A Slice and Dice Approach to Accelerate Compound Sparse Attention on GPU

Hailong Li Jaewan Choi Jung Ho Ahn

Seoul National University, Seoul, South Korea
{lhl2017, cjw9202, gajh}@snu.ac.kr

Abstract

Transformer-based models are vital to various research domains, including NLP, computer vision, and recommendation systems. However, because the attention mechanism with quadratic complexity limits computation and memory footprint in long sequences, numerous sparse attention-based transformer models are proposed to alleviate these problems. To efficiently infer these models on GPUs, prior solutions such as Triton and Sputnik have accelerated sparse attention in the sparse transformers, including optimizations for sampled dense-dense matrix multiplication (SDDMM), sparse softmax (SpSoftmax), and sparse matrix-matrix multiplication (SpMM). Although the existing optimizations have achieved impressive improvement, they have shortcomings, including low data reuse, unnecessary computation, and unnecessary memory accesses in the latest sparse transformer models based on compound sparse attention.

We propose Multigrain to accelerate sparse attention by applying compound sparse GPU kernels with multi-stream for compound sparse patterns. First, we take sparse patterns with high spatial locality and execute them on our customized coarse-grained kernels, which exploit data reuse and high-performance tensor core units. Second, we execute the sparse patterns with low spatial locality on the fine-grained kernels to reduce unnecessary computation and memory accesses. Last, we process the coarse-grained and fine-grained kernels in parallel using multi-stream for the SDDMM and SpMM. In the latest sparse transformer models such as Longformer and QDS-transformer, Multigrain achieved 2.07× and 1.55× end-to-end speedup, respectively, over Triton, which only uses the coarse-grained method. We also achieved 2.08× and 1.08× end-to-end speedup, respectively, compared to Sputnik, which only uses the fine-grained kernel.

1. Introduction

Transformer-based models have been leading the performance (quality) of natural language processing (NLP) problems such as question-answering (QA) [4, 12, 43, 46], machine translation [5, 39], and summarization [17, 29]. They achieve excellent research results in numerous other fields, including image classification [13, 19], speech recognition [2], recommendation [11, 33], and genomics [46]. 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, 34, 35]. 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 [32, 34]. The existing studies [10, 12] 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, 46] 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 \(L^2\) when the input length is \(L\). For \(L = 4096\), BERT-large [12] 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 transformers 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, 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.
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 for the fine-grained part to reduce unnecessary computation and memory access. We execute the coarse-grained and fine-grained kernels using multistream; hence they can run concurrently in different SMs to utilize the hardware resources better.

Multigrain outperformed the latest sparse transformers such as Longformer and QDS-Transformer up to 2.08× and 1.68× on the latest GPUs (A100 and RTX3090). We also evaluated our proposed method in the compound SA with various sparse patterns. We achieved 1.73×–2.34×, 5.06×–12.63×, and 1.79×–3.04× speedup in the sampled dense-dense matrix multiplication (SDMM), sparse softmax (SpSoftmax), and sparse matrix-matrix multiplication (SpMM), respectively, over Triton [36], which only uses the coarse-grained method. Moreover, we achieved 1.34×–5.81×, 1.26×–2.82×, and 1.23×–5.24× speedup compared to Sputnik, which only uses the fine-grained method.

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

2. Background and Motivation

2.1. Graphics Processing Unit

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 [8, 9].

NVIDIA GPUs introduce the concept of streams [20], 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.

2.2. Sparse operations in the sparse transformer

Transformer-based models have been proven effective in several fields. However, due to the quadratic complexity of the attention mechanism, the computation and the memory footprint increase proportionally to \( L^2 \) as the sequence length \( L \) increases. Recently, not only NLP but also computer vision has required long sequence processing. In document-based NLP problems, practical documents such as scientific literature and digital humanities consist of 100K or more words, where a word is one or more tokens. In computer vision, as the image resolution increases, input sequences grow longer accordingly. If a 4K image is processed with a 32×32 patch size (one token), it is regarded as an input sequence length of 8K. However, existing models often fail at capturing the contextual representations of potentially
important cross-partition information due to partitioning or shortening long sequences. Also, training a sequence length of 4096 or more is difficult with the existing hardware due to the larger memory demand for longer sequences. Sparse transformers alleviated these problems through SA, and recently numerous studies have been actively conducted, reporting state-of-the-art (SOTA) results [4, 35, 46].

In a sparse transformer, SA is performed with multiple heads in the same way as the multi-head attention [39] in a typical transformer. In other words, in an input matrix of $L \times D_m$, a single row vector of $1 \times D_m$ is split by the number of heads, and the matrix of $L \times D_h$ generated by applying it to other row vectors is repeatedly applied in parallel with a single head SA. Here, $L$ represents the sequence length, $D_h$ represents the head dimension of a single head, and $D_m$ (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). 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 $L \times D_h$, 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 $D_m$ are stacked in $L$ and are calculated by multiplying them with different weight matrices of $D_m \times D_m$. The hidden states refer to the input sequence consisting of $L$ tokens, an $L \times D_m$ 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 [31, 41]).

Scaling is an element-wise operation; a sparse matrix calculated by SDDMM is multiplied by scaling factors ($SF = L_h / m$). It alleviates the gradient vanishing problem, pushing the softmax function into regions with extremely small gradients as the result of SDDMM growing large in magnitude [39].

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.

### 2.3. Compound sparse patterns

SA-based sparse transformers use various sparse patterns (see Fig. 3), taking into account the compute efficiency and the inductive biases of languages and images. A local pattern attends the window-sized tokens on both sides with the current token. It is mainly for the language or image tasks considering their inductive bias as a basic pattern in the most sparse transformers.

A global/selected pattern attends the current token to all tokens with a one-to-all or all-to-one fashion, which is the special case of the dense pattern that uses an all-to-all fashion. It is usually used when the current token is a special token to connect local information globally. Special tokens distinguish between the start and end of a sentence, paragraph, or document, as well as between questions and documents. Therefore, their locations are highly dependent on the input sequence.

In a dilated pattern, the current token pays attention to the tokens located at a constant stride distance. It broadly represents the receptive field, an input space where the output element contributes significantly from the input feature, using the same computation [45]. It is mostly used as a dilated local pattern combined with a local pattern.
A random pattern represents that the current token attends to the random token, reflecting the interactions with tokens except for the adjacent one. A blocked pattern represents that the tokens are all-to-all connected within a corresponding block after the input sequence is divided into several non-overlapping blocks.

The latest sparse transformers use compound-sparse patterns that combine multiple atomic sparse patterns rather than a single atomic sparse pattern, and they have recorded SOTA results for the various tasks. For example, Longformer combined local, selected, and global patterns recording SOTA in triviaQA [16] and wikihop [37]. QDS-Transformer that combined local and selected patterns recorded an impressive accuracy improvement in the TREC Deep Learning Track Document Ranking task [28]. Sparse transformer models such as Big Bird-ETC [46] and Poolingformer [47] also recorded SOTA on several tasks.

2.4. Existing GPU solutions and their limitations

Existing GPU solutions are efficient in sparse operations with specific computational characteristics. However, they experience several limitations on the compound-sparse-pattern-based sparse operations as they have not considered the locality of each atomic-sparse pattern in the compound-sparse patterns. The limitations include unnecessary computation, unnecessary memory accesses, poor reuse of data, and hardware under-utilization problems.

There are two methods to perform sparse operations efficiently. One is a coarse-grained method using a blocked sparse format (e.g., BCOO and BSR). The other is a fine-grained method using an element-wise sparse format (e.g., CSR, COO, and CSC).

The coarse-grained method performs a matrix multiplication using a blocked sparse format representing a sparse matrix by uniform non-zero blocks and other metadata such as row indices, row offsets, and column indices. It treats each non-zero block as a dense matrix and regards the operation as a dense operation within the blocks. It increases data reuse and maximizes computational throughput by utilizing the tensor cores. However, applying the coarse-grained method to unstructured patterns with low locality causes unnecessary computation and memory access. If we represent the sparse matrix with the low locality in the blocked sparse format, non-zero blocks are mostly sparse blocks with a low degree of sparsity; hence the invalid elements of the block are also included in the operation. DeepSpeed [30], implemented by OpenAI using Triton, uses the coarse-grained method, so long as cuSPARSE library [23] of NVIDIA. In Triton, SDDMM uses BCOO, whereas SpMM uses BSR, and NVIDIA’s cuSPARSE library provides an API for handling SpMM as the blocked-ELL format.

By contrast, the fine-grained method processes SDDMM and SpMM using the element-wise sparse format (e.g., CSR, COO, and CSC). The non-zero elements are the only valid elements; hence there is no unnecessary data access or computation. Suppose we process the sparse operations of the structured pattern with the high locality in a fine-grained method. In that case, we cannot fully utilize the characteristics of locality and hardware resources. The fine-grained method targets sparse operations with unstructured sparsity, where it neither benefits from data reuse nor utilizes high-performance tensor cores. The representatives are Google’s Sputnik library using CSR and NVIDIA cuSPARER library using the CSR/CSC sparse formats.

Besides the two methods mentioned above, there are sliding chunk [4] and blockify [46] methods that efficiently process the sparse operations of particular sparse patterns. They are only available for local patterns and blocked local patterns. The methods consider the features of local or blocked local patterns consisting of overlapped or non-overlapped blocks to convert sparse matrices into smaller matrices and compute them as general matrix multiplication (GEMM) operations.

Although these methods can only access and perform operations on data in the valid element part and fully utilize the existing hardware that targets dense operations, they suffer from significant memory copy overheads needed for pre-processing and post-processing. For example, Longformer implements the sliding chunk method for local patterns (see Fig. 3), where the existing dense matrix is split in the row direction by the window size into blocks. The two split blocks are combined into one single chunk, and neighboring chunks share an overlapped block (called chunking).

3. The coarse-grained method implemented by OpenAI using Triton is hereinafter referred to as Triton.
3. Multigrain

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. 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. 4).

First, we classify and group 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.

Because the overlapped blocks are duplicated, they consume $2 \times$ the amount of memory. BigBird accommodates a blocked local pattern consisting of non-overlapped blocks with the blocking method where the non-overlapping blocks do not cause memory consumption without duplicate allocation. However, the chunked matrix is copied to the three equally structured dense matrices to process blocked local patterns. The first and the last matrices are rolled in the up and down directions of row dimension, respectively, to create a new matrix and combine it with the middle one. In other words, the stacking process creates a new right-hand side matrix, which incurs three times the memory consumption than the original matrix size.
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 3.3. 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 pattern parts using CUTLASS [1] kernels and perform SpSoftmax using the TensorRT’s [38] softmax kernel because these libraries perform more efficiently than Sputnik.

3.2. 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. 12).

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 [42]. In the blocked row-splitting scheme, we assign each row block in the output matrix represented by BSR to a single TB. Fig. 5 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-level 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 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. 5) to reuse registers (REG). If the warps inside a TB use too much of REG, we enforce
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 hierarchy 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 [14], 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. 6 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.

### 3.3. 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 [21]: 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 are global 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 sparse softmax kernel for other pattern parts in parallel.

4. 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 [40] and a base, officially-release model of QDS-Transformer. We measured the end-to-end execution time of Longformer using the hotpotQA [44] dataset, and of QDS-Transformer using the Microsoft MAchine Reading Compensation (MSMARCO) [24] dataset.

We run the PyTorch 1.8.2 framework [27] on two GPUs with different computational characteristics (A100 [25] and GeForce RTX 3090 [6]) running CUDA toolkit 11.3 [26]. Hardware specifications are shown in Table 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. 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. In SDDMM, the 1D tiling scheme wastes the SM 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.

5. 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.

5.1. Sparse Transformer

Multigrain accelerated Longformer and QDS-Transformer by optimizing compound SA during the inference. Fig. 7 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

### Table 1: Specifications of the GPUs used in the evaluation.

<table>
<thead>
<tr>
<th>GPU</th>
<th>Memory Bandwidth (GB/s)</th>
<th>TFLOPS (FP16 CUDA core)</th>
<th>TFLOPS (FP16 Tensor core)</th>
<th>L1 D$ per SM (KB)</th>
<th>L2 (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A100</td>
<td>1,555</td>
<td>42.3</td>
<td>169</td>
<td>192</td>
<td>40</td>
</tr>
<tr>
<td>RTX 3090</td>
<td>936.2</td>
<td>29.3</td>
<td>58</td>
<td>128</td>
<td>6</td>
</tr>
</tbody>
</table>

4. 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.

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

As a batch size increases (Fig. 8), Multigrain gains additional performance improvement because hardware resources can be utilized more efficiently. It shows up to $2.34 \times$ and $1.82 \times$ speedups compared to Triton and $2.13 \times$ and $1.17 \times$ 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 1. Therefore, Sputnik showed a greater performance improvement than Triton on RTX3090. In Longformer and QDS-Transformer, it shows $1.10 \times$ and $1.64 \times$ 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 $1.58 \times$ and $1.44 \times$ 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 $1.68 \times$ and $1.02 \times$ faster than Triton and Sputnik, respectively.

5.2. Compound 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.

5.2.1. 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. 9). 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. 9). 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 performs 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.

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. 9).
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

Figure 9: 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.

5.2.2. Compound 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. 10). In the former three
compound sparse patterns without a global pattern, our
customized kernel has 1.26×–1.31× performance improvements
over Sputnik, whereas it has 7.09×–12.63× 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. 10), our customized kernel achieves 2.20×–
2.82× and 5.06×–7.48× 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 10: Speedup of our compound sparse softmax kernel
(Multigrain) compared to Sputnik and Triton with various
compound-sparse patterns in SpSoftmax on A100.

not exploiting the thread-level parallelism when a kernel
allocates fewer TBs that perform a large tile for small-sized
matrix multiplications.
5.3. 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 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. 11). 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 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. 12, 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×.

6. Related work

6.1. Coarse-grained methods

Many studies have optimized the sparse attention on GPUs by applying the blocked sparse format. Deep-
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 [22] 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 [7] 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.

7. Conclusion

We have proposed Multigrain to enable acceleration of compound-sparse attention by reducing the execution time of the sparse transformers. Multigrain considers the locality of the sparse patterns and divides the sparse operations of the compound-sparse patterns into coarse-grained and fine-grained parts. We process the two parts separately with the corresponding kernels to accelerate sparse attention. We process the coarse-grained part using a blocked sparse format to take advantage of the high spatial locality in the sparse patterns. We process the fine-grained part using an element-wise sparse format, where only the valid elements are processed. Even with the low spatial locality, the fine-grained kernel does not suffer from unnecessary computation or memory accesses. Our proposed method retains the advantages of both the coarse-grained and fine-grained kernels, whereas we avoid their disadvantages. Compared to Triton, which only uses the coarse-grained kernel, Multigrain significantly reduces unnecessary computation and memory accesses. Compared to Sputnik, which only uses the fine-grained kernel, Multigrain can enable efficient data reuse and exploit the power of the high-performance tensor cores.

Acknowledgments

This research was supported by Samsung Advanced Institute of Technology, Samsung Electronics Co., Ltd., the Engineering Research Center Program through the NRF of Korea funded by the Korean Government MSIT (NRF-2018R1A5A1059921), and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) [NO.2021-0-01343, Artificial Intelligence Graduate School Program (Seoul National University)]. Jung Ho Ahn, the corresponding author, is with the Department of Intelligence and Information, and the Interdisciplinary Program in Artificial Intelligence. The ICT (Institute of Computer Technology) at Seoul National University provides research facilities for this study.

References


