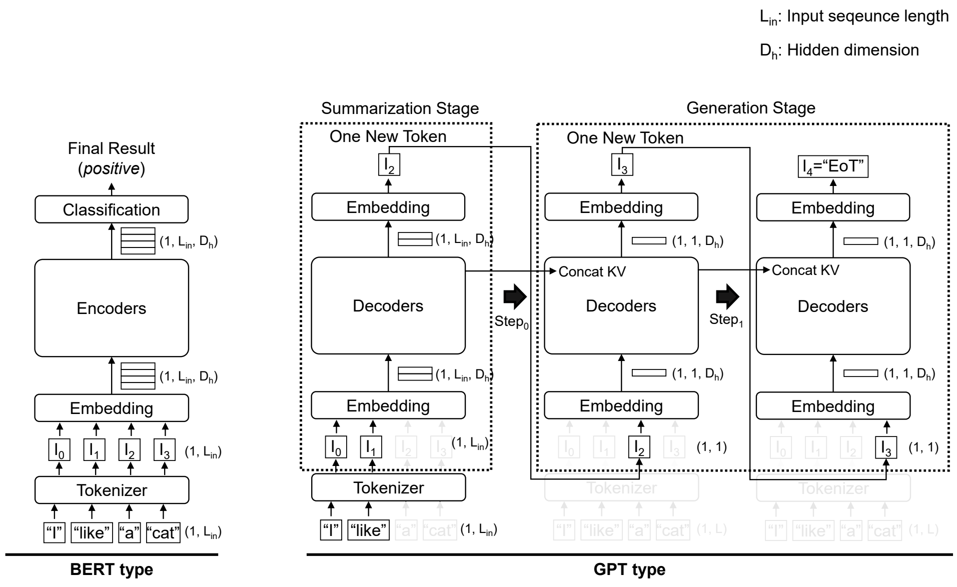


Figure 2.2: Transformer inference with regard to BERT and GPT2.

L_{in} is the input sequence length and D_h is the size of the embedding vector and also the size of the hidden dimension. This hidden states is passed to the encoder to perform the MHA and FF operations we described earlier. At last, it carries out the last layer (e.g., classification layer in the Fig. 2.2) to obtain the final result. Depending on the task, the last layer is different.

GPT family models, unlike BERT family, are usually divided into a summarization stage and a generation stage. The summarization stage, similar to the BERT type, processes multiple input tokens at once. However, in the Generation stage, it processes token-by-token. For instance, in the Generation stage (see the right graph in Fig. 2.2), it first takes one new token generated from the summarization stage as input. This one input token, denoted as I_2 in the figure, is converted into an embedding vector through embedding layer and then processed by the decoders. During the decoding process, matrix multiplication in the SDA is carried out in the "Q by K^T, $Prob$ by V" operations, which include the keys and values up to the previous step. To reduce redundant calculations, the previously calculated keys and values are reused. This is often referred to as ”storing in the KV cache”. The hidden vector generated through the decoders is then converted back into a new token via embedding layer. Text generation continues until an end of token (EoT) index is obtained.

However, high inference cost hinders the use of powerful Transformers for real-world tasks at scale [50]. This cost is often driven by large model size, long input sequences, and inefficient kernel operations. First, large models have a high memory footprint due to both the trained model parameters and the transient state needed during decoding. The model parameters usually do not fit in the memory of a single accelerator chip. Furthermore, key and value tensors of each layer, also known as the KV cache, must be stored in memory during the duration of decoding. This leads to a large amount of memory traffic and a high total memory bandwidth requirement, particularly to meet certain latency targets. Larger models need to be partitioned across many accelerator chips to fit in memory, but this introduces chip-to-chip communication costs. Second, the inference cost from the attention mechanism in large models scales quadratically with the input sequence length. The KV cache is unique for each sequence in the batch, meaning that it grows in size with longer sequences and larger batch sizes. This places a higher demand on memory capacity and bandwidth costs. For instance, for a 500B+ model with multihead attention and a context length of 2048 and a batch size of 512, the KV cache totals 3TB. Third, Sparse attention mechanisms, which are designed to reduce the computational and memory burden of attention operations in Transformer models, come with their own challenges. Sparse attention operations are not as efficiently supported on cur-

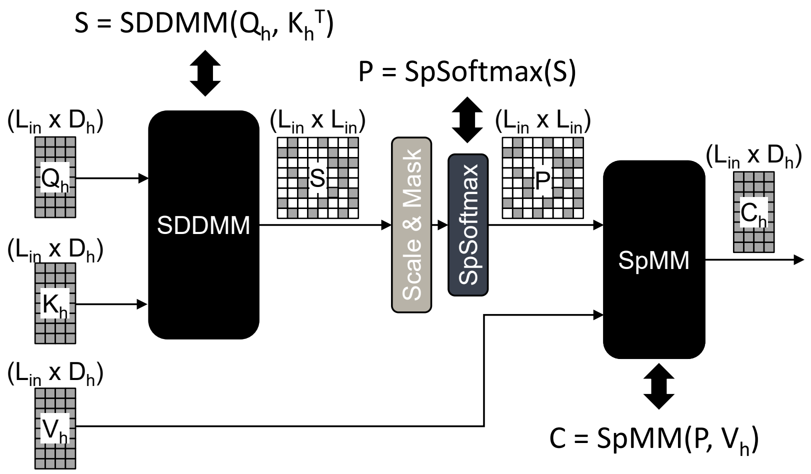


Figure 2.3: The sparse attention of a single head in a sparse Transformer. S and P are sparse matrices, and the other matrices are dense. L_{in} refers to the input sequence length, and D_h represents the head dimension of a single head.

rent hardware as dense operations. Modern hardware accelerators like GPUs and TPUs are designed to perform dense matrix operations efficiently, but they may not handle sparse operations as well. This mismatch can lead to inefficient utilization of hardware resources, thus increasing the inference cost.

We categorize the Transformer inference according to the model size, input sequence length, and batch size (see Fig. 1.1). In this thesis, we propose Transformer optimization methods to reduce inference cost (e.g., low latency, high throughput, and minimum hardware requirement) for these inference scenarios.

2.3 Sparse Attention-Based Transformers

In a sparse Transformer, SA is performed with multiple heads in the same way as the multi-head attention [61] in a typical Transformer. In other words, in an input matrix of (L_{in}, D_h), a single row vector of $(1, D_h)$ is split by the number of heads, and the matrix of $L_{in} \times D_h$ generated by applying it to other row vectors is repeatedly applied in parallel with a single head SA. Here, L_{in} represents the input sequence length, D_h represents the head dimension of a single head, and D_h (i.e., $D_h \times$ the number of heads) represents the vector size for the entire head.

The SA for a single head sequentially performs sparse operations consisting of SDDMM, scaling and masking, SpSoftmax, and SpMM (see Fig. 2.3). It finally obtains the context (C_h) by a single head of query (Q_h), key (K_h), and value (V_h). Q_h, K_h, and V_h are dense matrices whose shapes are (L_in, D_h), being split by the number of heads in the query, key, and value of the entire head to perform SA for a single head. The query, key, and value of the entire head are hidden states, dense matrices in which hidden vectors with a vector size of D_h are stacked in L_{in} and are calculated by multiplying them with different weight matrices of (D_h, D_h). The hidden states refer to the input sequence consisting of L_{in} tokens, an (L_{in}, D_h) dense matrix, which is the output matrix of the embedding layer, the previous layer of SA.

We describe each sparse operation below: SDDMM is a sparse operation that multiplies two dense input matrices to obtain a sparse output matrix. The sparse output matrix is generated by loading and calculating only the portion of the non-zero output elements using the metadata of a sparse format. In SA, SDDMM is an operation that multiplies Q_h and K_h to obtain an attention score (S) that shows the relevance between tokens (a word or word piece, which is different from the tokenizing methods [54, 63]).

Scaling is an element-wise operation; a sparse matrix calculated by SD-DMM is multiplied by scaling factors (SF=$1/\sqrt{D_h}$). It alleviates the gradient vanishing problem, pushing the softmax function into regions with extremely

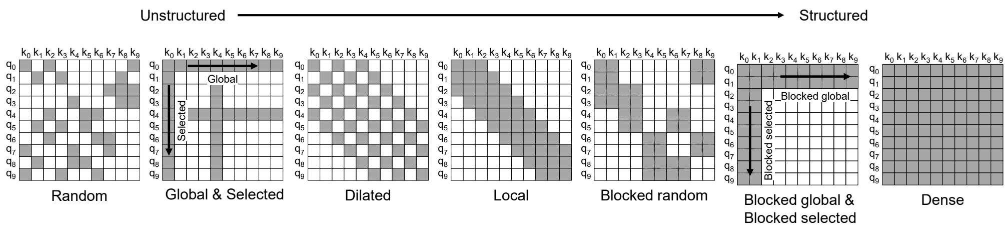


Figure 2.4: Various atomic sparse patterns in the existing sparse Transformers. From left to right, the patterns are listed from the least to the most structured pattern. In a sparse pattern, a row refers to the query vector and a column refers to the key vector. Gray squares refer to the valid elements resulting from an inner product of corresponding query and key vectors. Blank squares are invalid elements, and the corresponding query and key vectors are not used for the operations.

small gradients as the result of SDDMM growing large in magnitude [61].

Masking is an operation that masks out invalid elements in the mask matrix. If the input sequence length is smaller than the maximum sequence length the model can process, zero padding is conducted. Masking invalidates the zero-padded parts. In SA, it also masks out the invalid portion of the pre-defined sparsity. For invalidating zero-padded parts and the invalid portion, masking assigns an infinite negative value to them represented in the mask matrix.

SpSoftmax normalizes S to an attention probability (P) to mitigate the scale-up of S following SDDMM, scaling, and masking. SpSoftmax performs row-wise softmax only on non-zero elements of S, a sparse matrix represented by a sparse format. Similar to S, P is also an attention map expressing input tokens' importance.

SpMM is a sparse operation that multiplies the sparse input matrix represented by a sparse format with a dense input matrix to obtain a dense output matrix. The output matrix is calculated by loading only the non-zero elements in the input sparse matrix using the metadata of the sparse format and the corresponding element in the other input dense matrix. In SA, the left-hand side matrix is a sparse matrix P, the right-hand side matrix is a dense matrix V_h, and the output matrix is a dense matrix, referred to as context (C_h). Consequently, the context has a higher value for the more important hidden vector, i.e., it gives valuable weight to an important token; hence it can attend to important information.

2.4 Singular Value Decomposition (SVD)

Singular value decomposition (SVD) is a well-known technique for decomposing a matrix into three matrices, namely, the left singular matrix U, the singular values S, and the right singular matrix V. Given a matrix W of dimensions (K, N), SVD factorizes W into three matrices U, S, and V^T, such that $\mathrm{W} = \mathrm{U} * \mathrm{S} * \mathrm{V}^T$, where U is an (K, R) orthogonal matrix, S is an (R, R) diagonal matrix with non-negative values on the diagonal, and V^T is an (N, R) orthogonal matrix. To compute the low-rank approximation of W, we truncate the $\mathrm{U}, \mathrm{S}, \mathrm{V}^T$ matrices to a predetermined rank R', where $R' \ll min(K, N)$. This truncation results in $\mathrm{W} \approx \mathrm{U}_r * \mathrm{S}_r * \mathrm{V}_r^T$, where U_r, S_r and V_r^T represent the truncated matrices with shape of (K, R'), (R', R') and (N, R'), respectively.

2.4.1 SVD-Based Matrix Multiplication in Transformer models

Applying SVD-based low-rank approximation to the Transformer model enables model compression. By factorizing the weight matrices involved in the matrix multiplications of Transformer models, we can obtain smaller matrices, U_r, S_r, and V_r, decomposed by SVD. In addition, the S_r matrix with dimensions (R', R') can be multiplied by either the U_r matrix with dimensions (K, R') or the V_r matrix with dimensions (N, R'). This implies that the actual weight parameter only needs to retain two matrix parameters with dimensions equivalent to those of U_r and V_r^T. Therefore, if a weight matrix W with dimensions $(4096, 4096)$ is compressed by SVD, with R' being 1024, it can be decomposed into matrices U_r and V_r with dimensions $(4096, 1024)$ and $(1024, 4096)$, respectively. This makes it possible to achieve a compression ratio of $2:1$.

During inference, applying the SVD approach to matrix multiplications in Transformer models can reduce execution time by decreasing computation. Matrix multiplication accounts for the most of time in Transformer model [48], and the computation of this matrix multiplication based on SVD (SVD-matmul, i.e., $\mathrm{Y} = \mathrm{X} * \mathrm{U}_r * \mathrm{V}_r$) can be reduced as the rank size decreases. For example, if the dimensions of X and W are (M, K) and (K, N), respectively, and the dimensions of U_r and V_r obtained by factorizing W via SVD are (K, R') and (R', N), then the computation of the matrix multiplication changes from $2MKN$ to $2MR'(K+N)$. If $R' \ll min(K, N)$, the computation is reduced. For example, if $M = K = N = 4096$ and $R' = 1024$, the computation of the former is twice that of the latter. As R' decreases (i.e., higher compression ratio), the computation decreases even further. Therefore, we can accelerate Transformer models using SVD through the decrease in computation of matrix multiplication.

However, SVD-matmul (i.e., $\mathrm{Y} = \mathrm{X} * \mathrm{U}_r * \mathrm{V}_r$) is inefficient on GPUs. This is particularly noticeable in inference scenarios involving skewed matrix multiplication (i.e., including tall-and-skinny matrices) or small matrix multiplication (e.g., matrix shape dimensions under 1024). Without fusing the SVD-matmul (i.e., two GEMMs such as $\mathrm{P} = \mathrm{X}*\mathrm{U}_r, \mathrm{Y} = \mathrm{P}*\mathrm{V}_r$), there are two memory-bound matrix multiplications in the skewed matrix multiplication. It leads to a memory bandwidth bottleneck and could waste computing resources [8]. Additionally, if the partial matrix P is too large to be stored in the L2 cache, unnecessary memory accesses may occur. We can alleviate the above problems by fusing the SVD-matmul (i.e., launching $\mathrm{Y} = \mathrm{X} * \mathrm{U}_r * \mathrm{V}_r$ with one kernel), but this could lead to unnecessary data loading or multiplication. To obtain an output tile, repeatedly loading the whole U leads to increase the memory traffic and additional multiplication operations. This hinders performance because GPU compute and memory resources could be underutilized. To address the limitations of SVD-matmul and offer a more efficient solution for matrix multiplication, especially when implemented on GPUs, we propose a tiled singular value decomposition (TSVD) approach.

2.5 Parameter Efficient Fine-Tuning (PEFT)

Parameter-efficient fine-tuning (PEFT) [7, 16, 26] is a cutting-edge approach designed to mitigate the challenges that arise when fine-tuning large models (LMs). PEFT methods only fine-tune a small number of model parameters while freezing most parameters of the pre-trained models. This approach decreases the computational and storage costs and overcomes the issue of catastrophic forgetting observed during full fine-tuning of LMs. PEFT methods also generalize better to out-of-domain scenarios and are useful in low-data regimes. Furthermore, PEFT methods allow for portability, enabling users to get tiny checkpoints worth a few MBs compared to the large checkpoints of full fine-tuning. This is possible because the small trained weights from PEFT methods are added on top of the pre-trained models, so the same model can be used for multiple tasks by adding small weights without having to replace the entire model.

2.6 Graphics Processing Unit (GPU)

Modern NVIDIA GPU architecture includes arrays of streaming multiprocessors (SMs), and multiple SMs are connected to the L2 cache and device memory via the interconnect. In an SM, there are CUDA cores for arithmetic operations, special function units for transcendental functions, and tensor cores that support tensor operations to accelerate machine learning workloads. Register files (RFs), L1 cache, and shared memory (SMEM) store temporary data. L1 cache and SMEM are combined into a single memory block, which supports both types of memory accesses to provide bandwidth and capacity efficiently starting from the Volta architecture. The Ampere architecture starts to provide a new load-global-store-shared asynchronous copy instruction that saves SM internal bandwidth by bypassing the L1 cache and the RFs.

A GPU operates in a single instruction multiple thread manner, and a GPU kernel spawns numerous threads and processes them in parallel. These threads constitute a thread hierarchy consisting of thread, warp, thread block (TB), and grid. The consecutive 32 threads compose a warp, multiple warps compose a TB, and multiple TBs compose a grid. A kernel executes one or more grids in parallel. Modern GPUs can process up to four warps simultaneously within each SM, where an SM allocates tasks in units of TBs. One SM can allocate multiple

TBs if there is no capacity limit on the SMEM or RFs. When the operation for a single TB finishes, the next TB is assigned to the SM in a round-robin manner.

Besides CUDA cores, the Volta architecture starts to add tensor cores to the SM, drastically speeding up tensor operations. A tensor core performs one 4×4 matrix multiplication and accumulation in a single cycle, supporting FP16/FP32 mixed precision [11, 12].

NVIDIA GPUs introduce the concept of streams [36], a sequence of commands executed in order (i.e., possibly issued by different host threads) to manage concurrency by executing asynchronous commands. Multi-stream fully utilizes hardware resources by enabling concurrent execution of different streams.

Chapter 3

Multigrain

3.1 Contribution

We propose Multigrain, a new Transformer-specific optimization approach, to accelerate sparse operations on the compound-sparse patterns to overcome the problems of the aforementioned approaches. Multigrain can solve the problems of the existing solutions while maintaining the advantages of the coarse-grained and fine-grained approaches as much as possible and mitigating the disadvantages.

3.1.1 Multigrain mechanism

Sparse patterns (e.g., local, global, and selected) of compound-sparse-pattern based Transformer are determined offline by the model to be used, but the number and position of nonzeros are changed by the input data at every iteration. According to these environmental characteristics, Multigrain mechanism works with the following steps (see Fig. 3.1).

First, we classify and group sparse patterns into the coarse-grained and

This chapter is based on [32].
"A Slice and Dice Approach to Accelerate Compound Sparse Attention on GPU" ©2022 by Hailong Li, Jaewan Choi, and Jung Ho Ahn is licensed under CC BY 4.0. https://doi.org/10.1109/IISWC55918.2022.00019.

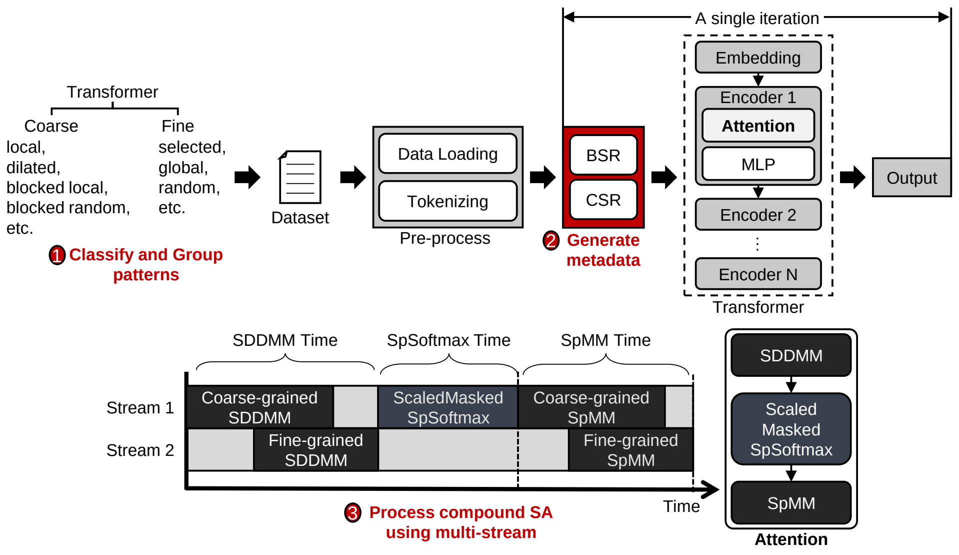


Figure 3.1: A Multigrain mechanism. 1) We categorize sparse patterns into coarse-grained and fine-grained patterns. 2) we generate the metadata (e.g., row offsets and column indices) for the compressed sparse matrices with the model configuration and positions of the special tokens before inferring the model. 3) In the compound SA, either SDDMM or SpMM is executed through both coarse-grained and fine-grained kernels, processed in parallel using multi-stream. SpSoftmax fused with scaling and masking is processed by a single kernel that can handle coarse-grained and fine-grained results represented by different sparse formats.

fine-grained parts according to the spatial locality before processing the sparse Transformer with Multigrain. We regard local, dilated, blocked local, blocked selected, blocked global, and blocked random patterns as the coarse-grained part and selected, global, and random patterns as the fine-grained part. We represent the coarse-grained part where nonzeros are structured and clustered as BSR format and the fine-grained part where nonzeros are unstructured and distributed as CSR format.

Second, when preprocessed input data are fed, we generate the BSR and CSR metadata. We use the window size in the model configuration to generate the BSR metadata for the coarse-grained part; and we use global indices (i.e., start and end tokens for a sentence, paragraph, and document) from the position of special tokens to generate the CSR metadata for the fine-grained part. However, the positions of special tokens change depending on each example in the datasets; we need to generate the CSR metadata at every iteration. In contrast, we generate the BSR metadata only once due to the fixed window size. We load the generated the metadata to the GPU for reuse in the repeated SA.

Third, we allocate the part represented in BSR format to the coarse-grained kernel and the part represented in CSR format to the fine-grained kernel. In the sparse GEMMs (i.e., SDDMM and SpMM), we process SDDMM/SpMM through both coarse-grained and fine-grained kernels in parallel using multi-stream.

We used a coarse-grained kernel we designed, which will be elaborated on in detail in Section 3.1.2. For the fine-grained kernels, we adopt Sputnik, which supports sparse GEMMs using CSR. We modify it to support half-precision (FP16) operations in SDDMM and optimize it further to achieve 3.3× to 6.2× speedups over the unmodified Sputnik. In SpSoftmax, we fused the scaling and masking operations with the sparse softmax using our customized sparse softmax kernel. In QDS-Transformer, we process SDDMM and SpMM of the local pattern part using our customized coarse-grained kernel using BSR metadata. Moreover, the selected pattern part is processed in the fine-grained kernel using CSR metadata.

We make exceptions for the special sparse patterns similar to the global pattern (a default pattern in the Longformer), whose parts can be processed independently in SpSoftmax and processed as dense operations in the sparse GEMMs. For those patterns, we perform SDDMM and SpMM for the special

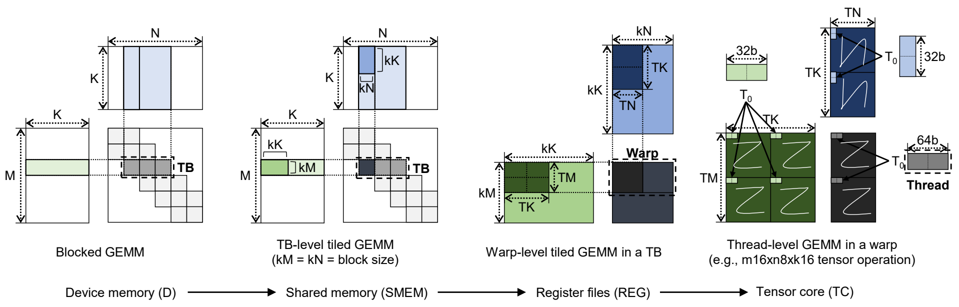


Figure 3.2: Hierarchical decomposition of SDDMM with the blocked row splitting scheme.

pattern parts using CUTLASS [1] kernels and perform SpSoftmax using the TensorRT's [60] softmax kernel because these libraries perform more efficiently than Sputnik.

3.1.2 Coarse-grained GPU kernels

We design two new coarse-grained kernels using BSR for the sparse GEMMs (i.e., SDDMM and SpMM), which handle coarse-grained pattern parts. Although we can use Triton for the coarse-grained part, it uses an inconsistent blocked sparse format between SDDMM and SpMM, requiring more memory spaces for storing the metadata of the different sparse formats. Also, Triton is not written in CUDA; hence it is difficult to process it with other kernels implemented by CUDA concurrently through multi-stream. Our design is on par with Triton in terms of the execution time, and outperforms Triton by 1.32× by 2.02×, particularly in batch processing (see Fig. 3.10).

Coarse-grained SDDMM kernel design: We design a new coarse-grained SDDMM kernel that embodies a blocked row-splitting scheme following the row-splitting scheme [64]. In the blocked row-splitting scheme, we assign each row block in the output matrix represented by BSR to a single TB. Fig. 3.2 shows the hierarchical decomposition of SDDMM using the blocked row-splitting scheme. We apply tiling to implement SDDMM efficiently by decomposing the blocked GEMM into a hierarchy of TB-level tiled GEMM, warp-level tiled GEMM, and thread-level tiled GEMM, similar to CUTLASS. LHS is the left-hand-side dense matrix, RHS is the right-hand-side dense matrix, and OUT is the sparse output matrix represented by BSR. The blocked GEMM, a part of SDDMM, is handled by a single TB. Each TB computes its output non-zero blocks (OUT blocks) in a row by iteratively loading the matrix blocks from RHS and LHS.

To exploit locality and parallelism, we partition the blocked GEMM into a TB-level tiled GEMM. We empirically set kM and kN, the sub-tile sizes of the column and row dimensions, as the block size of the non-zero blocks because the maximum number of TBs allocated to SM is limited depending on the memory resources actively used by a TB. A sub-tile block, a single non-zero block of the output matrix (OUT block), is obtained by accumulating the products of matrices by stepping through the K dimension (i.e., the row dimension in the LHS, and the column dimension in the RHS) in blocks. Then, the TB processes a series of different output non-zero blocks in the row sequentially to get the entire output row blocks. Thus, we reuse the LHS block repeatedly when processing OUT blocks sequentially in the TB-level tiled GEMMs, during which we load data from device memory (D) to $SMEM$ for data reuse.

In processing warp-leveled GEMM, we still follow the blocked row-splitting scheme. We assign each output row block on the warp level to a single warp. A warp performs a matrix multiply-accumulate (MMA) operation using the tensor core with an m16n8k16 shape supporting FP16 operations. We prevent

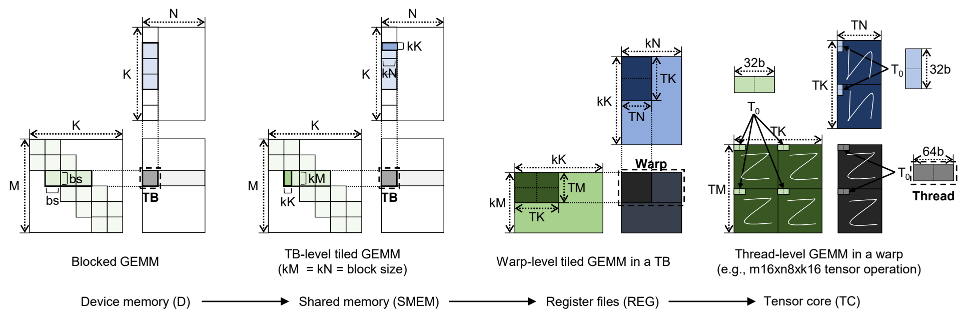


Figure 3.3: Hierarchical decomposition of SpMM with the blocked one-dimensional tiling scheme.

overflow by using an MMA instruction that supports FP32 for the data type of the output element. We split the warp-level LHS blocks and RHS blocks into kK dimensions (i.e., the row dimension of the warp-level LHS and the column dimension of the warp-level RHS, as shown in Fig. 3.2) to reuse registers (REG). If the warps inside a TB use too much of REG, we enforce SM to use fewer TBs and decrease the occupancy of GPU. It degrades the GPU performance because low occupancy always reduces the ability to hide latencies.

We use software pipelining (double buffering) to hide the latency of memory operations. The method feeds the output of each stage to its dependent stage during the next iteration and executes all stages of the GEMM hierachy in parallel within a loop. We eliminate the latency of the RHS loading from device memory to $SMEM$ by double buffering the tile size of $SMEM$ used by the RHS.

Coarse-grained SpMM kernel design: Our coarse-grained SpMM kernel uses a blocked one-dimensional (1D) tiling scheme. It follows the blocked row splitting scheme like our SDDMM, except that a single TB is not mapped to an entire output row block, but similar to the 1D tiling scheme [20], where we shard the output matrix into 1D tiles and map independent TBs to each tile.

We empirically set the output tile size the same as the non-zero block of BSR. Fig. 3.3 shows the hierarchical decomposition of SpMM using the blocked 1D tiling scheme, and we also apply tiling to implement SpMM efficiently, similar to our SDDMM kernel. LHS is a sparse input matrix represented by BSR, RHS is a dense input matrix, and OUT is a dense output matrix.

In a blocked GEMM, we accumulate the products of matrices by loading non-zero blocks of the LHS and the corresponding RHS and obtain the OUT blocks. However, if the number of non-zero blocks is large, a TB cannot load all LHS non-zero blocks at once due to the limited memory resources in the SMs. Therefore, we partition the blocked GEMM into the TB-level tiled GEMMs.

In the TB-level tiled GEMM, we accumulate the products of matrices by stepping through the non-zero blocks in a row of the sparse matrix to obtain the OUT blocks. We apply a tiling structure to reuse additional OUT blocks and allocate more TBs to SM. To process a single non-zero block, we split each non-zero block of LHS and the corresponding block of RHS into the K dimensions, and we load the slice of LHS and RHS to get the OUT block. Therefore, an OUT block can be reused by the number of non-zero blocks of LHS and the number of slices in a non-zero block. First, we store the split slice of the LHS block and the RHS block in $SMEM$ and then use the warp-level tiled GEMM. Second, $SMEM$ stores twice as much the slice of the LHS and RHS blocks. Similar to SDDMM, this is to use software pipelining to hide latency for data movement. The number of TBs that can be allocated in an SM is more limited by REG than by $SMEM$ because REG is generally smaller than $SMEM$ available for TB in an SM. The warp-level tiled GEMM follows the blocked row-splitting scheme, and we implemented the operations in the same way as SDDMM.

3.1.3 Compound sparse softmax GPU kernel

In SpSoftmax, we also design a new sparse softmax kernel. As opposed to SDDMM and SpMM, we use a single sparse softmax kernel to process the outputs of coarse-grained and fine-grained kernels altogether. It is difficult to obtain accurate softmax results with one type of SDDMM output if coarse-grained and fine-grained sparse patterns are in the same row as softmax sweeps all row elements (e.g., find the max or exponential sum). Prior to the sparse softmax operation, we process scaling and masking operations to reduce the memory access.

Before running the model, we invalidate the overlapped parts if the coarse-grained and fine-grained patterns are overlapped. It avoids inaccuracies from the softmax operations due to the overlapped fields. We use a mask matrix, an attention map where valid elements are represented as zeros and invalid elements are infinite negative values. The valid elements refer to the coarse-grained pattern because some non-zero blocks represented as BSR for the coarse-grained patterns, such as the local pattern, may be sparse. The invalid elements refer to the zero-padding portion to meet the maximum sequence length and overlapped parts between the coarse-grained and fine-grained patterns.

Our compound sparse softmax kernel follows the blocked row-splitting scheme. We assign a single TB to an entire output row block to perform a row-wise softmax operation. As the output row block appears in a combination of the non-zero blocks represented by BSR and the non-zero elements represented by CSR, the output row blocks processed by a single TB depend on the number of the non-zero blocks present in each row. We proceed with the following three steps to perform the safe softmax in each row [37]: First, the max-finding process searches for the maximum of non-zero elements. Second, in the exponential sum process, we subtract the maximum value from each element, exponent the differences, and sum the results. Due to the limited range of values representable in existing GPUs, the subtraction can prevent overflow or underflow during the exponent operations. Finally, the normalization process normalizes each exponent element executed in the previous process to obtain the final output.

In each step, the dataflow sweeps the row elements of the non-zero blocks in the BSR format and the non-zero elements in the CSR format. Taking the max-finding process as an example, we first sweep the non-zero blocks present in each row using the BSR metadata, and we find the maximum value among the ones held by each thread in the coarse-grained pattern part. Next, we sweep non-zero elements in each row using CSR metadata and find the maximum value among elements held by a thread in the fine-grained pattern part. Therefore, each thread holds the maximum value among the swept elements. We find the maximum element by comparing the elements between threads with each other through warp shuffling, which exchanges the register values between threads within the warp. As a result, each thread holds the maximum element among row elements.

However, we make an exception for the special patterns, which areglobal patterns that can perform softmax independently. If the compound sparse pattern includes the special pattern, we process the special pattern parts through the dense softmax kernel and use multi-stream to process with the compound

Table 3.1: Specifications of the GPUs used in the evaluation. *Peak rates are based on the GPU's base clock. **Recent GPU architectures combine L1 data cache and SMEM functionality into a single memory block.

	A100	RTX 3090
Memory Bandwidth (GB/s)	1,555	936.2
TFLOPS (FP16 CUDA core)*	42.3	29.3
TFLOPS (FP16 Tensor core)*	169	58
L1 D$ per SM (KB)**	192	128
L2 (MB) $	40	6

sparse softmax kernel for other pattern parts in parallel.

3.2 Evaluation

We evaluated the effectiveness of Multigrain with various batch sizes in the Longformer and QDS-Transformer, which are sparse Transformers based on the compound SA. To show the performance improvement in the region of interest, we evaluated Multigrain in the various compound patterns. They includes real workloads and synthetic workloads considering that the workloads will be applied to future models. Finally, we evaluated the performance improvement of the blocked row-splitting scheme. We compared our customized coarse-grained kernels to Triton at the sparse operation based on various coarse-grained patterns such as a local, blocked local, and blocked random pattern.

3.2.1 Experiment Setup

We evaluated inference speed for the sparse Transformer models using FP16 operations in real-world tasks. We use the Longformer and QDS-Transformer models, which achieved superior accuracy due to exploiting the compound SA.

Longformer employs local, selected, and global patterns and records SOTA scores on tasks such as question and answer (QA) and reading comprehension. QDS-Transformer utilizes local and selected patterns and records impressive accuracy on document ranking tasks. We used a large model of Longformer provided by HuggingFace [62] and a base, officially-release model of QDS-Transformer. We measured the end-to-end execution time of Longformer using the hotpotQA [66] dataset, and of QDS-Transformer using the Microsoft MAchine Reading Compensation (MS-MARCO) [41] dataset.

We run the PyTorch 1.8.2 framework [49] on two GPUs with different computational characteristics (A100 [43] and Geforce RTX 3090 [6]) running CUDA toolkit 11.3 [44]. Hardware specifications are shown in Table 3.1. The execution time and the off-chip memory accesses on GPUs are measured through NVIDIA Nsight Compute. We used a deep learning optimization library DeepSpeed (v0.5.1), released by Microsoft, that efficiently handles Transformer models. This library performs SA with OpenAI's Triton (v1.1.1) and processes it in the coarse-grained method. However, it suffers from excessive local memory accesses due to the register spill issue in SDDMM; hence we applied optimizations to SDDMM in our experiments.[1] We used Sputnik, which processes SDDMM and SpMM in the fine-grained method. We extended it for supporting FP16 and batched operations. Moreover, we optimized the SDDMM kernel using the row-splitting scheme instead of the 1D tiling scheme, which is the official Sputnik version.[2] In SDDMM, the 1D tiling scheme wastes the SM

[1]In SDDMM (a single batch, four multi-head, and 64 head dimensions), our optimized Triton was 6.24×, 6.23×, and 6.73× faster than the original version at the local, blocked local, and blocked random patterns, respectively.

[2]The SDDMM kernel optimized with the row-splitting scheme reduces execution time by 3.3× to 6.2× over the 1D tiling scheme.

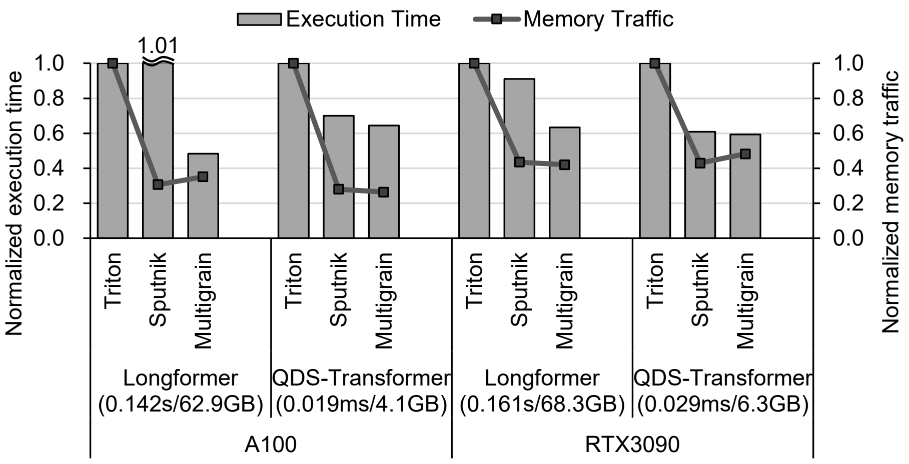


Figure 3.4: Execution time and memory traffic of Longformer-large and QDS-Transformer models by applying Triton, Sputnik, and Multigrain. We normalized the other results based on the Triton. The absolute execution cycle and memory traffic of each baseline is shown below each model.

resource because warps that do not perform operations cost extra TBs. Thus, it decreases the achieved active warps per SM and induces overhead from context switching in the warp scheduler, and the GPU performance degrades.

3.2.2 End-to-End latency on sparse Transformers

Multigrain accelerated Longformer and QDS-Transformer by optimizing compound SA during the inference. Fig. 3.4 shows the end-to-end execution time and memory traffic of Longformer and QDS-Transformer using Triton, Sputnik, and Multigrain on the GPUs (A100 and RTX3090) with different computational characteristics. In Longformer and QDS-Transformer, Multigrain reduced memory traffic by 2.84× and 3.78×, respectively, compared to Triton, while reducing execution time by 2.07× and 1.55×. Moreover, it reduced the execution time by 2.08× and 1.09×, respectively, with similar memory traffic compared to Sputnik.

Comparing the speedup in Longformer and QDS-Transformer, we observed that the degree of performance improvement varies because each model has a different ratio between the number of the sparse blocks and dense blocks due to the differences in the window size and the maximum sequence length. For example, the ratios between the number of sparse blocks and dense blocks are 1:3 and 2:1, respectively, when performing the sparse operation of a local pattern with a block size of 64 in Longformer and QDS-Transformer. The more low-sparsity sparse blocks there are, the more benefits the fine-grained kernel has over the coarse-grained kernel because the fine-grained kernel only operates on valid elements. Although the coarse-grained kernel improves performance using tensor cores in the fine-grained parts, the gains are negated by the unnecessary computation and memory accesses when processing sparse blocks with low sparsity. By contrast, the coarse-grained kernel performed better than the fine-grain kernel when performing operations with more dense blocks. It benefits from data reuse and high-performance tensor cores. Therefore, Multigrain has higher speedup in Longformer, where there are more dense blocks than QDS-Transformer. Moreover, Multigrain achieved more improvement in Triton than Sputnik for QDS-Transformer.

We evaluated the performance of Multigrain with DeepSpeed [53] and FasterTransformer [42], libraries that accelerate Transformer-based models (see Fig. 3.5). In the latest FasterTransformer (v5.3), the Longformer is faster than the DeepSpeed version. As a result, Multigrain demonstrates a speedup of 2.07× in DeepSpeed and 1.47× in FasterTransformer.

Multigrain exhibits improved performance as the batch size increases (see Fig. 3.6). The sparsity of the sparse patterns in SDDMM and SpMM is similar

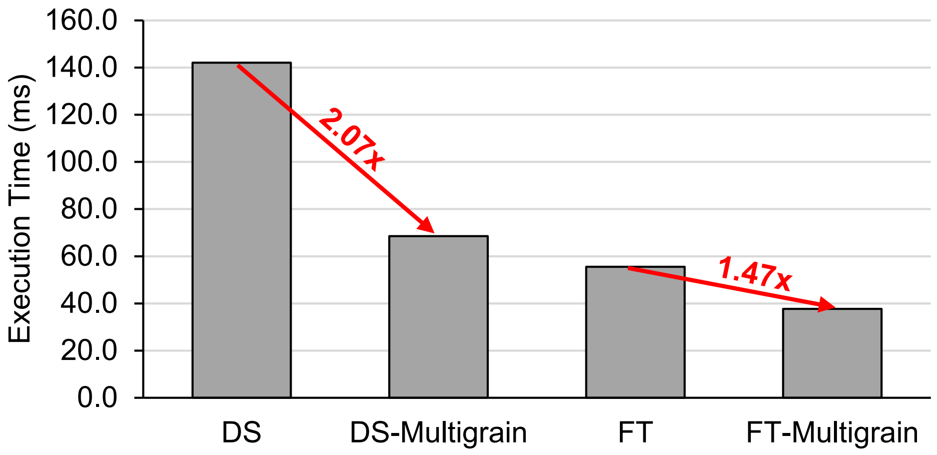


Figure 3.5: Execution time of Longformer-large at the DeepSpeed and Faster-Transformer libraries.

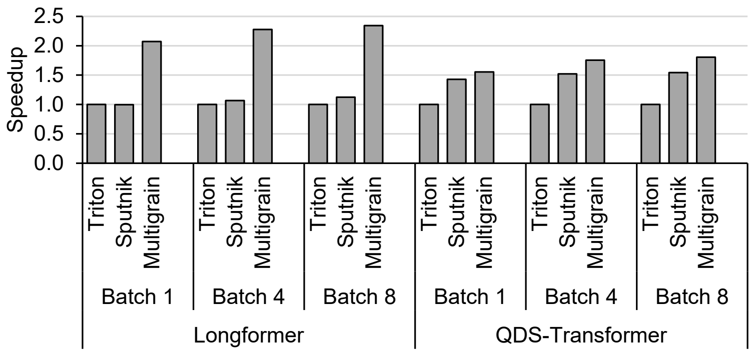


Figure 3.6: Speedup in Longformer-large and QDS-Transformer at the various batch sizes by applying Triton, Sputnik, and Multigrain.

for each batch. The local pattern beneficial for Triton remains consistent, while the global or selected pattern, which is advantageous for Sputnik, can vary based on the input sequence. Consequently, as the batch size increases, the limitations of Triton and Sputnik become more expanded. However, Multigrain effectively mitigates these shortcomings by fully utilizing GPU resources, leading to notable performance improvement. It shows up to 2.34× and 1.82× speedups compared to Triton and 2.13× and 1.17× speedups compared to Sputnik, respectively, in

Longformer and QDS-Transformer.

Multigrain achieved a similar degree of performance improvement for both RTX3090 and A100 by reducing memory traffic. However, compared to A100, RTX3090 experiences more performance degradation for the coarse-grained kernel using tensor cores compared to the fine-grained kernel using CUDA cores. It is because the peak floating-point operations per second (FLOPS) value of the tensor cores is reduced more than that of the CUDA cores, as shown in Table 3.1. Therefore, Sputnik showed a greater performance improvement than Triton on RTX3090. In Longformer and QDS-Transformer, it shows 1.10× and 1.64× speedup, respectively. Both coarse-grained and fine-grained kernels exist in Multigrain; hence their performances are determined by the GPU's computing abilities. Multigrain shows 1.58× and 1.44× speedups compared to Triton and Sputnik in Longformer, respectively, which shows less performance improvement than A100 due to the performance degradation of the tensor core. By contrast, in QDS-Transformer, Multigrain is 1.68× and 1.02× faster than Triton and Sputnik, respectively.

3.2.3 Speedup on the sparse attention

Multigrain improves performance for the compound sparse patterns besides the sparse patterns exploited in the existing sparse Transformers. In the following, we analyzed the results of sparse operations in the various sparse patterns, including existing compound-sparse and synthetic patterns.

Compound sparse GEMM

In the compound sparse GEMM (i.e., SDDMM and SpMM) of various compound-sparse patterns, Multigrain is faster than Sputnik and Triton (see in Fig. 3.7). Especially, Multigrain achieved more speedup if global patterns are included in the compound-sparse pattern. On the contrary, Multigrain showed limited performance improvement, if there are random patterns in the compound-sparse patterns.

In particular, Multigrain achieves a $2\times - 5\times$ speedup for the sparse operations of the compound-sparse patterns exhibiting the global pattern, where Multigrain improves the throughput of SDDMM up to $5.81\times$ and $2.02\times$, and SpMM up to $5.24\times$ and $2.13\times$, respectively, compared to Sputnik and Triton (see the last two patterns in Fig. 3.7). Sputnik causes load imbalance issues for SM, which degrades its performance when processing the sparse operations of such compound-sparse patterns. Sputnik maps an entire row of the output matrix to a TB because it processes SDDMM as a row-splitting scheme. This way, more elements that require operations are assigned to the TB while processing the global pattern. The achieved active warp per SM value drops as these TBs are allocated to the SM along with other TBs. Using the Nsight Compute, we measured the ratio of the achieved and theoretical occupancies, a metric representing the load imbalance. The smaller value of the metric, the more severe the imbalance. In the kernel that processes the sparse operations of the local, selected, and global patterns (L+S+G), the metric exhibited 61.2%, which was lower than that of the local and selected patterns (L+S, 89%), which occurred due to the global pattern part.

The load imbalance issues exist in SPMM as well, even though Sputnik per-

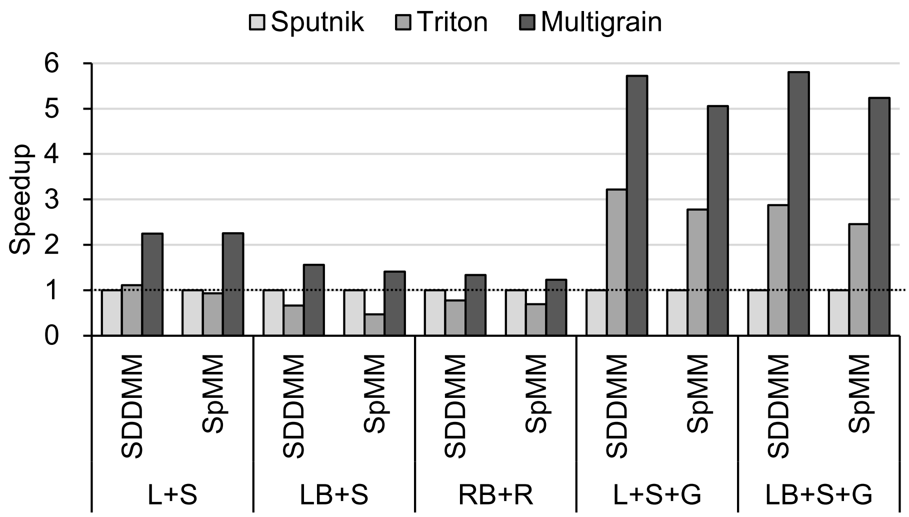


Figure 3.7: Speedup of Multigrain compared to Sputnik and Triton in the compound-sparse GEMM with various compound sparse patterns. (L: local, S: selected, G: global, R: random, LB: blocked local, and RB: blocked random sparse pattern). Compound-sparse GEMMs are sparse operations in the sparse attention (i.e., SDDMM and SpMM). The operations are processed with parameters such as 1 batch size, 4096 input sequence length, 4 multi-heads, 64 head dimensions, and 95% sparsity in each row.

forms SpMM by sharding the output into one-dimensional tiles and mapping independent TBs to each. In SpMM, the computation and data required to process by a TB depend on the sparse matrix. For Triton, the load imbalance issue that SDDMM and SpMM operations experience has been mitigated. While processing SDDMM, Triton does not suffer from load imbalances because each TB processes the same number of non-zero blocks represented by the blocked sparse format. In SpMM, Triton alleviates the load imbalance issue because the tile size processed by a TB is larger than Sputnik, and the operations perform quickly with high-performance tensor cores. Multigrain resolves this problem by processing the global pattern part, especially with the CUTLASS [1] kernel instead of the fine-grained method, considering the dense row units in the global patterns.

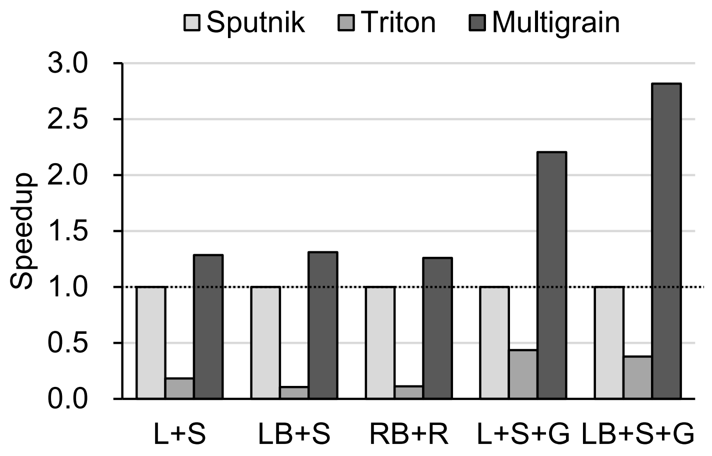


Figure 3.8: Speedup of our compound sparse softmax kernel (Multigrain) compared to Sputnik and Triton with various compound-sparse patterns in SpSoftmax on A100.

In the sparse GEMMs for the compound sparse patterns without a global pattern, our method is faster than Sputnik and Triton by $1.34\times - 2.25\times$ and $1.73\times - 2.34\times$ in SDDMM, and by $1.23\times - 2.25\times$ and $1.79\times - 3.04\times$ in SpMM (see Fig. 3.7). In the sparse operation of a blocked random and random (RB+R) pattern, the randomness leads to load imbalances, leading to a lower speedup than the other compound sparse patterns. Moreover, it is difficult to hide overhead due to not exploiting the thread-level parallelism when a kernel allocates fewer TBs that perform a large tile for small-sized matrix multiplications.

3.2.4 Speedup on the sparse softmax

SpSoftmax cannot be processed simultaneously by being divided into multiple parts according to locality differences due to the row-wise operations of the softmax. We designed a new compound-sparse softmax kernel that processes the coarse-grained and fine-grained output matrices, whose results are represented by BSR or CSR in the SDDMM, into a single kernel.

We compared our designed kernel with Sputnik and Triton to evaluate its performance; our kernel is faster than both up to 12.63× (see Fig. 3.8). In the former three compound sparse patterns without a global pattern, our customized kernel has $1.26\times - 1.31\times$ performance improvements over Sputnik, whereas it has $7.09\times - 12.63\times$ speedup over Triton. Using the fine-grained method, Sputnik sends more memory requests for loading and storing the non-zero elements. This is wasteful because the clustered non-zero elements can be referenced in blocks for local patterns and blocked local patterns. By contrast, the coarse-grained method can reduce the number of memory requests because it refers to the non-zero blocks through the metadata of a blocked sparse format. Memory requests dropped by up to 80% when we switched from Sputnik to Triton.

Still, Triton is significantly slower than Sputnik because Triton deals with its fine-grained pattern parts using the coarse-grained method, inducing unnecessary computation and memory access. When the fine-grained pattern part is processed in a coarse-grained method, Triton wastes operations on the invalid elements in the sparse blocks with low sparsity.

In the latter two compound sparse patterns with a global pattern (see Fig. 3.8), our customized kernel achieves $2.20\times - 2.82\times$ and $5.06\times - 7.48\times$ speedups compared to Sputnik and Triton, respectively. We processed the global pattern part separately using a dense kernel like sparse GEMMs. The global pattern has the characteristic that the entire row is dense; hence the operation of the global pattern part is not affected by the other patterns. It leads to performing the global pattern part separately and can alleviate the load imbalance issue in the end.

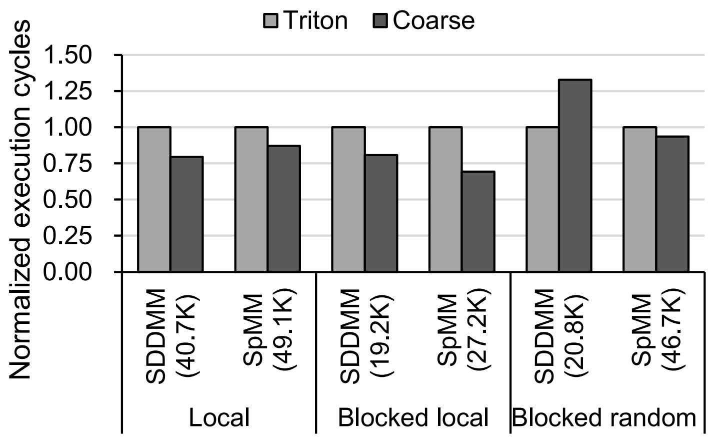


Figure 3.9: Execution cycles of SDDMM and SpMM (sequence length=4096, batch size=1, number of heads=4, window size=256, window block=3, block size=64, and sparsity=95%) by applying Triton and our customized coarse-grained kernel (Coarse). We normalize the values based on the execution cycles of Triton. The absolute execution cycle of each Triton is shown next to SDDMM and SpMM.

3.2.5 Speedup on the coarse-grained kernel

We implemented a coarse-grained kernel without using Triton to perform the coarse-grained method efficiently. We evaluated SDDMM and SpMM using the coarse-grained patterns, including local, blocked local, and blocked random patterns. We decided the parameters[3] of coarse-grained patterns based on Longformer and QDS-Transformer.

In the local or blocked local pattern, our coarse-grained kernel improves performance up to 1.26× and 1.24× in SDDMM, and 1.15× and 1.44× in SpMM compared to Triton (see Fig. 3.9). However, our coarse-grained kernel is 25% slower than Triton on SDDMM in the blocked random patterns. Our kernel suffers from the load imbalance issue, where non-zero blocks in each

[3]Window size is 256 in a local pattern, window block is 3 in a blocked local pattern, and the sparsity of the matrix is 95% in a blocked random pattern. Both blocked patterns have a block size of 64.

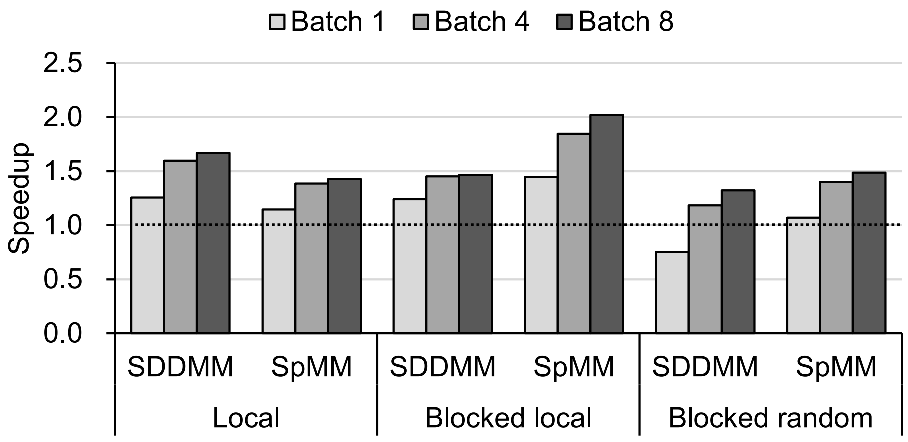


Figure 3.10: Speedup of our customized coarse-grained kernel (Coarse) compared to Triton with various batch sizes on A100.

row may differ in the blocked random pattern. Therefore, it induces a longer execution time because each TB is assigned to a row in the output matrix in our row splitting scheme. If the batch size is large, the overhead from load imbalances would be alleviated due to the increased number of TBs. In Fig. 3.10, our kernel for the blocked random patterns improves up to 1.32× in performance compared to Triton with four or eight batches. The performance of the other coarse-grained patterns improves as the batch size increases. The number of TBs is larger as the batch size increases. Hence the number of active warps per SM becomes larger, where SMs can exploit warp-level parallelism to reduce latency caused by the warp stall. In SpMM, our kernel performs better as batch size increases for the same reason. In the local pattern, blocked local pattern, and blocked random pattern, we achieve speedup up to 1.43×, 2.02×, and 1.49×.

3.3 Related Work

3.3.1 Coarse-grained methods

Many studies have optimized the sparse attention on GPUs by applying the blocked sparse format. DeepSpeed [53] uses OpenAI's Triton [58] to process the sparse attention. It is effective in the sparse operations for the coarse-grained patterns because it uses the blocked sparse format. However, as the sparse operations of the fine-grained patterns must be processed in a coarse-grained method, the method has a disadvantage in that the operations result in unnecessary computation and memory access.

NVIDIA introduced cuSPARSE library [40] that provides a set of APIs for sparse operations. The library provides the cusparseSpMM and cusparseSDDMM APIs with a practical speedup on 95% or higher degree of sparsity [9]. Among these APIs, an API can process an SpMM with a coarse-grained method using the blocked-ELL format. However, it must process a single batch sequentially for multiple batch operations. Furthermore, the library does not support SDDMM with a coarse-grained method, so it could not be applied to sparse attention.

There have been studies to process the sparse operations using dense libraries after reshaping matrices. BigBird proposed the blockify [67] scheme to process sparse operations of the blocked local pattern, and Longformer proposed sliding chunk [4] to process sparse operations of the local pattern. However, these methods require pre-processing and post-processing, which induces considerable memory copy overhead.

3.3.2 Fine-grained methods

To process sparse operations efficiently with irregular sparsity in SA, existing research has explored several ways to implement the fine-grained method. The Sputnik library [20] developed by Google accelerates sparse operations such as SDDMM, sparse softmax, and SpMM on GPU. Sputnik exploits the reusability of operands by using a one-dimensional tiling scheme to decompose the computation across processing elements. Sputnik also uses vector instructions on misaligned memory addresses and introduced an approach for load-balanced computations. However, high-performance tensor-core units are not used in the sparse operations of the coarse-grained patterns with a high spatial locality. Moreover, Sputnik does not support batched operations; it only supports FP32 operations for SDDMM.

The NVIDIA cuSPARSE library provides fine-grained kernel APIs for sparse operations using CSR or CSC. Similar to Sputnik, it does not use tensor cores, and thus, it has issues with low data reuse in the sparse operation of coarse-grained patterns. Starting from the Ampere and Hopper architecture GPUs, NVIDIA introduced sparse tensor cores [38] to accelerate the sparse operation of the 2:4 (50%) fine-grained structured patterns. NVIDIA provides a cuSPARSELt library to perform sparse GEMMs. As the sparsity has been reduced by half, the cuSPARSELt APIs reduce the execution time by half compared to the dense GEMM. However, the library only supports the 2:4 fine-grained structured sparse pattern, making it difficult to be applied to the existing compound SA-based sparse Transformers.

Besides Sputnik, NVIDIA cuSPARSE, and cuSPARSELt, the VectorSparse library [9] processes the sparse operations of the special fine-grained patterns.

It introduced a column−vector−sparse−encoding method and a tensor−core−based one−dimensional octet tiling scheme, which led to efficient memory access and computation under a small grain size. The library is efficient for sparse operations of the column−vector sparse pattern. However, it underperforms when processing other fine−grained sparse patterns, such as global and random patterns.

Chapter 4

Tiled Singular Value Decomposition (TSVD)

4.1 Contribution

We propose a TSVD-based low-rank approximation approach for accelerating Transformer-based models. TSVD divides an weight matrix into smaller tiles, factorizes each tile using SVD, and then performs TSVD-matmul during inference. It enables model compression and computation reduction. Compared to the conventional SVD approach, TSVD-matmul is efficient on GPUs, particularly in cases where matrices are small or tall-and-skinny shapes. At equivalent compression ratio as SVD, TSVD achieves a speedup ranging from 1.02 to 2.26$\times$ on GPUs, effectively utilizing on-chip memory bandwidth and GPU parallelism. When deployed to GPT-2 for the E2E NLG task, TSVD achieves a speedup up to 1.11$\times$ and increases accuracy by a 1.5-point BLEU score without a hyperparameter search, compared to full fine-tuning. We further propose TSVD-common, a parameter-efficient fine-tuning method, which improves the accuracy decreased by model compression. It leads to enhance the model accuracy to the original level or beyond.

Consequently, the TSVD approach not only facilitates model compression almost without accuracy degradation in small models such as GPT-2 but also reduces the execution time. Moreover, for large models such as GPT-3, the TSVD approach presents the additional advantage of minimizing the number of

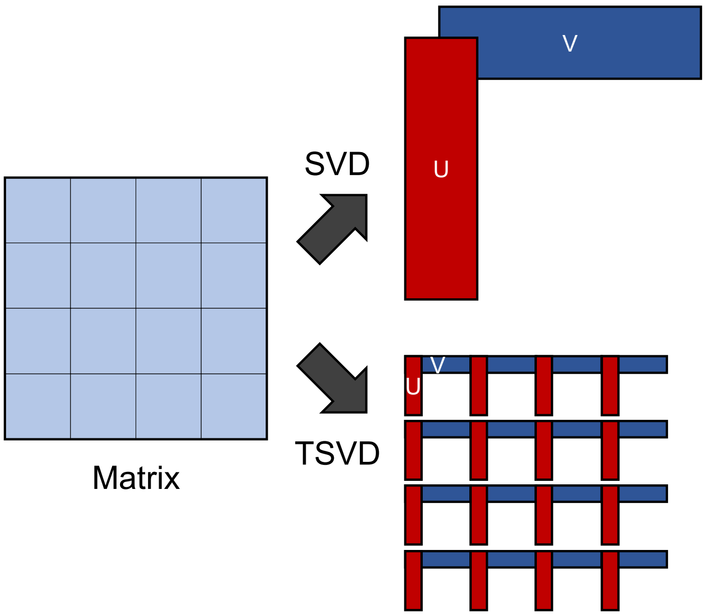


Figure 4.1: Concept of the TSVD, which divides a matrix into smaller tiles and factorizes each tile using SVD.

GPU requirements due to model compression.

4.1.1 Tiled Singular Value Decomposition (TSVD)

TSVD is a low-rank approximation technique that breaks down a matrix into smaller tiles and factorizes each tile using SVD (see Fig. 4.1). Given a matrix W with dimensions (K, N), we divide it into smaller tiles, denoted as W_{tile}, each with dimensions (tK, tN). Each W_{tile} is then independently factorized by SVD to obtain three smaller tiles: a (tK, tR) orthogonal matrix U_{tile}, a (tR, tR) diagonal matrix S_{tile}, and a (tN, tR) orthogonal matrix V_{tile}. Thus, W_{tile} can be expressed as $W_{tile} = U_{tile} * S_{tile} * V^T_{tile}$. We truncate the U_{tile}, S_{tile}, and V^T_{tile} to a predetermined tiled rank size of tR', where $tR' \ll min(tK, tN)$. As a result, $W_{tile} \approx U_{tile}r * S_{tile}r * V^T_{tile}r$. Here, $U_{tile}r$, $S_{tile}r$, and $V^T_{tile}r$ represent the truncated matrices with shape of (tK, tR'), (tR', tR') and (tN, tR'), respectively.

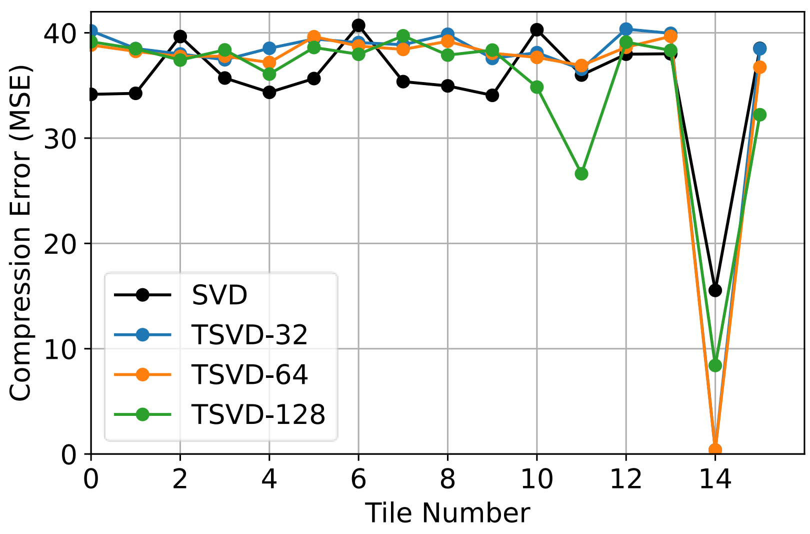


Figure 4.2: Compression errors on the various SVD methods at a 2× compression ratio. The compression error is calculated by mean square error (MSE) of each tile between recovered matrix after compression and original matrix with dimension of (256, 256). We randomly set an important tile (e.g., the 14th tile values are much larger than other tiles in the matrix), and the size of this important tile is (64, 64). As the tile size decreases, TSVD preserves more important information.

TSVD demonstrates greater robustness than the SVD, as it is more effective at preserving essential information. To compare the robustness between SVD and TSVD, we measure the mean square error of each tile between recovered matrix after compression (i.e., SVD and TSVD) and original matrix (see Fig. 4.2). TSVD retains important information (i.e., 14th tile) more effectively than the SVD after compression. By factorizing into smaller regions, TSVD can help mitigate important data loss during the overall factorization process. The data is confined within its respective tiles and does not propagate to other tiles. As a result, TSVD can preserve important information in each tile.

4.1.2 TSVD-based matrix multiplication in Transformer models

We can accelerate Transformer models during inference using TSVD-based low-rank approximation. Similarly to SVD, applying TSVD to the models can also compress the model and reduce computation. In Fig. 4.3, the total computation and weight parameters in GPT-2 medium with TSVD decrease as the compression ratio increases.

Deploying TSVD to Transformer models involves two key steps. Firstly, we compress the weight matrices involved in matrix multiplications through TSVD-based low-rank approximation. This results in smaller matrix sets, namely $\mathrm{SU}_{tile}r$, $\mathrm{SS}_{tile}r$, and $\mathrm{SV}^T_{tile}r$. $\mathrm{SS}_{tile}r$ is multiplied in advance by either the $\mathrm{SU}_{tile}r$ or $\mathrm{SV}^T_{tile}r$. Thus, our weight parameters consist only of the $\mathrm{SU}_{tile}r$ and $\mathrm{SV}_{tile}r$. By using this approach, we can achieve the same reduction in the size of weight parameters as SVD at equivalent compression ratios. We empirically set the tile size according to the input matrix size and hardware architecture. Secondly, We perform TSVD-based matrix multiplication (TSVD-matmul, i.e., $\mathrm{Y} = \mathrm{X} * \mathrm{SU}_{tile}r * \mathrm{SV}^T_{tile}r$) during inference, instead of the original matrix multiplication ($\mathrm{Y} = \mathrm{X} * \mathrm{W}$). When $tK = tN$ for simplicity, we can achieve the same computation reduction as the SVD method at equivalent compression ratios.

TSVD-matmul is more efficient on GPUs than SVD approach. Through the roofline analysis (see Fig. 4.8), TSVD-matmul is leveraging GPU resources more efficiently than other approaches in case 3 and 4 because it employs a small partial matrix P_{tile} (i.e., the results of the $\mathrm{X}_{tile} * \mathrm{U}_{tile}$) which can be stored in faster memory. This reduces unnecessary data load and multiplication. Furthermore, TSVD-matmul can mitigate inefficiency by transitioning the memory-bound characteristics from the whole SVD-matmul to the tile-level, as tile-level op-

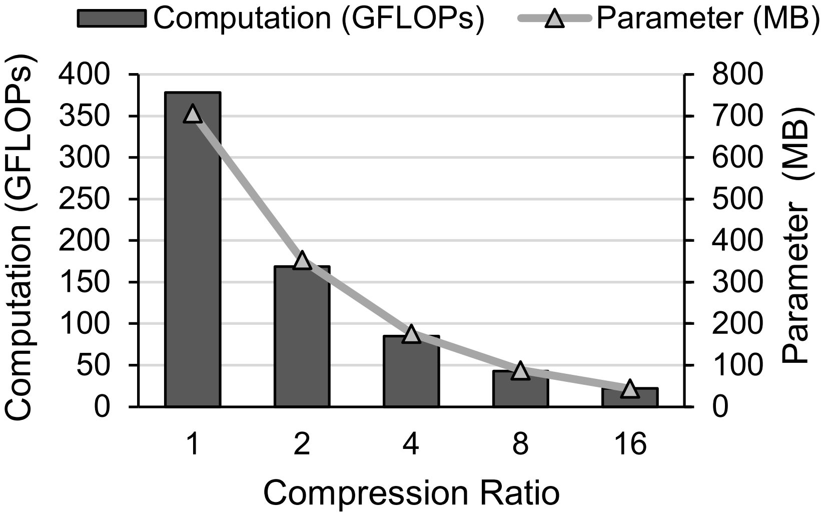


Figure 4.3: Total computation and weight parameters in GPT-2 medium with TSVD as the compression ratio increases. When the compression ratio is one, the computation and parameters are identical to the original GPT-2 medium with 512 input sequences and 64 output tokens. These results are calculated by applying TSVD to all trainable parameters, such as the embedding table and weights in the attention and feedforward.

erations can fully utilize the on-chip memory bandwidth.

4.1.3 Kernel Design

We designed a customized TSVD-matmul GPU kernel that adheres to the general matrix multiplication mechanism on a GPU, taking reference from NVIDIA CUTLASS [1] (v2.11.0). Within the thread block that processes the tile-level operations, it fuses and executes the operations $\mathrm{P}_{tile} = \mathrm{X}_{tile} * \mathrm{U}_{tile}r$ and $\mathrm{Y}_{tile} = \mathrm{P}_{tile} * \mathrm{V}_{tile}r$ with tensor cores. For warp-level processes and matrix multiply-and-accumulate (MMA) operations, we adhere to the CUTLASS method. Alg. 1 presents the pseudo-codes for our TSVD-matmul.

We optimize our TSVD-matmul kernel using the stream-K [46] method

Algorithm 1: TSVD-matmul

```
Input: $X \in \mathcal{R}^{M \times K}$, $SU_{tile} \in \mathcal{R}^{\#tile \times tK \times tR}$, and $SV_{tile} \in \mathcal{R}^{\#tile \times tR \times tN}$ in
    $gmem$
Output: $Y \in \mathcal{R}^{M \times N}$ in the $gmem$
foreach thread block $TB_n, n \in (0, \frac{MN}{tMtN})$ do
    $\_\_shared\_\_$ $smem[2\times$ size$(X_{tile} + U_{tile} + V_{tile})]$
    Set registers $reg_x$, $reg_u$, $reg_v$, $reg_p$, $reg_y$
    Load $X_{tile0}$, $U_{tile0}$, and $V_{tile0}$ from $gmem$ to $smem$
    for $ki = 0$ to $\frac{K}{tK} - 1$ do
        if $ki \neq \frac{K}{tK} - 1$ then
            Load $X_{tile(ki+1)}$, $U_{tile(ki+1)}$, and $V_{tile(ki+1)}$ from $gmem$ to
             $smem$
        end
        Load $X_{tile(ki)}$, $U_{tile(ki)}$, and $V_{tile(ki)}$ from $smem$ to $reg_x$, $reg_u$,
         $reg_v$
        Compute $reg_p = reg_x * reg_u$
        Compute $reg_y = reg_p * reg_v + reg_y$
    end
    Store $Y_{tile_n}$ from $reg_y$ to $smem$
    Store $Y_{tile_n}$ from $smem$ to $reg_y$
    Store $Y_{tile_n}$ from $reg_y$ to $gmem$
end
```

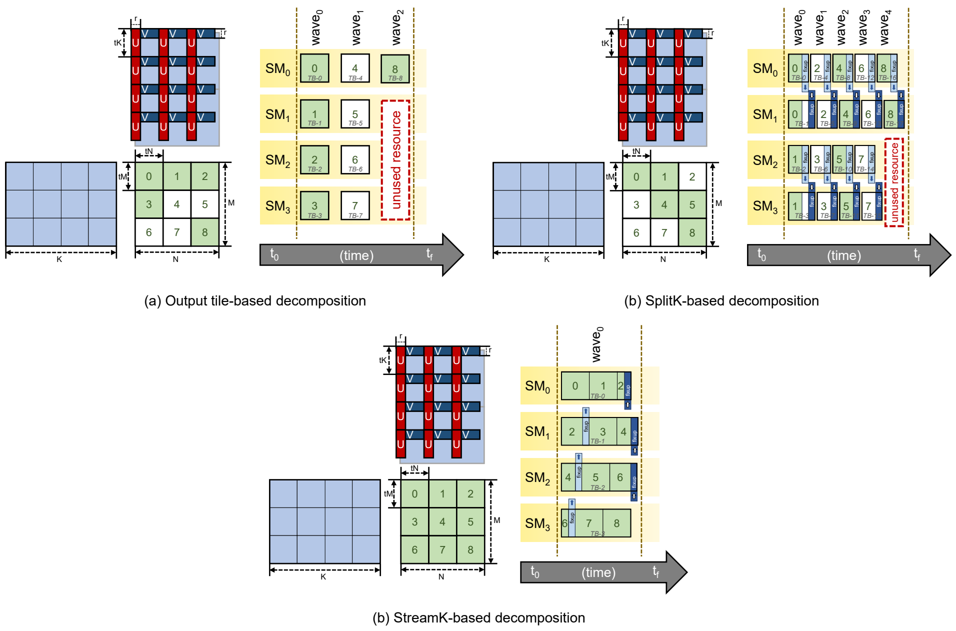

(a) Output tile-based decomposition
(b) SplitK-based decomposition
(b) StreamK-based decomposition
wave0
wave1
wave2
wave3
wave4
SM0
SM1
SM2
SM3
unused resource
fixup
t0
(time)
tf
tK
tM
tN
K
N
M
U
V

Figure 4.4: A various parallel workload decomposition techniques [46] at the TSVD-matmul. The green and white blocks in the SMs represent output tiles. In the figure (a), the entire output tile 0, allocated by thread block 0 (TB_0), is processed on SM_0. The two types of blue blocks represent reduction operations for load and store from the global memory.

(TSVD-K), a work-centric parallelization technique, to address load imbalance issues. Instead of allocating an output tile to a thread block as in typical output tile-based decomposition (see Fig. 4.4(a)), TSVD-K equally distributes inner loop iterations among processing output tiles, ensuring optimal utilization of computing resources (see Fig. 4.4(c)). The SplitK-decomposition, which equally distributes k-dimension inner loop iterations to thread blocks (see Fig. 4.4(b)), also fully utilizes computing resources. However, it incurs significant accumulation overhead to achieve complete output tiles (e.g., fixup block, as indicated by light or deep blue blocks in the output matrix). TSVD-K mitigates this overhead by reducing the number of thread blocks, while still maintaining a count nearly equal to the number of streaming multiprocessors (SMs). Additionally,

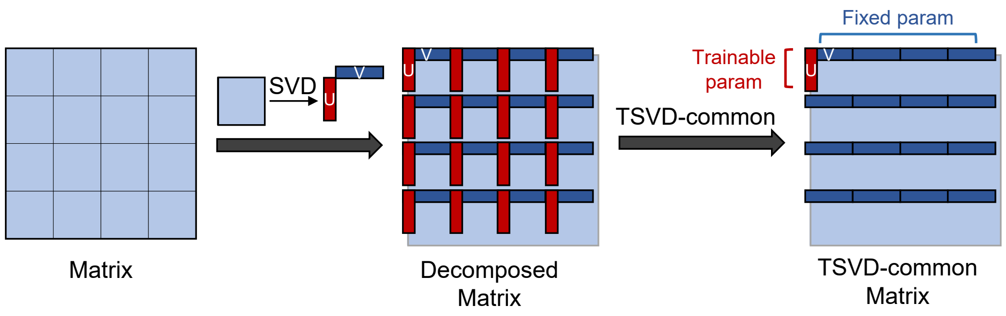


Figure 4.5: The concept of the TSVD-common method. It shares one of the submatrices decomposed by SVD in each tile across all tiles and fine-tunes only the common submatrix (red U) during training.

TSVD-K can reuse left-hand-side matrix blocks in shared memory. Therefore, this strategy not only enables efficient use of GPU computing resources, but also reduces accumulation overheads.

4.1.4 TSVD-Common: a PEFT of low-rank approximation

We propose TSVD-common, a parameter efficient fine-tuning method, to improve the decreased accuracy by the compression. TSVD-common (see Fig. 4.5 shares one of the submatrices decomposed by SVD in each tile across all tiles and fine-tunes only the common submatrix during training. It can reuse common singular vectors such as U_{tile} or V_{tile} in each thread block when performing tile-level operations, so we could optimize our TSVD-matmul kernel with common singular vectors additionally. To reduce the compression error after TSVD-common, We decompose the weight matrices as shown in Fig. 4.6. First, we split the weight matrix by the height of the tile and concatenate it as in Fig. 4.6 (a). Next, we obtain U and multiple V after performing SVD as in Fig. 4.6 (b). If we then reshape V as depicted by the dashed box, all V will share U. In the end, we can obtain submatrices factorized by the TSVD-common method.

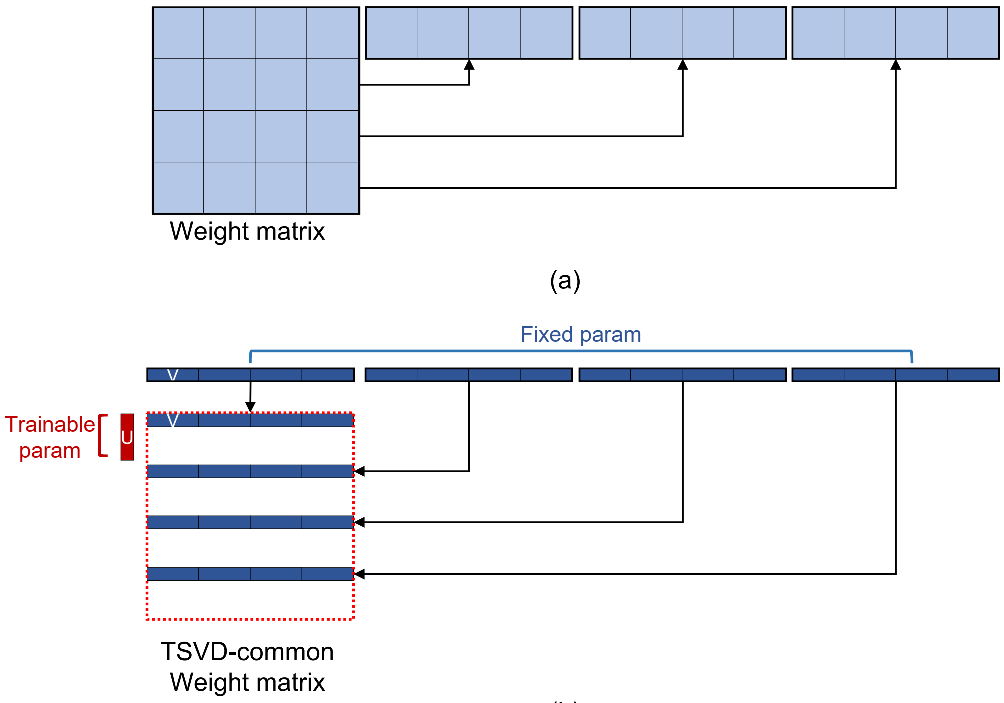


Figure 4.6: TSVD-common decomposition.

TSVD-common presents two distinct advantages over LoRA [26], a representative method for parameter-efficient fine-tuning. The first advantage is a reduction in the number of parameters due to model compression, and the second is a decrease in execution time also resulting from model compression. As LoRA is not a model compression technique, it maintains the same number of parameters as the original model, and its execution time is similar to, if not slightly slower than, traditional fine-tuning. In contrast, TSVD is superior to LoRA in terms of both parameter size and computational time.

4.2 Evaluation

4.2.1 Experiment Setup

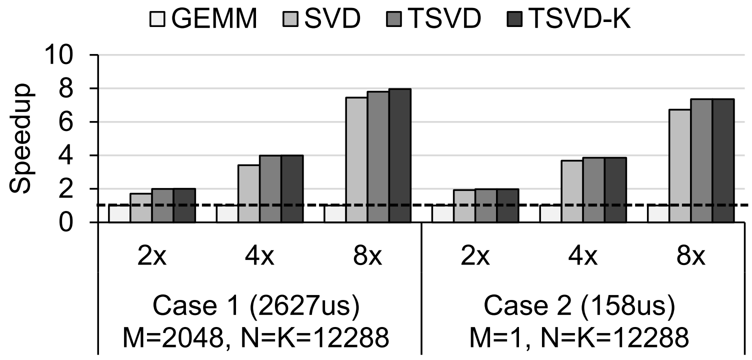
We evaluated TSVD-matmul on the NVIDIA A100 using FP16. We executed the general matrix multiplication (GEMM) using the NVIDIA cuBLAS library, and performed SVD-matmul with two GEMMs, as no existing library currently supports SVD-matmul fusion. We designed the customized TSVD-matmul GPU kernel based on the NVIDIA CUTLASS library. We also evaluated the speedup in GPT-2 inference by applying TSVD to the matrix multiplications in the attention layers and feedforward layers. To maintain the model's accuracy after compression, we fine-tuned GPT-2 medium on the E2E NLG task [18] using TSVD with various compression ratios. We evaluated TSVD-common, an efficient parameter fine-tuning method, on the WebNLG [22] task, which is more challenging than the E2E NLG [18] task. Our evaluation considered factors such as the number of shared submatrices, tile size, and the adaptation location (e.g., attention or feedforward).

4.2.2 End-to-end latency on various input and output tokens for GPT models

Fig. 4.7 illustrates the speedup for realistic problem cases at different compression ratios. Cases 1 and 2 involve specific matrix multiplication in the 175B GPT-3 model's inference process, with Case 1 handling an input sequence of 2048 during the summarization stage and Case 2 producing a single output token during the generation stage. Conversely, Cases 3 and 4 pertain to a similar operation in the GPT-2 medium model's inference, with Case 3 dealing with an input sequence of 512 in the summarization stage and Case 4 generating a single output token during the generation stage. The comparison is made with respect to GEMM without compression (used as the baseline), SVD-matmul, TSVD-matmul, and TSVD-matmul optimized with the Stream-K method (TSVD-K).

As shown in the figure, TSVD performance improvement is more significant as the compression ratio increases. In cases where large matrices are decomposed by SVD or TSVD (refer to cases 1 and 2 in Fig. 4.7), similar trends can be observed with SVD. However, TSVD achieves greater speedup than SVD at higher compression ratios, as the large matrices become tall-and-skinny after compression. Conventional GPU libraries such as cuBLAS tend to process skewed matrix multiplications, including tall-and-skinny matrices, inefficiently due to memory-bound operations. In addition, the operations result in issues such as load imbalance or limited parallelism. In contrast, TSVD can mitigate these problems by transitioning the tall-and-skinny feature from full matrices to tile-level. This strategy fully utilizes the fast memory bandwidth in on-chip memory, enabling to alleviate the memory-bound bottleneck. Furthermore, the stream-K method helps resolve load imbalance issues. This method fully utilizes the computing resources because TSVD-K distributes inner loop iterations processed by the SMs equally.

In cases where smaller matrices are decomposed by SVD or TSVD (refer to cases 3 and 4 in Fig. 4.7), TSVD achieves a considerable speedup compared to SVD, attributable to two main factors. First, both TSVD and TSVD-K reduce kernel launch time. While SVD requires two kernel launches for operations, both TSVD and TSVD-K need a single kernel launch. In the case 3 and 4, due to a small amount of computation and memory traffic, the portion of time spent



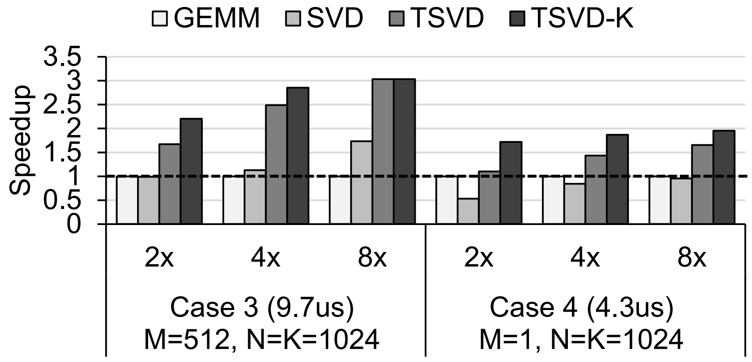


Figure 4.7: Speedup on matrices of various sizes using different compression ratios ($N\times$) on the A100 GPU. In Cases 1 and 3, the GEMM is compute-bound, and the TSVD tile size is 256. In Cases 2 and 4, the GEMM is memory-bound, and the TSVD tile size is 64. The dotted line represents the baseline GEMM without any low-rank approximation in each case. Additionally, the execution time of each baseline is displayed below the legend for the respective cases.(TSVD-K: TSVD with stream-K, Ratio: compression ratio)

on kernel launch increases compared to previous cases. Second, SVD is difficult to fully utilize computing resources due to limited grid sizes (i.e., the number of thread blocks). Even though adapting to the splitK-decomposition method to increase grid size, it lags behind TSVD and TSVD-K due to accumulation overhead. In the case 4, SVD performs even slower than our baseline, which is executed without compression, mainly due to the time consumed by kernel launch. Therefore, both TSVD and TSVD-K demonstrates greater efficiency

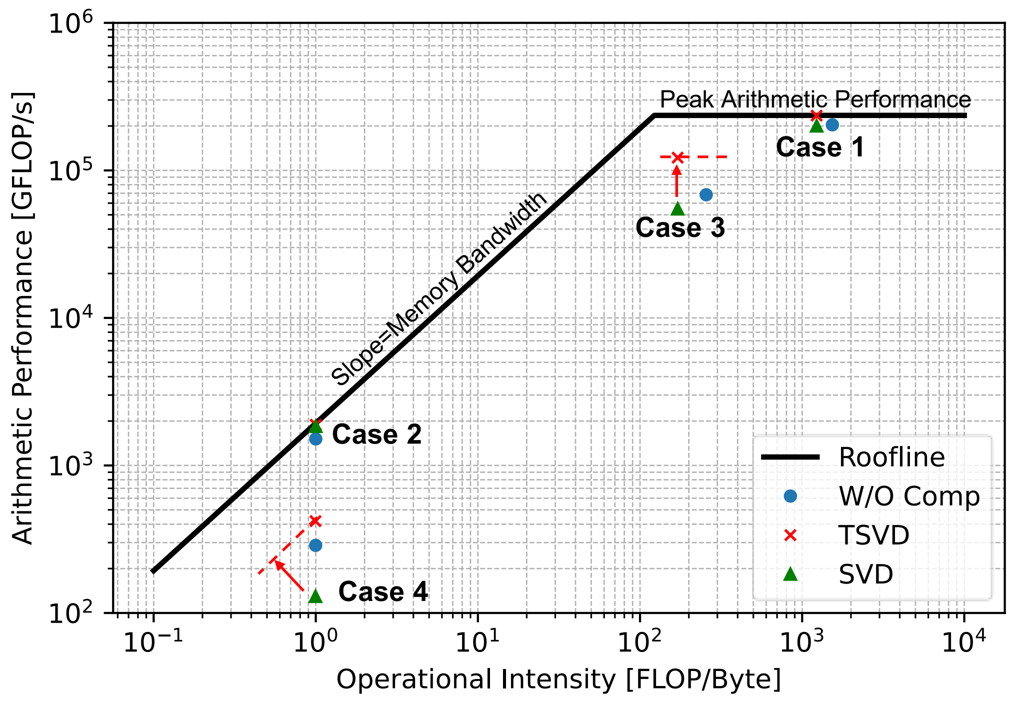
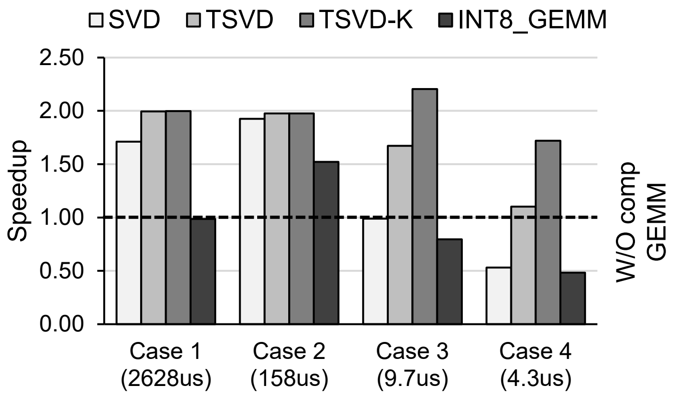


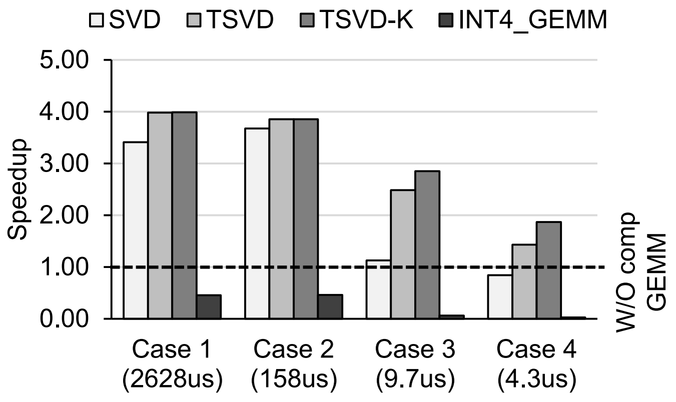
Figure 4.8: Roofline model for the matrix multiplication in four cases. The performance ceiling is based on the NVIDIA GPU A100 with 80GB HBM achieving up to 1,935GB/s with the base frequency of SM. The blue circles in the figure are matrix multiplications without compression as a baseline in Fig. 4.7 In case 3 and 4, where smaller matrices are involved in the matrix multiplication, TSVD-based matrix multiplications are faster than the other approaches by utilizing more GPU resources.

than SVD.

The roofline performance model, as depicted in Fig. 4.8, serves as a performance evaluation tool that shows the computational characteristics of the operation within computer systems. The x-axis denotes the arithmetic intensity, which is quantified as the ratio of arithmetic operations to singular memory access. Correspondingly, the y-axis stands for the computational throughput in terms of arithmetic operations executed per second. The performance of the operation on the side with a diagonal line is bounded by memory bandwidth, where



(a) At the compression ratio 2



(b) At the compression ratio 4

Figure 4.9: Comparison of speedup among SVD, TSVD, and low-precision quantization-based GEMM. Here, INT8-GEMM denotes the 8-bit quantization GEMM from Faster Transformer [42], while INT4-GEMM refers to the 4-bit quantization GEMM from GPTQ [19] Triton-based GPU kernel.

the slope means the memory bandwidth. Conversely, in the region bounded by the horizontal line, performance is limited by the peak arithmetic performance of the processor. Through the roofline analysis, TSVD-based matrix multiplication is leveraging GPU resources more efficiently than other approaches in case 3 and 4. As a results, TSVD approach achieves greater speedup in these cases (see Fig. 4.7)

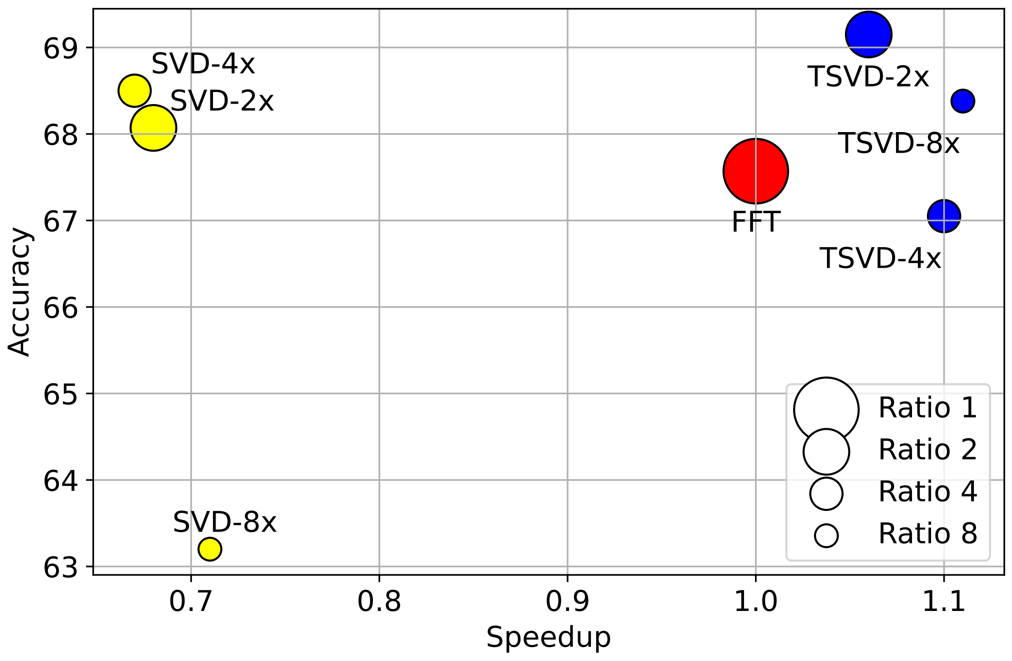
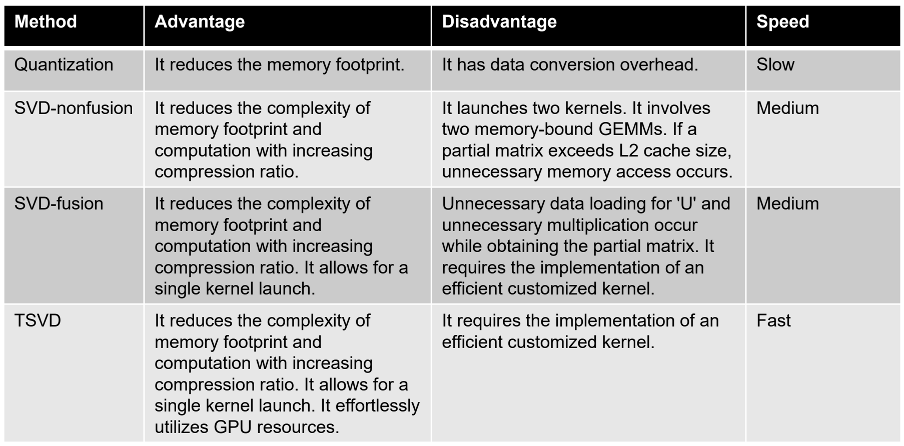


Figure 4.10: Tradeoff between accuracy and speedup on A100 GPU. Ratio is the compression ratio for the weights in GPT2-M except for the embedding layer. (FFT: full fine-tuning, SVD: singular value decomposition, TSVD: tiled singular value decomposition)

We compared our TSVD to the quantization-based matrix multiplication used for accelerating Transformer-based models on GPUs. Fig. 4.9 represents the evaluation of INT8-GEMM and INT4-GEMM as quantization approaches [1]. Faster Transformer [42] supports INT8-GEMM, while INT4-GEMM uses the GPTQ [19]'s Triton-implemented GPU kernel. As a result, SVD-based methods outperform the quantization-based approaches, given that existing GPUs are not particularly optimized for low-precision quantization operations. Fig. 4.11 represents differences among quantization, SVD, and TSVD methods during matrix multiplication.

Fig. 4.10 demonstrates the trade-off between accuracy and speedup when

[1]Our applied quantization-based matrix multiplication has a data conversion overhead.



Method	Advantage	Disadvantage	Speed
Quantization	It reduces the memory footprint.	It has data conversion overhead.	Slow
SVD-nonfusion	It reduces the complexity of memory footprint and computation with increasing compression ratio.	It launches two kernels. It involves two memory-bound GEMMs. If a partial matrix exceeds L2 cache size, unnecessary memory access occurs.	Medium
SVD-fusion	It reduces the complexity of memory footprint and computation with increasing compression ratio. It allows for a single kernel launch.	Unnecessary data loading for 'U' and unnecessary multiplication occur while obtaining the partial matrix. It requires the implementation of an efficient customized kernel.	Medium
TSVD	It reduces the complexity of memory footprint and computation with increasing compression ratio. It allows for a single kernel launch. It effortlessly utilizes GPU resources.	It requires the implementation of an efficient customized kernel.	Fast

Figure 4.11: Differences among Quantization, SVD, and TSVD Methods during Matrix Multiplication.

incorporating TSVD into GPT-2 for the E2E NLG [18] task. TSVD not only performed competitively with full fine-tuning without compression, but also it achieved a speedup of 1.06 to 1.11× at compression ratios ranging from 2 to 8, while increasing the accuracy by a 1.5-point BLEU score. By preserving important information after compression compared to SVD, TSVD mitigates the degradation of the model quality. It is beneficial to recover the accuracy of the model during additional fine-tuning.

Fig. 4.12 illustrates the speedup of the end-to-end execution time in the GPT-2 medium model as the length of the output sequence increases. The input sequence length is fixed at 512, and a beam search width of 10 is used with a single batch size. The red dotted line is our baseline without any compression methods. As shown in the figure, TSVD is more efficient at shorter output sequence lengths due to the predominance of the summarization portion.

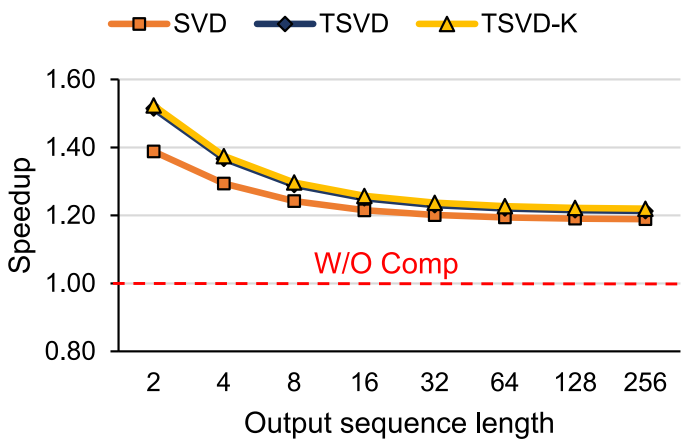


Figure 4.12: speedup of the end-to-end execution time in the GPT-2 medium model as the length of the output sequence increases. The input sequence length is fixed at 512, and a beam search width of 10 is used with a single batch size. The red dotted line is our baseline without any compression methods. TSVD is more efficient at shorter output sequence lengths due to the predominance of the summarization portion.

Table 4.1: Impact of increasing the number of trainable parameters while keeping the compression ratio fixed, by changing the number of shared matrices or tile sizes when applying TSVD-common to GPT-2 medium on WebNLG [22] (In our method name, 'T' denotes tile size, and 'U' denotes the number of shared matrices.)

Method Name	#Trainable Param	Model Size	Unseen Score	Seen Score	All
Full fine-tuning [51]	354.9M	1420MB	32.7	62.0	48.4
LoRA [26]	0.35M	1420MB	45.5	64.3	55.8
TSVD-U1T64	0.59M	710MB	12.4	60.2	38.9
TSVD-U1T128	2.36M	710MB	17.2	62.5	40.8
TSVD-U1T256	9.44M	710MB	25.3	62.1	45.2
TSVD-U1T512	37.75M	710MB	32.4	62.6	49.0
TSVD-T128U1	2.36M	710MB	15.7	61.7	40.1
TSVD-T128U4	9.44M	710MB	20.9	62.3	42.7
TSVD-T128U16	37.75M	710MB	30.7	62.6	48.1

4.2.3 TSVD-common accuracy on various cases

To recover the model quality compromised by compression, TSVD-common only trains shared submatrices (i.e., submatrices U in our evaluation), while keeping the other submatrices fixed. We employ TSVD-common to improve accuracy, taking into account various factors such as the number of shared matrices, tile size, shared shape (e.g., row-wise, column-wise, and square), and changes in adaptation location (e.g., attention or feedforward).

Table. 4.1 illustrates the BLEU score associated with increasing the number of trainable parameters, accomplished by altering the number of shared matrices or tile sizes. As the number of trainable parameters increases, accuracy improves. Particularly, when the tile size is set to 512, the accuracy exceeds that achieved by the full fine-tuning method. However, despite these improvements, the model's quality does not fully recover, as indicated by the low unseen score, signifying that the model's generality remains compromised.

We applied TSVD-common separately to both the attention and feedforward components to investigate whether model compression-related information loss occurs in either of these areas. As illustrated in Table 4.2, both methods demonstrate an improvement in BLEU scores as the tile size increases. This improvement even surpasses the accuracy of the full fine-tuning method when the tile size is set to 256. Furthermore, applying TSVD-common to the attention component results in a better unseen score. Therefore, we can infer that the feedforward component more significantly impairs the model's generality.

We compressed the model at different ratios (see Table 4.3). The table shows that when using TSVD-common without compression, its accuracy closely matches that of LoRA. However, as the compression ratio increases, the model's generality is dramatically compromised.

We evaluated TSVD-common on the different tasks such as DART [39] and E2E NLG [18] task. Table. 4.4 shows in the DART a similar trend to

Table 4.2: Impact of adaptation location when applying TSVD-common to GPT-2 medium on WebNLG [22] We set the compression ration as 2. ('attn' denotes attention part, and 'ff' denotes the feedforward part.)

Method Name	Adaptation Location	#Trainable Param	Model Size	Unseen Score	Seen Score	All
Full fine-tuning [51]		354.9M	1420MB	32.7	62.0	48.4
LoRA [26]		0.35M	1420MB	45.5	64.3	55.8
TSVD-U1T64	attn	0.20M	1110MB	28.3	56.4	43.8
TSVD-U1T128	attn	0.79M	1112MB	32.5	62.6	49.2
TSVD-U1T256	attn	3.15M	1122MB	35.3	63.5	50.9
TSVD-U1T64	ff	0.39M	909MB	26.5	59.9	45.1
TSVD-U1T128	ff	1.57M	914MB	31.1	62.9	48.9
TSVD-U1T256	ff	6.29M	932MB	34.9	63.1	50.8

Table 4.3: Impact of the different compression ratio when applying TSVD-common to GPT-2 medium on WebNLG [22] We set the tile size as 256. ('all' denotes both attention and feedforward part. 'R' represents the compression ratio of the fixed parameter size.)

Method Name	Adaptation Location	#Trainable Param	Model Size	Unseen Score	Seen Score	All
Full fine-tuning [51]		354.9M	1420MB	32.7	62.0	48.4
LoRA [26]		0.35M	1420MB	45.5	64.3	55.8
TSVD-U1T256R1	all	18.87M	1420MB	44.3	63.8	55.2
TSVD-U1T256R2	all	9.44M	710MB	25.3	62.1	45.2
TSVD-U1T256R4	all	3.15M	355MB	8.8	59.7	36.4

WebNLG [22]. When using TSVD-common without compression, its accuracy also competitively matches that of LoRA. However, in the E2E NLG task, ㅑ t exhibits performance almost close to LoRA even at high compression rates. Therefore, the effectiveness of our TSVD method varies with the difficulty of the downstream task.

Table 4.4: Impact of the different tile size and compression ratios when applying TSVD-common to GPT-2 medium on DART [39] and E2E [18] We set the adaptation location as 'all'.

Method Name	#Trainable Param	Model Size	BLEU score
	DART [39] task		
Full fine-tuning [51]	354.9M	1420MB	46.0
LoRA [26]	0.35M	1420MB	47.5
TSVD-U1T64	1.18M	1420MB	47.6
TSVD-U1T128	4.72M	1420MB	47.3
TSVD-U1T256	18.87M	1420MB	45.6
TSVD-U1T64R2	0.59M	710MB	41.7
TSVD-U1T128R2	2.36M	710MB	44.1
TSVD-U1T256R2	9.44M	710MB	45.0
	E2E NLG [18] task		
Full fine-tuning [51]	354.9M	1420MB	67.5
LoRA [26]	0.35M	1420MB	70.2
TSVD-U1T256R1	0.59M	710MB	69.7
TSVD-U1T256R2	2.36M	710MB	69.6
TSVD-U1T256R4	9.44M	710MB	68.2

4.3 Related Work

4.3.1 Model compression

Model compression is a critical area of research in machine learning, particularly for deploying large models on resource-constrained devices. There are popular methods for model compression such as singular value decomposition (SVD) and quantization.

Low-rank approximation

SVD is a linear algebra technique that decomposes a matrix into three other matrices. In the context of model compression, SVD is often used to approximate the weight matrices of neural networks. The idea is to decompose the weight matrix into two lower-rank matrices, which can significantly reduce the number of parameters without a substantial loss in performance. One of the earliest works to apply SVD for model compression is by [23], who used SVD to compress fully connected layers in neural networks. They demonstrated that SVD could reduce the model size significantly without a substantial drop in accuracy. In the work by [25], the authors introduced Fisher information to measure the importance of parameters in SVD, but it required a large amount of labeled data to fine-tune the compressed model. In conclusion, while SVD is a powerful tool for model compression, it has limitations, particularly when applied to Transformer models. Especially, it minimizes the squared error towards reconstructing the original matrix without considering the importance of different parameters. This can lead to larger reconstruction errors for parameters that significantly impact task accuracy. In addition, SVD operation is inefficient on a GPU at high compression ratio. Recent research has focused on developing new methods that can overcome these limitations and provide more effective model compression.

Quantization

Quantization is a widely used technique for model compression and acceleration. It involves reducing the precision of the weights in a model, which can significantly decrease the model size and computational requirements. [48] propose a system that uses quantization to compress the model size of large-scale generative language models. Their system, LUT-GEMM, reduces the latency of individual GPUs and the overall inference process, providing significant performance improvements. It supports both non-uniform and uniform quantization formats and can greatly reduce energy consumption. Another significant work is [14], which develops a procedure for Int8 matrix multiplication for feed-forward and attention projection layers in Transformers. This method cuts the memory needed for inference by half while retaining full precision performance. The authors developed a two-part quantization procedure, LLM.int8(), which uses vector-wise quantization with separate normalization constants for each inner product in the matrix multiplication to quantize most of the features. [19] quantized GPT models with 175 billion parameters in approximately four GPU hours, reducing the bitwidth down to 3 or 4 bits per weight, with negligible accuracy degradation relative to the uncompressed baseline.

4.3.2 PEFT

Parameter-efficient fine-tuning is a crucial area in machine learning, aiming to adapt large pre-trained models to specific tasks by updating only a small fraction of the model's parameters. Early work includes adapter tuning [24], which inserts small adapter modules into the pre-existing layers of a pre-trained model. Low Rank Adaptation (LoRA) [26] is a famous method. LoRA decomposes the attention weight update into low-rank matrices, thereby reducing the number of trainable parameters. This method has been shown to be effective for fine-tuning large language models like GPT-3. More recent work includes LLaMA-Adapter [21] and LLaMA-Adapter V2 [68] by researchers from Stanford University and Google Research. These methods introduce learnable adaption prompts and a zero-init attention mechanism to adaptively inject new instructional cues into the model, with the V2 version incorporating more learnable parameters and an early fusion strategy. These methods have shown promise in adapting large pre-trained models to specific tasks, reducing computational and memory requirements, and representing an important direction for future research in machine learning.

Chapter 5

Conclusion

In conclusion, this thesis has developed and proposed novel optimization methods for Transformer-based models on Graphics Processing Units (GPUs) to mitigate the increasing inference costs arising from the expanding size of these models and the lengthening of their input sequences. The proposed strategies, Multigrain and TSVD, have proven to be highly effective in specific scenarios, reducing inference costs significantly.

Multigrain is an Transformer-specific optimization method particularly effective when the input length (L_{in}) is significantly larger than the hidden dimension (D_h). By efficiently processing the coarse-grained and fine-grained parts of the attention mechanism with high-performance tensor cores and CUDA cores respectively, Multigrain achieves remarkable speedups.

The TSVD method, on the other hand, proves to be invaluable in scenarios where L_{in} is similar to or smaller than D_h. By utilizing tiled singular value decomposition (TSVD) to compress matrices, memory footprint and computation are substantially reduced, leading to lowered inference costs.

However, we acknowledged the inherent trade-off between reduced inference costs and decreased accuracy when applying TSVD to models. To mitigate this, we introduced the TSVD-common fine-tuning method, which maintains a shared submatrix across all tiles, fine-tuning only the common submatrix during training. This method has successfully improved accuracy even when the model is compressed.

These approaches present a promising avenue for further enhancing the performance of Transformer-based models, making them more accessible and cost-effective for various complex, real-world applications. Future work could focus on refining these methods or exploring additional ways of reducing inference costs without sacrificing model performance.

REFERENCES

[1] J. D. Andrew Kerr, Duane Merrill and J. Tran, "CUTLASS: Fast Linear Algebra in CUDA C++," https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/, 2017.

[2] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen, E. Chu, J. H. Clark, L. E. Shafey, Y. Huang, K. Meier-Hellstern, G. Mishra, E. Moreira, M. Omernick, K. Robinson, S. Ruder, Y. Tay, K. Xiao, Y. Xu, Y. Zhang, G. H. Abrego, J. Ahn, J. Austin, P. Barham, J. Botha, J. Bradbury, S. Brahma, K. Brooks, M. Catasta, Y. Cheng, C. Cherry, C. A. Choquette-Choo, A. Chowdhery, C. Crepy, S. Dave, M. Dehghani, S. Dev, J. Devlin, M. D□az, N. Du, E. Dyer, V. Feinberg, F. Feng, V. Fienber, M. Freitag, X. Garcia, S. Gehrmann, L. Gonzalez, G. Gur-Ari, S. Hand, H. Hashemi, L. Hou, J. Howland, A. Hu, J. Hui, J. Hurwitz, M. Isard, A. Ittycheriah, M. Jagielski, W. Jia, K. Kenealy, M. Krikun, S. Kudugunta, C. Lan, K. Lee, B. Lee, E. Li, M. Li, W. Li, Y. Li, J. Li, H. Lim, H. Lin, Z. Liu, F. Liu, M. Maggioni, A. Mahendru, J. Maynez, V. Misra, M. Moussalem, Z. Nado, J. Nham, E. Ni, A. Nystrom, A. Parrish, M. Pellat, M. Polacek, A. Polo-

zov, R. Pope, S. Qiao, E. Reif, B. Richter, P. Riley, A. C. Ros, A. Roy, B. Saeta, R. Samuel, R. Shelby, A. Slone, D. Smilkov, D. R. So, D. Sohn, S. Tokumine, D. Valter, V. Vasudevan, K. Vodrahalli, X. Wang, P. Wang, Z. Wang, T. Wang, J. Wieting, Y. Wu, K. Xu, Y. Xu, L. Xue, P. Yin, J. Yu, Q. Zhang, S. Zheng, C. Zheng, W. Zhou, D. Zhou, S. Petrov, and Y. Wu, "Palm 2 technical report," arXiv:2305.10403, 2023.

[3] I. Beltagy, A. Cohan, H. Hajishirzi, S. Min, and M. E. Peters, "Beyond Paragraphs: NLP for Long Sequences," in Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL), 2021.

[4] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The Long-Document Transformer," arXiv:2004.05150, 2020.

[5] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," in Proceedings of Neural Information Processing Systems (NeurIPS), 2020.

[6] J. Burgess, "RTX on—The NVIDIA Turing GPU," Micro, IEEE, vol. 40, no. 2, 2020.

[7] J. Chen, A. Zhang, X. Shi, M. Li, A. Smola, and D. Yang, "Parameter-efficient fine-tuning design spaces," arXiv:2301.01821, 2023.

[8] J. Chen, N. Xiong, X. Liang, D. Tao, S. Li, K. Ouyang, K. Zhao, N. DeBardeleben, Q. Guan, and Z. Chen, "Tsm2: Optimizing tall-and-skinny matrix-matrix multiplication on gpus," in Proceedings of the ACM International Conference on Supercomputing (ICS), 2019.

[9] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie, "Efficient Tensor Core-Based GPU Kernels for Structured Sparsity Under Reduced Precision," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2021.

[10] J. Choi, H. Li, B. Kim, S. Hwang, and J. H. Ahn, "Accelerating transformer networks through recomposing softmax layers," in 2022 IEEE International Symposium on Workload Characterization (IISWC), 2022.

[11] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 Tensor Core GPU: Performance and Innovation," Micro, IEEE, vol. 41, no. 2, pp. 29–35, 2021.

[12] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and Programmability," Micro, IEEE, vol. 38, no. 2, pp. 42–52, 2018.

[13] Z. Dai, Z. Yang, Y. Yang, W. W. Cohen, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context," arXiv:1901.02860, 2019.

[14] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "GPT3.int8(): 8-bit matrix multiplication for transformers at scale," in Advances in Neural Information Processing Systems, 2022.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL), 2019.

[16] N. Ding, Y. Qin, G. Yang, F. Wei, Y. Zonghan, Y. Su, S. Hu, Y. Chen, C.-M. Chan, W. Chen, J. Yi, W. Zhao, X. Wang, Z. Liu, H.-T. Zheng, J. Chen, Y. Liu, J. Tang, J. Li, and M. Sun, "Parameter-efficient fine-tuning of large-scale pre-trained language models," Nature Machine Intelligence, 2023.

[17] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," in Proceedings of the International Conference on Learning Representations (ICLR), 2021.

[18] O. Dusek, J. Novikova, and V. Rieser, "Findings of the E2E NLG challenge," in Proceedings of the 11th International Conference on Natural Language Generation (INLG), 2018.

[19] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training compression for generative pretrained transformers," arXiv:2210.17323, 2022.

[20] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse GPU Kernels for Deep Learning," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2020.

[21] P. Gao, J. Han, R. Zhang, Z. Lin, S. Geng, A. Zhou, W. Zhang, P. Lu, C. He, X. Yue, H. Li, and Y. Qiao, “Llama-adapter v2: Parameter-efficient visual instruction model,” arXiv:2304.15010, 2023.

[22] C. Gardent, A. Shimorina, S. Narayan, and L. Perez-Beltrachini, “Creating training corpora for NLG micro-planners,” in Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers, 2017.

[23] Y. Gong, L. Liu, M. Yang, and L. Bourdev, “Compressing deep convolutional networks using vector quantization,” arXiv:1412.6115, 2014.

[24] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. De Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, “Parameter-efficient transfer learning for NLP,” in Proceedings of the 36th International Conference on Machine Learning, 2019.

[25] Y.-C. Hsu, T. Hua, S. Chang, Q. Lou, Y. Shen, and H. Jin, “Language model compression with weighted low-rank factorization,” in International Conference on Learning Representations (ICLR), 2022.

[26] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “Lora: Low-rank adaptation of large language models,” arXiv:2106.09685, 2021.

[27] J.-Y. Jiang, C. Xiong, C.-J. Lee, and W. Wang, “Long Document Ranking with Query-Directed Sparse Transformer,” in Proceedings of the Empirical Methods in Natural Language Processing (EMNLP), 2020.

[28] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” arXiv:2001.08361, 2020.

[29] S. J. Kwon, J. Kim, J. Bae, K. M. Yoo, J.-H. Kim, B. Park, B. Kim, J.-W. Ha, N. Sung, and D. Lee, “AlphaTuning: Quantization-aware parameter-efficient adaptation of large-scale pre-trained language models,” in Findings of the Association for Computational Linguistics: EMNLP 2022, 2022.

[30] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, “Albert: A lite bert for self-supervised learning of language representations.” in ICLR, 2020.

[31] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, “BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension,” in Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL), 2020.

[32] H. Li, J. Choi, and J. Ahn, “A Slice and Dice Approach to Accelerate Compound Sparse Attention on GPU,” in IEEE International Symposium on Workload Characterization (IISWC), 2022.

[33] H. Li, J. Choi, S. Lee, and J. H. Ahn, “Comparing bert and xlnet from the perspective of computational characteristics,” in 2020 International Conference on Electronics, Information, and Communication (ICEIC), 2020.

[34] L. Liu, Z. Qu, Z. Chen, Y. Ding, and Y. Xie, “Transformer Acceleration with Dynamic Sparse Attention,” arXiv:2110.11299, 2021.

[35] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, “Swin Transformer: Hierarchical Vision Transformer Using Shifted Windows,” in Proceedings of the International Conference on Computer Vision (ICCV), 2021.

[36] J. Luitjens, “CUDA Streams: Best Practices and Common Pitfalls,” https:// on-demand.gputechconf.com/ gtc/ 2014/ presentations/ S4158-cuda-streams-best-practices-common-pitfalls.pdf, 2014.

[37] M. Milakov and N. Gimelshein, “Online Normalizer Calculation for Softmax,” arXiv:1805.02867, 2018.

[38] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius, “Accelerating Sparse Deep Neural Networks,” arXiv:2104.08378, 2021.

[39] L. Nan, D. Radev, R. Zhang, A. Rau, A. Sivaprasad, C. Hsieh, X. Tang, A. Vyas, N. Verma, P. Krishna, Y. Liu, N. Irwanto, J. Pan, F. Rahman, A. Zaidi, M. Mutuma, Y. Tarabar, A. Gupta, T. Yu, Y. C. Tan, X. V. Lin, C. Xiong, R. Socher, and N. F. Rajani, “DART: Open-domain structured data record to text generation,” in Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL), 2021.

[40] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, “CUSPARSE Library,” in GPU Technology Conference, 2010.

[41] T. Nguyen, M. Rosenberg, X. Song, J. Gao, S. Tiwary, R. Majumder, and L. Deng, “MS MARCO: A Human Generated Machine Reading Comprehension Dataset,” in Proceedings of the Workshop on Cognitive Computation: Integrating neural and symbolic approaches 2016 co-located with the 30th Annual Conference on Neural Information Processing Systems CoCo@NeurIPS, 2016.

[42] NVIDIA, “Faster Transformer,” https://developer.nvidia.com/gtc/2020/slides/s21417-faster-transformer.pdf, 2020.

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

[44] NVIDIA, “CUDA Toolkit Documentation,” 2021. [Online]. Available: https://docs.nvidia.com/cuda/archive/11.3.1/

[45] OpenAI, “Gpt-4 technical report,” arXiv:2303.08774, 2023.

[46] M. Osama, D. Merrill, C. Cecka, M. Garland, and J. D. Owens, “Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the gpu,” in Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP), 2023.

[47] K. Osawa, A. Sekiya, H. Naganuma, and R. Yokota, “Accelerating matrix multiplication in deep learning by using low-rank approximation,” in 2017 International Conference on High Performance Computing and Simulation (HPCS), 2017.

[48] G. Park, B. Park, M. Kim, S. Lee, J. Kim, B. Kwon, S. J. Kwon, B. Kim, Y. Lee, and D. Lee, "Lut-gemm: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models," arXiv:2206.09557, 2023.

[49] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic Differentiation in PyTorch," in Proceedings of Neural Information Processing Systems (NeurIPS), 2017.

[50] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," arXiv:2211.05102, 2022.

[51] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever et al., "Language models are unsupervised multitask learners," OpenAI blog, 2019.

[52] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," Journal of Machine Learning Research (JMLR), vol. 21, no. 140, pp. 1–67, 2020.

[53] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "DeepSpeed: System Optimizations Enable Training Deep Learning Models with over 100 Billion Parameters," in Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2020.

[54] R. Sennrich, B. Haddow, and A. Birch, “Neural Machine Translation of Rare Words with Subword Units,” in Proceedings of the Association for Computational Linguistics (ACL), 2016.

[55] U. Shaham, E. Segal, M. Ivgi, A. Efrat, O. Yoran, A. Haviv, A. Gupta, W. Xiong, M. Geva, J. Berant, and O. Levy, “SCROLLS: Standardized Comparison over Long Language Sequences,” arXiv:2201.03533, 2022.

[56] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, “Long Range Arena : A Benchmark for Efficient Transformers,” in Proceedings of the International Conference on Learning Representations (ICLR), 2021.

[57] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, “Efficient Transformers: A Survey,” arxiv:2009.06732, 2020.

[58] P. Tillet, H. T. Kung, and D. Cox, “Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations,” in Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL), 2019.

[59] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozi□re, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, “Llama: Open and efficient foundation language models,” arXiv:2302.13971, 2023.

[60] H. Vanholder, “Efficient Inference with TensorRT,” https://on-demand.gputechconf.com/ gtc-eu/ 2017/ presentation/ 23425-han-vanholder-efficient-inference-with-tensorrt.pdf, 2016.

[61] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention Is All You Need," in Proceedings of Neural Information Processing Systems (NeurIPS), 2017.

[62] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, "Transformers: State-of-the-Art Natural Language Processing," in Proceedings of the Empirical Methods in Natural Language Processing (EMNLP), 2020.

[63] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation," arXiv:1609.08144, 2016.

[64] C. Yang, A. Buluc□and J. D. Owens, "Design Principles for Sparse Matrix Multiplication on the GPU," arXiv:1803.08601, 2018.

[65] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized Autoregressive Pretraining for Language Understanding," in Proceedings of Neural Information Processing Systems (NeurIPS), 2019.

[66] Z. Yang, P. Qi, S. Zhang, Y. Bengio, W. Cohen, R. Salakhutdinov, and C. D. Manning, "HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering," in Proceedings of the Empirical Methods in Natural Language Processing (EMNLP), 2018.

[67] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big Bird: Transformers for Longer Sequences," in Proceedings of Neural Information Processing Systems (NeurIPS), 2020.

[68] R. Zhang, J. Han, A. Zhou, X. Hu, S. Yan, P. Lu, H. Li, P. Gao, and Y. Qiao, "Llama-adapter: Efficient fine-tuning of language models with zero-init attention," arXiv:2303.16199, 2023.

국문초록

최근 Transformer 기반의 모델들은 자연어 처리와 컴퓨터 비전 등 다양한 분야에서 높은 성능을 보여주고 있다. 기존 강력한 모델들은 커지면서 모델이 복잡한 데이터 관계를 학습하고 나타낼 수 있게 된다. 또한 입력 시퀀스 길이를 늘려 문맥 학습을 향상시켜 복잡한 문제도 효과적으로 해결한다. 다만 이러한 모델들은 큰 메모리 사용량, 어텐션 레이어에서 입력 길이에 의한 2차복잡도 문제, 또한 커널 최적화가 되어 있지 않아 높은 추론 비용을 야기한다.

본 논문에서는 Transformer 기반 모델들의 크기, 입력 시퀀스 길이, 배치 크기에 따라 추론 비용을 줄이는 최적화 방법을 제안한다. 먼저, 입력 길이(L_{in})가 은닉 차원(D_h)보다 큰 시나리오를 최적화하는 Multigrain 방법을 제안한다. 기존 희소 어텐션 기법은 긴 입력 시퀀스에서 연산량과 메모리 사용량을 효과적으로 줄일 수 있지만 GPU에서 비효율적으로 처리되며 여전히 대부분 수행시간을 차지한다. Multigrain은 희소 어텐션의 복합적인 희소 패턴을 파악하고 거친 희소 패턴은 고성능 텐서 코어를 사용한 커널로 처리하고 세밀한 패턴은 CUDA 코어를 사용한 커널로 각각 멀티 스트림으로 동시에 처리한다. 그 결과로 Longformer 모델을 DeepSpeed에서 추론을 실행한 기준 시스템에 비해 2.07배 더 빠른 것을 보여준다.

그리고 본 논문에서는 L_{in}이 D_h와 비슷하거나 작은 시나리오에서 추론 비용을 줄이는 tiled singular value decomposition(TSVD) 방법을 제안한다. TSVD는 행렬을 타일로 나누고 각 타일을 특이값 분해(SVD)하며 저랭크 근사를 이용하여 행

렬을 압축하는 기법이다. Transformer 기반 모델에서 어텐션 레이어와 피드포워드 레이어의 기본 연산인 행렬 곱을 저랭크 근사를 이용한 TSVD기반의 행렬 곱으로 수행하면 메모리 사용량을 줄일 수 있고 연산량도 줄일 수 있으므로 추론 비용을 상당히 줄일 수 있다. 결과적으로 행렬을 2배 8배까지 압축 시, TSVD기반의 행렬 곱은 압축하지 않은 행렬 곱보다 1.02배-2.26배 빠른 것을 보인다. 다만 모델에 적용 시 수행시간이 줄어들지만 정확도가 하락하는 문제점이 존재한다.

이러한 문제점을 해결하기 위해 본 논문에서는 TSVD 기반의 매개변수 효율적 미세조정(parameter efficient fine-tuning) 방법인 TSVD-common을 제안한다. 각 타일에서 SVD로 분리된 두 서브행렬들 중 하나를 모든 타일에서 공유하는 형태로 하고 공동의 해당 서브행렬만 미세조정 시켜 학습시키는 방법이다. 결과적으로 제안한 TSVD-common은 GPT2 모델에서 2배 또는 4배 압축 시 E2E 태스크에서는 압축하지 않은 전체 매개변수를 미세조정하는 방법(full fine-tuning)보다 정확도가 2%정도 향상되었고 매개변수 효율적 미세조정 최신 방법인 LoRA와 근접한 정확도를 보여준다.

주요어: 트렌스포머 가속, 모델 최적화, 희소 어텐션, 매개변수 효율적 미세조정, 모델 압축

학번: 2017-37405