Accelerating Transformer Networks through Recomposing Softmax Layers

Jaewan Choi† Hailong Li† Byeongho Kim‡ Seunghwan Hwang† Jung Ho Ahn†

†*Seoul National University, Seoul, South Korea*
{cjw9202, lhl2017, seunghwan149, gajh}@snu.ac.kr
‡*Samsung Electronics, Hwasung, South Korea*
bh1122.kim@samsung.com

Abstract

The transformer model has become a crucial deep learning model as it provides accuracy superior to that by conventional models in various domains, such as image processing, genomics, and natural language processing. Recently, the importance of the softmax layer has increased as a growing number of transformer models process longer sequences to improve their accuracy rates. However, it is difficult to accelerate a softmax layer with methods introduced in previous studies due to the differences in the data access patterns of softmax layers and other adjacent layers. We address this challenge by accelerating the softmax layer through a recomposition strategy. By decomposing the softmax layer into multiple sub-layers, we change its data access pattern. Then, we fuse the decomposed softmax sub-layers with the subsequent and preceding operations. Softmax recomposition achieves up to 1.25×, 1.12×, 1.57×, and 1.65× speedups in inferring BERT, GPT-Neo, BigBird, and Longformer on a modern GPU by significantly reducing the off-chip memory traffic.

1. Introduction

The transformer model has recently played a key role in developing deep neural networks across various domains. Since the advent of the vanilla transformer [40], several variants, such as Google's BERT [8] and OpenAI's GPT [34], have been proposed. The accuracy of a transformer surpasses human performance. It is easily transferred to various natural language processing (NLP) tasks, such as sequence-to-sequence, summarization, and question and answer tasks.

These capabilities now mean that the transformer model is a vital operation actively used by various industries and researchers. Many more operations are required to process the transformer model compared to existing CNN models, such as ResNet [12]. Therefore, researchers have been actively trying to accelerate the transformer model using various types of hardware, such as GPU [17, 19, 25, 36, 39], FPGA [18, 31, 32], and ASIC [11, 42]. In particular, various optimized libraries, such as TensorRT [39], Faster Transformer [25], and DeepSpeed [36], have been proposed for the GPU, which is one of the most widely used hardware components.

Recently, the transformer models have shown increased input sequence lengths to improve the prediction accuracy. A transformer receives a sequence of tokens as input and finds a correlation between the tokens through an attention mechanism. To achieve higher accuracy, it is crucial to create a better contextual representation of the input sequence. Recent studies [3, 44] show that increasing the sequence length can improve the accuracy by increasing the amount of information in the contextual representation. Thus, the importance of a transformer model to process long sequences is growing.

As the sequence length increases in the transformer model, the execution time portion of the multi-head attention (MHA) block, a key operation of the transformer model, also increases. MHA identifies the attention (relationship) between the tokens in a sequence through operations such as matrix multiplication and normalization on the matrices containing the sequence information. The amount of computation and memory footprint required for processing the MHA block scales quadratically in proportion to the sequence length. In contrast, operations other than the MHA block are linearly proportional to the sequence length.

Unlike other layers, the softmax layer in MHA is challenging to accelerate by conventional methods such as kernel fusion. The softmax layer divides each value's exponent by the sum of a group of values' exponents, normalizing each value. The matrix multiplication (MatMul) operation in MHA can be accelerated by using a tensor core with high computing power in modern GPUs. Element-wise layers such as the scale and mask layer can be optimized through the kernel fusion methods [41] proposed in previous studies. However, for the softmax layer, it is difficult to optimize these layers with existing methods due to the differences in data access patterns with the adjacent layers. Therefore, the softmax layer has become more important, accounting for a significant portion of the total execution time.

Sparse attention [7] has been directed to alleviate the problem of the quadratic growth of computational and memory complexity. It shows high accuracy even if each token is only related to a small number of tokens in a sequence. Therefore, an MHA block is processed sparsely to have a computational amount proportional to the sequence

length. However, even when using sparse attention, the softmax layer still accounts for more than 40% of execution time of transformers.

We propose to reduce the execution time of softmax substantially by recomposing it. First, we decompose the softmax layer into three sub-layers, after which we fuse the first sub-layer to its preceding MatMul operation. Lastly, we fuse the last sub-layer to its following MatMul operation. The decomposed softmax sub-layers perform identically to the existing softmax layer in terms of mathematics. By applying softmax recomposition to the inference of various transformer models on the latest GPU, we achieved higher performance than state-of-the-art implementations using TensorRT and DeepSpeed. Consequently, we achieved a reduction in the per-inference latency by 28% and in the off-chip access energy by 29% on average without any GPU hardware modifications.

Our key contributions are as follows:

- We identify the importance of the softmax layer in the transformer model and show that the softmax layer cannot be optimized with existing optimization techniques due to heterogeneity in its data access patterns.
- We decompose the softmax layer into sub-layers to match their data access patterns with adjacent layers. Then, we fuse the sub-layers into their preceding and subsequent operations, significantly reducing the off-chip memory access of the softmax layer.
- We accelerate the inference process up to 1.25×, 1.12×, 1.57×, and 1.65× compared to the baselines for various transformer models (BERT, GPT-Neo, BigBird, and Longformer) on the latest GPU.

2. Background and Motivation

2.1. Transformer model architecture

A vanilla transformer [40] is a sequence-to-sequence model consisting of an encoder and a decoder, receiving a sequence of tokens as input and producing a translated sequence of tokens (e.g., a sentence in English as input and a sentence translated into German as output). The encoder converts an embedding matrix generated from an embedding layer by processing an input sequence into a hidden matrix. The decoder decodes the hidden matrix and generates a translated sequence. Transformer models are classified into autoencoder models (e.g., BERT) that use only encoders, autoregressive models (e.g., GPT) that use only decoders, and sequence-to-sequence models that use both encoders and decoders. The encoder and decoder have multi-head self-attention (MHA) blocks and FeedForward (FF) blocks as building blocks. Both have a structure in which MHA and FF blocks are stacked repeatedly, with residual connection and normalization layers stacked between them. Therefore, transformer models are structured similarly regardless of their classification.

An MHA block receives three matrices as input and produces one matrix (see Fig. 1). Each input and output matrix is a hidden matrix representing a sequence with L tokens. We refer to the sequence length as L. Each $L \times D_m$ matrix consists of L hidden vectors, each representing a single token, and one hidden vector with a size of D_m. The MHA block consists of a scaled dot-product attention (SDA) block, three parallel fully connected (FC) layers generating three inputs of the SDA block from the inputs of the MHA block, and an FC layer that post-processes the output matrix of the SDA block. Each FC layer preceding the SDA block generates an $L \times D_m$ matrix by performing MatMul with different $D_m \times D_m$ weight matrices, using it as the input of the SDA block. The SDA block receives three $L \times D_m$ matrices and produces one $L \times D_m$ matrix. The FC layer following the SDA block generates an $L \times D_m$ matrix by performing MatMul with a $D_m \times D_m$ weight matrix, and the MHA block outputs the resulting matrix.

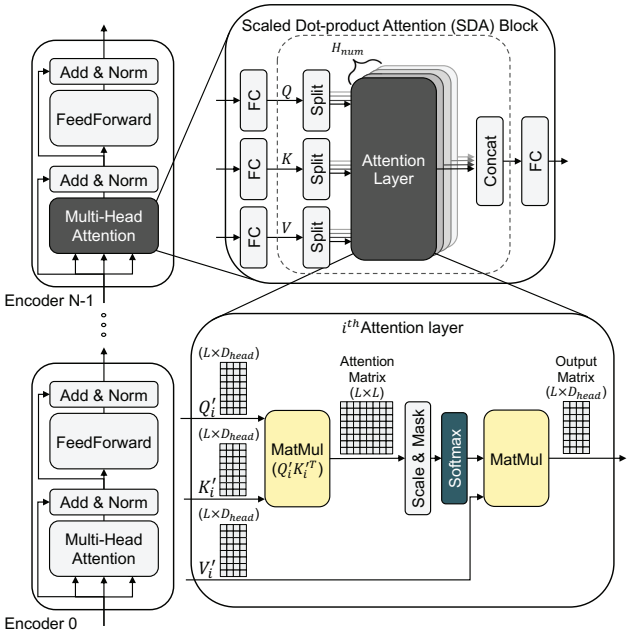

Figure 1: An encoder example in the transformer model. The scaled dot-product attention (SDA) block in the multi-head self-attention (MHA) block is a key operation of the transformer model.

The SDA block is the key operation of an MHA block. An SDA block receives three matrices of the same size as the input, referred to as a Query (Q), Key (K), and Value (V). The SDA block first splits Q, K, and V into multiple matrices through a Split layer, passes the split Q, K, and V to an array of attention layers, and finally concatenates the matrices generated from the array of attention layers through a Concat layer. This method of dividing the input, processing it in parallel, and merging the results is called the multi-head. We denote the number of heads, identical to the number of attention layers in the array, as H_{num}.

The SDA block splits the Q, K, and V matrices along the hidden vector axis and generates H_{num} matrices Q'_i, K'_i, and V'_i, each with a size of $L \times D_{head}$, where $D_{head} = D_m / H_{num}$, $0 \leq i < H_{num}$. A single attention layer initially transposes K'_i and performs MatMul with Q'_i to create an $L \times L$ attention matrix, which represents the

relationships between tokens in a sequence, and then divides each element in the attention matrix by $\sqrt{D_{head}}$. Then, a mask layer is utilized on the attention matrix to make the elements that fall short of certain criteria equal to $-\infty$. The masked attention matrix is passed through the softmax layer for normalization. The resulting matrix generates an $L \times D_{head}$ matrix as the output from the attention layer by performing MatMul with V_i'. Finally, H_{num} matrices resulting from the array of attention layers are concatenated, producing the $L \times D_m$ matrix.

The softmax layer normalizes the elements of the attention matrix to values between 0 and 1. This operation is conducted along the row vector of the attention matrix. Because the range of values representable in actual hardware is limited, safe softmax [21] is adopted to prevent overflow or underflow conditions due to exponent operations. The safe softmax operation subtracts the maximum value of the vector from each element in the vector and then performs the first proposed softmax operation. Let the row vector be X, the maximum value of X be m, the normalization term be d, where $X = \{x_0, x_1, .. , x_{L-1}\}$ and $Y = \{y_0, y_1, .. , y_{L-1}\}$. The formula for safe softmax is as follows, where $m = \max_i(x_i)$, $d = \sum_{i=0}^{L-1} e^{x_i - m}$,

$$y_i = \frac{e^{x_i - m}}{\sum_{i=0}^{L-1} e^{x_i - m}} = \frac{e^{x_i - m}}{d} \quad (1)$$

Strict data dependency exists in the softmax[1] layer as m is required to produce d, and d is required in the process of normalization. Therefore, to process the softmax layer, it is necessary to read the row vector three times in order to determine the values, m and d, and to normalize the vector.

An MHA block receives three matrices as the input, where the actual input matrices depend on whether an MHA block is used as an encoder or decoder. For the former case, all inputs receive the same output matrix from the previous block. For the latter case, there are two situations: one in which all three inputs will receive the output matrix of the previous block, identically to the encoder case, and the other in which one input will receive the output matrix of the previous block, with the two other inputs receiving the matrix produced from the encoder.

An FF block consists of two FC layers and an activation layer. It receives an $L \times D_m$ matrix from the previous block as input and produces a matrix of the same size. The first FC layer performs matrix-vector multiplication on each vector in the L hidden vectors with a $D_m \times D_{ff}$ weight matrix and adds a bias vector, which has a size of D_{ff}. In general, D_{ff} is four times larger than D_m. The activation layer performs an element-wise operation on the matrix, mainly with a GeLU or ReLU operation. The second FC layer operates identically to the first FC layer and produces an $L \times D_m$ matrix.

1. Hereafter, we refer to safe softmax simply as softmax.

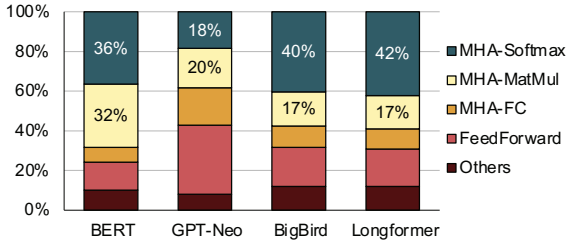

Figure 2: Execution time breakdown of BERT, GPT-Neo, BigBird, and Longformer on an A100 GPU (L = 4,096). Section 4 details the workloads and the configuration.

2.2. Trends in Transformer models

In recent studies, transformer models that process long sequence lengths have been proposed to enhance accuracy of outcomes. With regard to OpenAI GPT [5, 34, 35], a typical network model in NLP tasks, the sequence length increases from 512 to 2048 as the model evolves. Various transformer models, including BigBird [44], Longformer [3], and Sparse transformer [7], have shown that the accuracy improves as the sequence length increases to 4,096 or longer in various fields, such as NLP, genomics, and image processing.

According to [44], as the sequence length increases, the loss of key context data in long documents decreases, thus improving the amount of input information. A transformer model uses the first L tokens of the document as input when the number of tokens exceeds the maximum sequence length (L) of the model (e.g., 512 for BERT). Because the key semantics in long documents can be distributed evenly rather than at the beginning, increasing the sequence length reduces the possibility of missing important semantics. Similarly, increasing the sequence length increases the amount of syntax information, thus further improving the accuracy of the model. However, this also slows down the execution time of the model.

To reduce the execution time and the memory footprint while processing a model with a long sequence length, sparse attention is proposed. When processing a single sequence, a model processing a longer sequence takes more time due to its larger number of arithmetic operations and larger memory footprint. Sparse attention reduces the computation and memory footprint by processing only a portion of the attention matrix, assuming that each token is only related to a small number of tokens, not all. In studies proposing sparse attention [3, 7, 13, 29, 33, 44], various types of sparse attention patterns have been proposed. Sparse attention reduces the memory complexity of the attention matrix in the SDA block from $\mathcal{O}(L^2)$ to $\mathcal{O}(L)$.

2.3. Challenges of the softmax layer in an SDA block

In the GPU environment, which is known to be suitable for processing transformer models, the SDA blocks use more of the execution time of a transformer as the sequence length increases. The computational complexity of the FC

layers in the FF blocks and MHA blocks in the model is $\mathcal{O}(L)$, whereas that of the attention layer in the SDA block is $\mathcal{O}(L^2)$. As L increases, the number of arithmetic instructions and the size of memory footprint increase in proportion to L^2. When BERT with a sequence length of 4,096 is processed on the latest GPU, the SDA block accounts for 68% of the total execution time, which is significant (see Fig. 2).

By applying sparse attention, the amount of computation and the overall memory access time can be greatly reduced, but the proportion of the execution time of the SDA block is still significant. Compared to dense attention models, sparse attention models attempt to reduce the computational complexity of the attention matrices from $\mathcal{O}(L^2)$ to $\mathcal{O}(L)$, which substantially reduce the memory access as well. BigBird, a representative model of sparse attention, reduces the amount of computation in the attention layers to 14.3% compared to BERT. Even so, the execution time of the SDA block in the entire model accounts for approximately 57% of the total.

In an SDA block, the execution time of the softmax layer is as long as that of the MatMul operation. As shown in Fig. 2, the softmax layer in the SDA block uses 36%, 18%, 40%, and 42% of the total execution time of BERT, GPT-Neo, BigBird, and Longformer, respectively. When the operation in the SDA block is processed, the attention matrix is read or written. Considering the BERT-large model where L is 4,096, the attention matrix is 512MB in size for a single batch assuming a half-precision floating-point number per element. Therefore, it cannot be stored in the on-chip memory of even the latest GPU A100, which has an L2 cache of 40 MB (ordinary GPUs have several MBs of L2 cache in general). Therefore, each time a layer is processed in the SDA block, the attention matrix must be read from and written to off-chip memory.

As scale and mask layers perform element-wise operations on the entire attention matrix, it is possible to eliminate their off-chip memory access by combining these layers with the adjacent MatMul operation using a method introduced in [41]. The softmax layer is difficult to optimize simply because there is a strict constraint that memory access must be performed along the row vector of the attention matrix. Moreover, due to the memory wall problem, where the memory bandwidth is less scalable compared to the computational power, the softmax layers could take even more of the total execution time in future GPUs. We suggest reducing the off-chip memory access of the softmax layer and improving the performance of the transformer models by analyzing and optimizing the softmax layer in GPUs.

3. Softmax Layer Recomposition

3.1. A deep dive into the softmax layer

The softmax layer goes through the following steps: it 1) finds the max value (m) for each row vector of an attention matrix, 2) calculates the normalization term (d) by accumulating the exponential result from each element

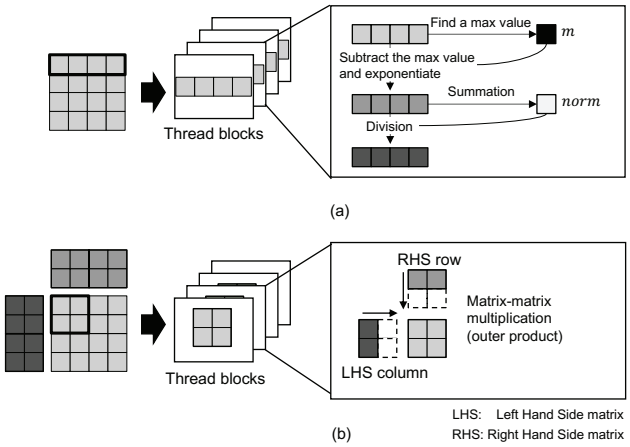

Figure 3: Data access patterns of (a) a softmax layer and (b) a MatMul on a GPU. The left side shows how work is distributed among the thread blocks (TBs). The square allocated within each TB is the working set exclusively processed by that TB. The right side simplifies the operation and required data inside the TB. The arrows on the LHS and RHS of MatMul indicate that the LHS column and the RHS row are read in a streaming manner.

subtracted by m of the row vector, and 3) scales the row vector by $1/d$. All operations except for the first one have strict dependencies on the previous operation, meaning that it is necessary to sweep (read or write) the row vector several times. These dependencies extend the data reuse distance in the softmax layer, causing the GPU to process the softmax layer inefficiently.

As shown in Fig. 3 (a), when processing the softmax layer, a single GPU kernel[2] is launched with each thread block (TB) holding a row vector for maximal data reuse. That is, the row vector required by each TB is stored in the shared memory or the register file of the SM where the TB is scheduled. Therefore, even if there is a long reuse distance between the internal operations of the softmax layer, processing can take place without additional off-chip memory accesses other than loading the attention matrix initially.

Because the softmax layer has a low compute-to-memory access ratio, its execution time is bounded by the off-chip memory traffic to load the attention matrix. Suppose the data type of each element is a half-precision floating-point; the softmax layer performs five operations per element and thus the operational intensity is 2.5 Op/B. On modern GPUs [26, 27], the maximum FLOPS compared to the maximum off-chip memory bandwidth exceeds 25

2. GPU terms are used throughout this paper. A GPU consists of hardware units called streaming multiprocessors (SMs). It operates by scheduling thread blocks (TBs), each representing a group of threads, for the SM. A kernel is a user-specified function called by CPU. A single GPU kernel is composed of several TBs. The shared memory and the register file can be allocated to the TB to store the data on-chip during operation. The allocated memory space can only be shared by the threads inside the TB.

FLOP/B; therefore, the execution time of the softmax layer is determined by the number of off-chip memory accesses.

The mask and scale layers, which are memory-bound similarly to the softmax layer, can eliminate off-chip memory accesses through kernel fusion because those layers are processed in an element-wise manner, exhibiting a simple data access pattern. Thus, kernel fusion can be conducted regardless of the data access pattern of the adjacent kernel. However, as the softmax layer operates for each row vector, whether kernel fusion can be conducted or not is determined by the data access pattern of its adjacent kernels. Therefore, it is necessary to analyze the data access pattern of its neighboring kernel, the MatMul kernel.

Fig. 3 (b) shows how a MatMul kernel operates and the data access pattern within its TB. The MatMul kernel divides its output matrix into square tiles and allocates each tile to a different TB. Each TB uses an outer-product dataflow, a representative type of dataflow for processing MatMul operations in a GPU to maximize data reuse [2]. TBs adopting the outer-product dataflow operate as follows: TBs 1) sequentially load LHS (left hand side matrix) columns and RHS (right hand side matrix) rows into SM, 2) perform the outer-product operation and accumulate the resulting matrix into the output tile, 3) keep the output tile in on-chip memory until the accumulation finishes, and 4) store the output tile into off-chip memory. The output tile kept in on-chip memory must fit the on-chip memory to ensure high data reuse.

It is difficult to perform kernel fusion on the softmax kernel with its preceding and subsequent kernels due to differences in data access patterns between the softmax kernel and its adjacent MatMul kernels. In the MatMul kernel, which precedes the softmax kernel, each TB creates a square matrix that is a portion of the attention matrix, whereas each TB receives the row vector of the attention matrix as input in the softmax kernel. Because two consecutive kernels have different ways of allocating the attention matrix to its TBs and the data stored in the memory space of one TB cannot be shared with other TBs, the two adjacent kernels cannot be fused. The TB of the MatMul kernel following the softmax kernel sequentially loads the columns of the attention matrix in the row direction. Because this data access pattern differs from that of the softmax kernel, fusing softmax with the subsequent MatMul kernel is prohibited.

To solve this problem, our proposal is to decompose the softmax layer into multiple sub-layers so that the sub-layers have the same data access pattern as the adjacent MatMul layers. Then, we fuse some of the decomposed sub-layers with adjacent MatMul operations, thereby reducing off-chip memory access to the attention matrix in the softmax layer.

3.2. Softmax Decomposition

We decompose the softmax layer into three sub-layers: Local Softmax (LS), Inter-sub-vector Reduction (IR), and Global Scaling (GS). All elements in a row vector should be retrieved to obtain the per-row-vector max value and the normalization term prior to normalizing the individual element, which imposes strict dependencies across the elements of the row vector. We alleviate this strict data dependency issue by initially performing the operation within each sub-vector split from a single row vector and then normalizing the value throughout the entire row vector. Let the size of the sub-vector be T, the number of sub-vectors in the row vector be N_{sv}, where $N_{sv} = L/T$, and the kth sub-vector and kth result of the sub-vector correspondingly be $X_k = (x_{k,0}, x_{k,1}, ..., x_{k,T-1})$ and $Y_k = (y_{k,0}, y_{k,1}, ..., y_{k,T-1})$. The softmax decomposition formula is defined as shown below, where we define $m'_k = \max_i(x_{k,i})$, $d'_k = \sum_{j=0}^{T-1}(e^{x_{k,j}-m'_k})$:

$$
\begin{aligned}
y_{k,i} &= \frac{e^{x_{k,i}-m}}{\sum_{j=0}^{L-1} e^{x_j-m}} \\
&= \frac{e^{x_{k,i}-m}}{\sum_{k=0}^{N_{sv}-1}(\sum_{j=0}^{T-1} e^{x_{k,j}-m})} \\
&= \frac{e^{x_{k,i}-m'_k} \times e^{m_k-m}}{\sum_{k=0}^{N_{sv}-1}(e^{m'_k-m} \times \sum_{j=0}^{T-1}(e^{x_{k,j}-m'_k}))} \\
&= \frac{e^{x_{k,i}-m'_k} \times e^{m_k-m}}{\sum_{k=0}^{N_{sv}-1}(e^{m'_k-m} \times d'_k)}
\end{aligned}
\tag{2}
$$

The decomposed softmax mathematically has a result identical to that of the original softmax layer. Softmax decomposition performs a softmax operation for each sub-vector and reconstructs it into the original softmax result by utilizing m' and d' defined for each sub-vector as intermediate values. We decompose the softmax kernel into LS, IR, and GS kernels according to the equation above. The LS kernel undertakes the softmax operation and produces m' and d'. The IR kernel receives m' and d' calculated from the LS kernel as input and performs reduction to calculate m and d. Finally, the GS kernel scales the intermediate value calculated by the LS kernel to obtain a result precisely matching the original softmax.

Fig. 4 shows the operation of the decomposed LS, IR, and GS kernels. The left side shows the work distribution of the TBs for each kernel, and the right side shows the operations and data access patterns within the TBs. Because the softmax operation is performed per row vector, we represent all notations based on the row vector instead of the attention matrix despite the fact that each TB of the kernel is assigned with a two-dimensional matrix. All row vectors in the TB behave in the same way. In the figure, edges represent the dataflows within the TBs and between the kernels.

The LS kernel receives an attention matrix as input and performs the softmax operation for each sub-vector. It divides the attention matrix into square tiles and assigns them to different TBs. Unlike the original softmax kernel, which allocates one row vector per TB, each TB receives the sub-vectors of multiple row vectors as input. Each TB

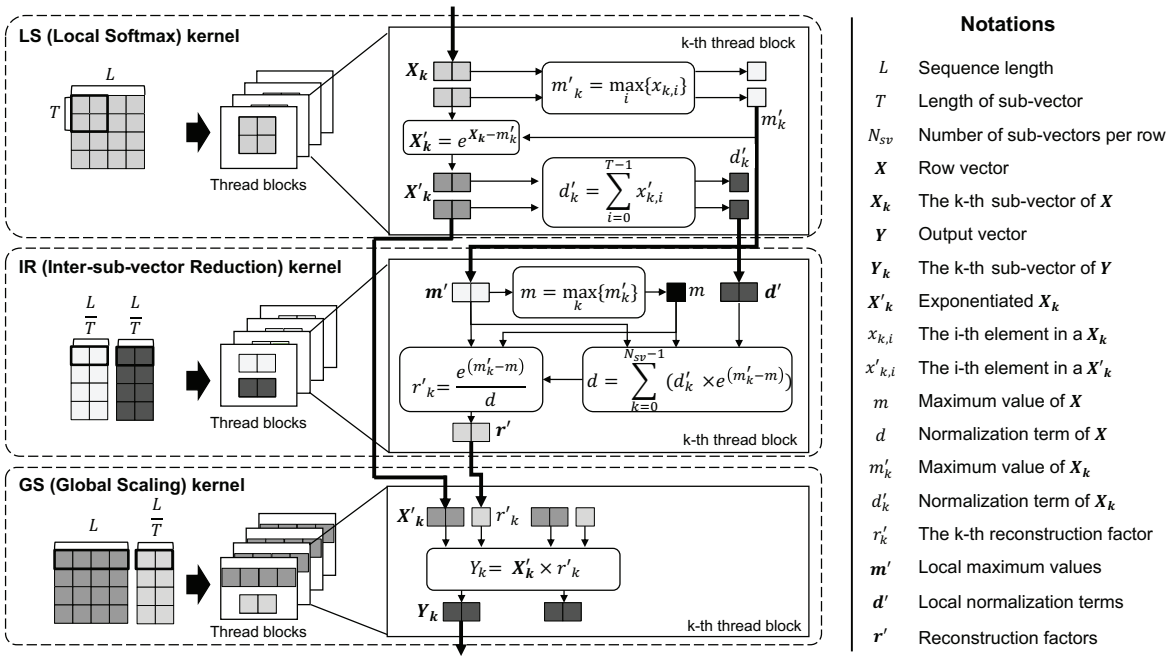

Figure 4: Illustration of the softmax decomposition. A softmax layer is divided into local softmax (LS), inter-sub-vector reduction (IR), and global scaling (GS) sub-layers, each with a single kernel. In the dotted box, the figure on the left shows the work distribution of the TBs for each kernel. The figure on the right shows the operations and data access patterns within the TBs. In the column on the right, we denote all notations based on the row vector instead of the attention matrix, even if each TB of the kernel is assigned with a two-dimensional matrix. Edges represent dataflows within the TBs and between the kernels.

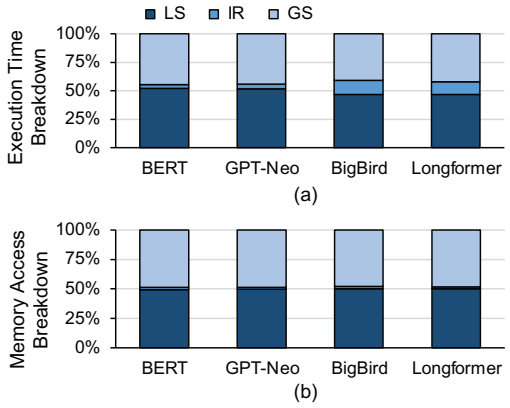

Figure 5: Proportion of each sub-layer within decomposed softmax processed on an A100 GPU: (a) execution time breakdown, and (b) off-chip memory access breakdown

outputs X', the result of the softmax operation on the sub-vector in addition to normalization, and outputs m' and d' equal to the number of allocated rows. Because the same of X' is identical to that of X and given that there are N_{sv} elements for m' and d' per row vector, considering all rows and heads, the size of X' is L^2H_{num} and the size of m' and d' is $LN_{sv}H_{num}$.

The IR kernel receives m' and d' from the LS kernel and computes m and d, after which it generates the reconstruction factor r' from m, d, and m'. The reconstruction factor is a factor multiplied by each value when the GS kernel operation is performed. Similar to m' and d', there is one reconstruction factor per sub-vector; thus, the total number of r' elements in the attention matrix of all heads is $LN_{sv}H_{num}$.

The GS kernel receives X' and r' as inputs and produces Y through the element-wise scaling of X' with r'. The number of r' elements loaded by a single TB depends on N_{sv} and the number of rows to be processed, but those numbers are relatively small. The total number of r' elements is $1/T$ of the number of elements in X', a small number when T is 32 or more. The data access pattern of the GS kernel is simple as the operation is conducted in an element-wise manner, with the same r' reused within all elements of the sub-vector.

Fig. 5 shows the execution time breakdown of the decomposed sub-layers. The LS and GS kernels account for most of the execution time. The decomposed kernels are memory-intensive with little data reuse; thus, the performance is bounded by the off-chip memory accesses. Compared to the LS and GS kernels, for which the sweep data are identical in size to the attention matrix, the IR kernel only requires the sweeping of data for small-sized intermediate values. As the number of the intermediate values is approximately $T\times$ fewer than that of the attention matrix, the proportion of IR is less than 12.5% in terms of time. More memory sweeps are needed to process the

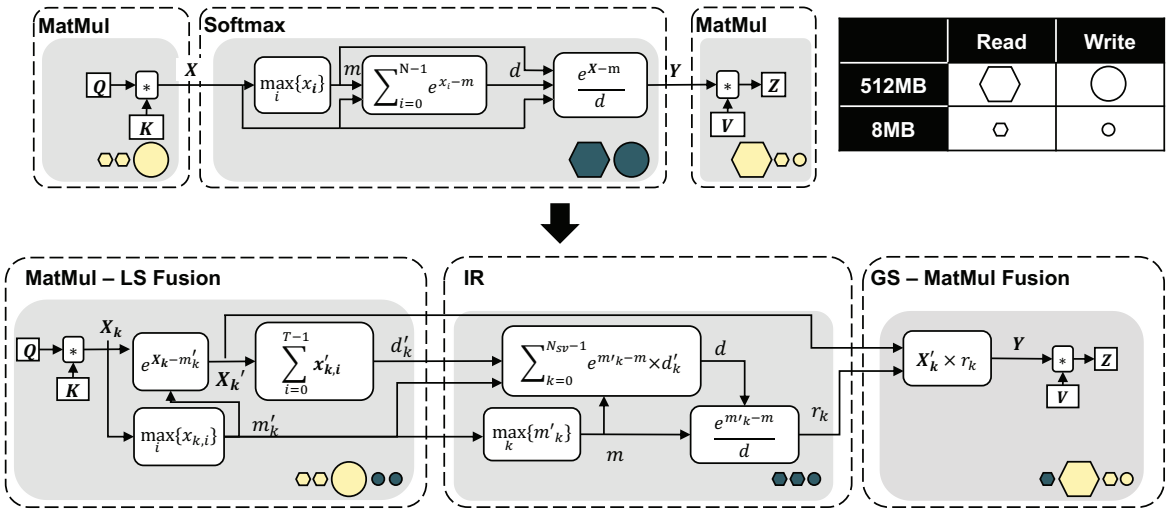

Figure 6: Illustration of softmax recomposition on the inference. We omit the scale and mask layers for convenience of explanation and follow the notations used in Fig. 4. Gray and white boxes indicate operations and computational nodes inside the kernel, respectively. We present data dependency with arrows and memory sweeps as circles and hexagons. We assumed for the BERT-large model values of L and T of 4,096 and 64, respectively, with half-precision. The illustration format of this figure is referenced from [16].

sub-layers compared to the original softmax layer, but if the LS and GS kernels are fused with their adjacent kernels, the total number of off-chip memory accesses would be reduced significantly.

3.3. Softmax Fusion

Kernel fusion is a method that aggregates multiple GPU kernels into one kernel. Typically, the kernel loads the data to be processed from the off-chip memory to the shared memory or the register file, performs the operations, and stores the result back in the off-chip memory. If the output from each TB of the former kernel fits into the input of each TB of the latter kernel, off-chip memory access between the two adjacent kernels can be eliminated by fusion.

We propose the fusion of the LS sub-layer with its preceding MatMul and of the GS sub-layer with its subsequent MatMul. This is feasible as the operation preceding the softmax kernel produces the kernel's input and the operation following the softmax kernel consumes the output of the kernel as the LHS matrix. Decomposition enables both the LS and GS sub-layers to be fused with their adjacent layers. By setting T of the LS kernel equal to the output tile width of the MatMul kernel, the LS kernel can be fused to its preceding MatMul kernel. Similarly, if element-wise scaling can be performed prior to the MatMul operation by providing r' corresponding to X' allocated to the TB of the MatMul kernel following the GS kernel, the GS kernel can easily be fused with its subsequent MatMul kernel. As most of the off-chip memory accesses occur in the LS and GS kernels among the decomposed softmax kernels, fusing the two kernels can greatly reduce the number of off-chip memory accesses.

Fig. 6 illustrates how the operation and off-chip memory accesses change after softmax fusion is applied to the SDA block. We omit the Scale and Mask layers in Fig. 6 for convenience of explanation. Because these layers perform element-wise operations, they are generally fused with their adjacent MatMul operations and do not affect the data movement. In the figure, the gray and white boxes indicate the operations and computational nodes inside the kernel, respectively. Arrows indicate the data movement direction. Circles and hexagons represent memory accesses for read and write operations, respectively.

The original softmax requires a total of four off-chip memory accesses to the attention matrix between the softmax layer and the adjacent MatMul. As mentioned earlier, the size of the attention matrix for all heads is too large to be stored in the on-chip memory; accordingly, a kernel that uses the attention matrix as the input or output must read or write the attention matrix from the off-chip memory each time. In the original softmax kernel, as each row vector is stored in the shared memory or in the register file in the SM and reused during computation, memory accesses due to the attention matrix do not occur between computational nodes in the softmax kernel.

Kernel fusion significantly reduces the number of off-chip memory accesses. When softmax decomposition and fusion of the LS and GS kernels are applied to adjacent MatMul kernels, the softmax layer no longer requires off-chip memory access to the attention matrix, which halves the memory access of the attention matrix (large circles and hexagons in Fig. 6). In contrast, the IR kernel cannot be fused. Therefore, it operates as a separate kernel, causing additional off-chip memory accesses to read and write m', d', and r'. As mentioned above, m', d', and r' are $1/T$ of the size of the attention matrix, where T is identical to the output tile width of the preceding MatMul. The MatMul operation that utilizes the outer product dataflow sets the

size of the output matrix tile as large as possible to process the operation efficiently. We empirically observed that in general, in MatMul kernels for transformers, T is set to a large number (greater than 64); thus, the overhead due to memory accesses for m', d', and r' is negligible.

3.4. Applying softmax fusion to sparse attention

Sparse attention differs from dense attention in how it processes the layers of the SDA block, as only a few elements in the attention matrix in the SDA block are left as non-zeroes. An appropriate sparse format is needed to represent a sparse attention matrix effectively. Earlier works [10, 36] demonstrated that the block-sparse format is efficient for representing the sparse attention matrix and for performing operations inside the SDA block. In this format, sparsity is defined in the unit of square blocks. Therefore, the unit for skipping the zero elements is not an element but a square block. Block-sparse computation, a block-sparse format-based operation, guarantees dense computation within a block and therefore has an advantage in that it can utilize a tensor core within a block and increase the data reuse in the SM.

Softmax recomposition can be applied to sparse attention in the same way as dense attention. The MatMul operation of the block-sparse computation operates identically to dense attention, except for element-wise post-processing operations for masking the zero-value square blocks. Therefore, applying softmax fusion is similar to dense attention and can be implemented with a straightforward modification during the implementation of fusion in dense attention. We implemented a block sparse MatMul kernel that operates identically to DeepSpeed [36] and applied softmax fusion to the kernel.

4. Experimental setup

We evaluated the inference performance of several transformer models using a half-precision floating-point (FP16) operation. We used BERT, GPT-Neo [4], BigBird, and Longformer. BERT is well known for dense attention, and BigBird and Longformer are representative models for sparse attention. There are models with both dense and sparse attention, the most popular being the GPT-3 [5] model. Because the source code of GPT-3 is not publicly available, we used GPT-Neo, an open-source model with a structure similar to that of GPT-3. We used a large model for all model tests (BERT-large, GPT-Neo-1.3B, BigBird-large, and Longformer-large) and set the model parameters according to the pre-trained model from HuggingFace [43]. We used the long document dataset TriviaQA [15] for all experiments.

We used three GPUs (A100 [26], Geforce RTX 3090 [27], and Tesla T4 [23]) running the PyTorch 1.8.1 framework [30] with CUDA toolkit 11.1 [24]. We mainly used the results of A100 throughout the evaluation. When showing the performance improvement, the results are also shown for RTX 3090 and T4, which have different hardware characteristics from A100. The hardware specifications are shown in Table 1. We measured the execution time and the off-chip memory accesses using NVIDIA Nsight Compute [28].

TABLE 1: Specifications of the GPUs used in the evaluation. *Peak rates are based on the GPU base clock. **The recent GPU architecture combines the L1 data cache and shared memory functionality into a single memory block.

	A100	RTX 3090	T4
Memory Bandwidth (GB/s)	1,555	936.2	320
TFLOPS (FP16 CUDA)*	42.3	29.3	24.0
TFLOPS (FP16 Tensor)*	169	58	24.0
L1 D$ per SM (KB)**	192	128	64
L2 (MB) $	40	6	4

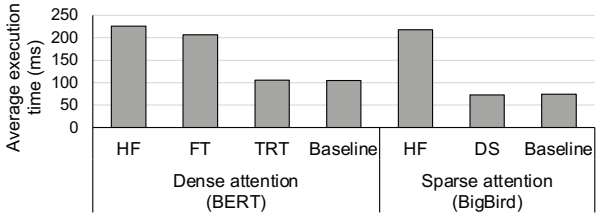

Figure 7: Average execution time for existing GPU libraries and our baseline.(HG: HuggingFace [43], FT: FasterTransformer [25], TRT: TensorRT [39], and DS: DeepSpeed [36]) We evaluated BERT-large and BigBird-large (L = 4,096, batch size=1) using TriviaQA dataset.

We used a library that efficiently processes the Transformer model called DeepSpeed (v0.5.1, released by Microsoft) as the baseline implementation. This library calls and processes the cuBLAS API [22] for operations processing dense formats and uses the kernels generated by Triton [38] for operations processing sparse formats.

We partially changed the baseline implementation of the SDA block in DeepSpeed. We replaced MatMul of the SDA block with a kernel that is feasible for use with softmax recomposition while guaranteeing performance similar to that in the existing case. We changed the dense attention implementation with CUTLASS [2] (v2.5.0) and the sparse attention implementation with our custom MatMul kernel, showing a performance difference of less than 8% from that of DeepSpeed. We replaced the baseline softmax implementation in the dense attention with the softmax kernel of TensorRT [39] (v8.0), which outperforms DeepSpeed. Fig. 7 presents the results of a performance comparison between the existing GPU libraries and our baseline implementation. TensorRT and DeepSpeed are the best performing libraries for each of dense and sparse attention operation, with only minor differences between those and our baseline. We also compared our baseline with AutoTVM [6] (v0.10), a deep learning compiler, and confirmed that our baseline is 1.49x faster than it for BERT-large. Specifically, in BERT, our baseline and TensorRT were similar (difference less than 1%) in terms of the execution time, and in GPT-Neo, BigBird, and Longformer, our baseline was slightly slower than the original DeepSpeed by 2%.

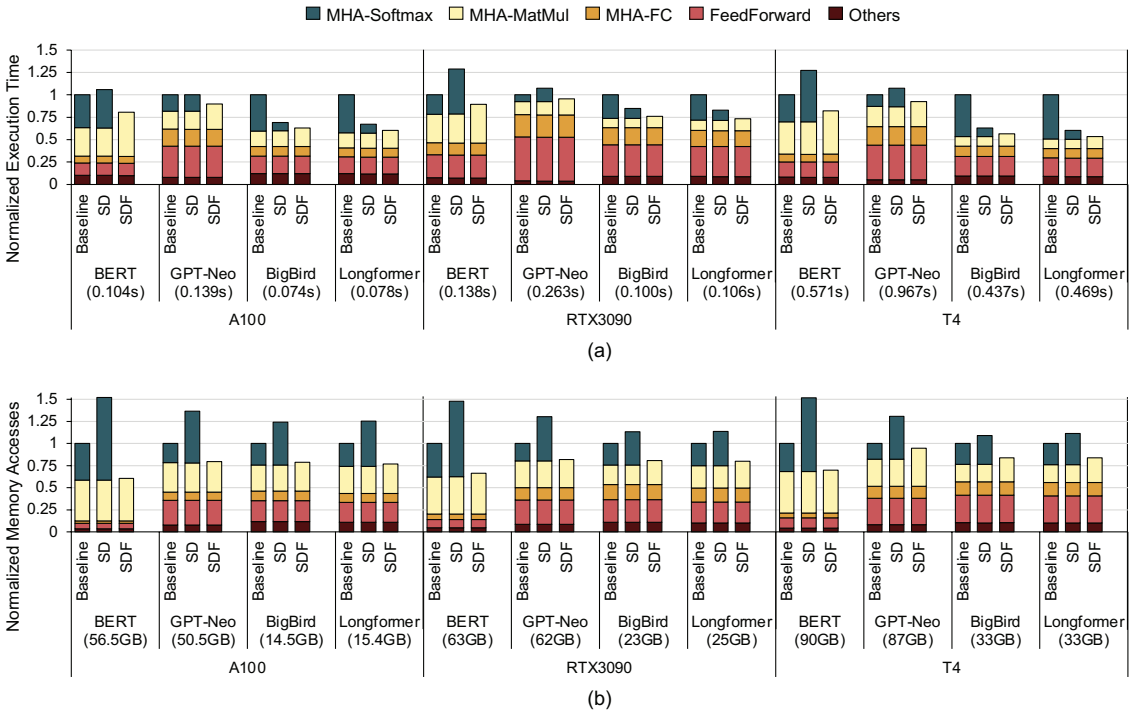

Figure 8: (a) Execution time and (b) memory accesses per iteration (L = 4,096, batch size=1) by applying SD (softmax decomposition), SDF (softmax decomposition and fusion) for BERT, GPT-Neo, BigBird, and Longformer. We normalized the values of the SD and SDF methods based on the baseline execution time and memory access. The absolute execution time and the memory access for each baseline are shown below the model names.

5. Evaluation

We evaluated the effectiveness of softmax recomposition for BERT, GPT-Neo, BigBird, and Longformer. We present the performance while incrementally applying the softmax decomposition and softmax fusion, after which we analyze the effect of softmax recomposition according to the batch size and L. In summary, for each of the BERT, GPT-Neo, BigBird, and Longformer models, our softmax recomposition strategy achieves speedups of 1.25×, 1.12×, 1.57×, and 1.65×, respectively.

5.1. Performance evaluation

When softmax decomposition (SD) is applied, the performances of BERT and GPT-Neo drop by 0.94× and 0.99×, whereas the performances of BigBird and Longformer increase correspondingly by 1.44× and 1.49× (see Fig. 8). By decomposing the softmax layer, the off-chip memory traffic to the attention matrix is doubled. This is achieved at the cost of having to change the data access patterns to process the softmax layer for each sub-vector. In dense attention with a large attention matrix, such as BERT and GPT-Neo, the performance drops due to the increased number of off-chip memory accesses. In contrast, in the sparse attention assessments, such as with BigBird and Longformer, changes in the data access pattern increase the memory bandwidth utilization, thus improving the performance.

Memory bandwidth utilization improves because the number and location of nonzeros for each row vector are not deterministic when using the softmax layer in sparse attention. In the GPU programming model, all thread blocks (TBs) in a single kernel are allocated the same amount of resources. In the baseline configuration, each TB is allocated memory space equal to the size of the row vector in the worst case to ensure that all row vectors remain in on-chip memory during the softmax layer operation. This conservative allocation scheme causes the TBs to occupy considerable amounts of SM resources. However, because the row vector is sparse, most threads in the TB do not issue memory instructions, thus lowering the memory utilization rate. SD allocates memory space for each TB according to the size of the sub-vector, not L (the length of an entire row vector). Therefore, resources are not allocated to sub-vectors for which all elements are zero. This finer-grain allocation method increases the efficiency of SM resources and allows more threads to issue memory instructions at the same time.

Applying fusion to the decomposed softmax (SDF) and the adjacent MatMuls improves the performance of BERT, GPT-Neo, BigBird, and Longformer by 1.32×, 1.11×, 1.10×, and 1.11× compared to decomposition, mainly due to the reduced number of off-chip memory accesses of the softmax layer by about 1.58× to 2.51× through kernel fusion, thus significantly decreasing the execution time of the softmax layer. The number of off-chip memory accesses by MatMul, which is increased due to the intermediate values (m', d', r') generated during the softmax sub-layer operations, is less than 9.3% of that of the existing softmax layer. The

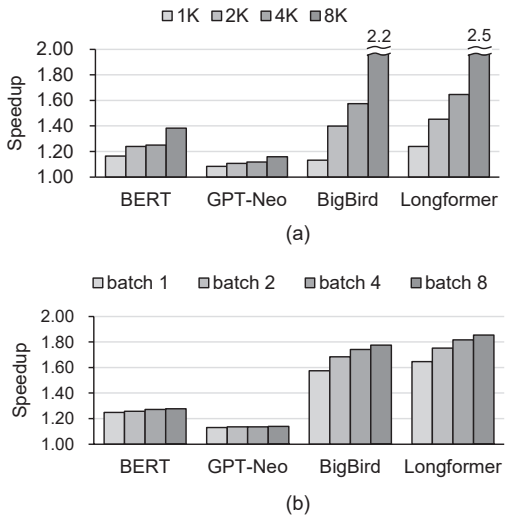

Figure 9: Speedups of transformer models compared to the baseline with various (a) sequence lengths and (b) batch sizes on A100

execution time of the remaining IR kernel is negligible (less than 2.9% of the original softmax layer). The execution time of MatMul increases by approximately 28%∼55% due to operations such as the exponent and sum operations, and the finding of the maximum value transferred from the adjacent softmax sub-layers to MatMul.

On other GPUs with various specifications, applying softmax recomposition improves the performance. On the RTX 3090 GPU, our method achieves speedups for BERT, GPT-Neo, BigBird, and Longformer of 1.12×, 1.05×, 1.32×, and 1.36×, outcomes relatively lower than that by the A100 GPU. The speedup differences stem from the execution time ratio of the softmax layers in the baseline. As shown in the middle of Fig. 8(a), MatMul, FC, and FeedForward, which are processed by the tensor core, utilize a significant proportion of the baseline's execution time. They have a larger proportion of the execution time on RTX 3090, which has a smaller ratio of tensor core FLOPS to the memory bandwidth than A100. In other words, on RTX 3090, the proportion of softmax layers in the execution time is slightly reduced compared to that of A100. However, it is still 7∼28%, and softmax recomposition improves the performance by 1.05× to 1.36×.

On a T4 GPU, for the same reason as RTX 3090, the proportions of softmax layers in the execution time for BERT and GPT-Neo are less than that of A100, but those for BigBird and Longformer are greater because the degree of improvement in memory bandwidth utilization is greater on T4, where the number of threads processible by SM is relatively small. Therefore, applying softmax recomposition on a T4 GPU achieves speedups of 1.22×, 1.08×, 1.77×, and 1.87× for BERT, GPT-Neo, BigBird, and Longformer, respectively.

5.2. Case studies

The sequence length L and the batch size change the softmax portion of the baseline in the execution time, which directly affects the effectiveness of softmax recomposition. Fig. 9(a) shows the speedup as a function of L when softmax recomposition is applied. As mentioned in Section 2.3, a longer L for the dense attention model increases the proportion of the softmax layer in the execution time, meaning that the speedup increases as L becomes longer in BERT and GPT-Neo. For BigBird and Longformer, the speedup increases as L becomes larger due to the high sparsity of the attention matrix. For sparse attention, even if L increases, the computation and memory access proportions utilized by the softmax layer overall remain unchanged. However, the sparsity of the attention matrix increases linearly. High sparsity lowers memory bandwidth utilization, which leads to an increase in the proportions of the execution time used by the softmax layers.

A large batch size increases the speedup in the sparse attention model (see Fig. 9(b)). When the batch size is large, MatMul in the SDA block is processed more efficiently, which reduces the proportion of MatMul in the execution time and increases the proportion of softmax relatively. In sparse attention, the MatMul operation following the softmax layer takes a sparse attention matrix as input. Because the number of nonzeros per row of an attention matrix is irregular, each TB has a different processing time, causing a load-imbalance problem. A large batch size can alleviate this load-imbalance problem by increasing the number of TBs [20]. When the batch size increases from 1 to 8, the portion of MatMul decreases from 17% to 10%, and the portion of softmax increases from 40% to 48% in the execution time.

6. Discussion

Applying softmax recomposition to training: Softmax recomposition can be applied to the forward pass of training because the softmax layer does not need to store the input in off-chip memory. In general, the backward pass of a layer produces derivatives from the inputs of the layer, which forces the storage of all inputs into off-chip memory upon the forward pass. Softmax recomposition refers to the idea of improving the performance by skipping the storing of the input in off-chip memory, but it is still applicable due to the mathematical nature of the derivative in the softmax layer. Suppose an error of training is E, the input vector is $X = \{x_0, x_1, .. , x_{L-1}\}$, and the output vector is $Y = \{y_0, y_1, .. , y_{L-1}\}$ for the softmax layer. The derivative of the error is as follows:

$$\frac{\partial E}{\partial x_k} = \sum_{i=0}^{L-1} \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial x_k}$$
$$\frac{\partial y_i}{\partial x_k} = \begin{cases} y_i(1 - y_i) & i = k \\ -y_i y_k & i \neq k \end{cases} \quad (3)$$

Equation (3) is proved in [1]. The derivative of the error is expressed as y, which means that the input of the softmax layer does not need to be stored in off-chip memory.

7. Related work

GPU kernel fusion: Many studies sought to optimize memory-bound layers on GPUs by applying the kernel fusion approach. [16] proposed a method that reduces memory accesses by dividing a batch normalization layer into sub-layers and fusing them to their neighboring convolution layers. [14, 36, 39] explored various layer-fusion opportunities within the transformer model. They mainly focused on optimizing element-wise layers, such as the bias add, mask, and scale types. They proposed fusing an element-wise layer with MatMul or a normalization layer such as a softmax and a LayerNorm layer. We are the first to propose the fusion of MatMul and a softmax layer. Moreover, our method can be applied to existing fusion methods. [25, 36, 39] provide a kernel function that fuses MHA blocks, including softmax, into a single kernel. However, this kernel function is only applicable when the input sequence is short (e.g., less than 384 in [25]).

Softmax optimization: [21] proposes the processing of the max and normalization terms of the softmax operation for each sub-vector on the GPU and the fusing of the softmax layer with a TopK operation, which finds the top K values among the vector. However, this work did not analyze differences in data access patterns between the softmax layer and adjacent MatMul operations in the transformer model. It does not suggest a method for the fusion of MatMul and softmax. [37] designed an accelerator to process softmax using an algorithm, similar to our work. While this work implemented a unit to accelerate the softmax layer, we obtained an equivalent effect only by changing the software in the existing hardware (GPU). [9] increases SM utilization when processing the softmax layer by allocating a batch of row vectors to each thread block. However, this approach did not consider models with long sequence lengths and did not optimize the memory accesses of the attention matrix. The method supports sequence lengths up to 1,024, and even if the length is extended, the method does not reduce the number of memory accesses of the attention matrix, which is an important factor for long sequence lengths.

8. Conclusion

We have proposed a method to accelerate the transformer models on GPUs, the most popular hardware platform for deep learning, through recomposing softmax. As the data access patterns of softmax layers differ from those of adjacent operations, applying the kernel fusion technique naïvely can be challenging. To solve this problem, we decompose softmax into sub-layers, restructure their access patterns, and fuse them to the preceding and subsequent matrix multiplication operations. We demonstrated performance improvements in representative transformer models using dense/sparse attention. By applying softmax recomposition, we achieved up to 1.25×, 1.12×, 1.57×, and 1.65× speedups by significantly reducing off-chip memory accesses when inferring BERT, GPT-Neo, BigBird, and Longformer on modern GPUs.

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), National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2020R1A2C2010601), 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, the Interdisciplinary Program in Artificial Intelligence, and the Research Institute for Convergence Science, SNU.

References

[1] E. Alpaydın, *Introduction to Machine Learning 3rd edition.* The MIT Press, 2014.

[2] 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.

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

[4] S. Black, G. Leo, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow," 2021.

[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 *Advances in Neural Information Processing Systems*, 2020.

[6] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," *Advances in Neural Information Processing Systems*, 2018.

[7] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating Long Sequences with Sparse Transformers," *arXiv:1904.10509*, 2019.

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

[9] J. Fang, Y. Yu, C. Zhao, and J. Zhou, "TurboTransformers: An Efficient GPU Serving System for Transformer Models," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[10] S. Gray, A. Radford, and D. P. Kingma, "GPU Kernels for Block-Sparse Weights," 2017.

[11] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee, and D.-K. Jeong, "A^3: Accelerating attention mechanisms in neural networks with approximation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[13] J. Ho, N. Kalchbrenner, D. Weissenborn, and T. Salimans, "Axial attention in multidimensional transformers," *arXiv:1912.12180*, 2019.

[14] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data Movement Is All You Need: A Case Study on Optimizing Transformers," in *Proceedings of Machine Learning and Systems*, 2021.

[15] M. Joshi, E. Choi, D. Weld, and L. Zettlemoyer, "TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017.

[16] W. Jung, D. Jung, B. Kim, S. Lee, W. Rhee, and J. Ahn, "Restructuring Batch Normalization to Accelerate CNN Training," in *The Conference on Systems and Machine Learning*, 2019.

[17] S. Lee, S. Hwang, M. J. Kim, J. Choi, and J. Ahn, "Future Scaling of Memory Hierarchy for Tensor Cores and Eliminating Redundant Shared Memory Traffic Using Inter-Warp Multicasting," *IEEE Transactions on Computers*, 2022.

[18] B. Li, S. Pandey, H. Fang, Y. Lyv, J. Li, J. Chen, M. Xie, L. Wan, H. Liu, and C. Ding, "Ftrans: energy-efficient acceleration of transformers using fpga," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2020.

[19] 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.

[20] P. Micikevicius, "GPU Performance Analysis and Optimization," https://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf, 2012.

[21] M. Milakov and N. Gimelshein, "Online Normalizer Calculation for Softmax," *arXiv:1805.02867*, 2018.

[22] NVIDIA, "cuBLAS," https://docs.nvidia.com/cuda/cublas/index.html.

[23] NVIDIA, "NVIDIA TURING GPU Architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018.

[24] NVIDIA, "CUDA Toolkit Documentation," https://docs.nvidia.com/cuda/archive/11.1.0/, 2020.

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

[26] 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.

[27] NVIDIA, "NVIDIA AMPERE GA102 GPU Architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf, 2020.

[28] NVIDIA, "Nsight Compute Documentation," https://docs.nvidia.com/nsight-compute/NsightCompute/index.html, 2021.

[29] N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, A. Ku, and D. Tran, "Image transformer," in *Proceedings of the 35st International Conference on Machine Learning*, 2018.

[30] 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 *Advances in Neural Information Processing Systems*, 2017.

[31] H. Peng, S. Huang, S. Chen, B. Li, T. Geng, A. Li, W. Jiang, W. Wen, J. Bi, H. Liu, and C. Ding, "A length adaptive algorithm-hardware co-design of transformer on fpga through sparse attention and dynamic pipelining," in *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022.

[32] P. Qi, E. H.-M. Sha, Q. Zhuge, H. Peng, S. Huang, Z. Kong, Y. Song, and B. Li, "Accelerating framework of transformer by hardware design and model compression co-optimization," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021.

[33] J. Qiu, H. Ma, O. Levy, S. W. tau Yih, S. Wang, and J. Tang, "Blockwise self-attention for long document understanding," *arXiv:1911.02972*, 2019.

[34] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving Language Understanding by Generative Pre-Training," https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf, 2018.

[35] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models are Unsupervised Multitask Learners," http://www.persagen.com/files/misc/radford2019language.pdf, 2019.

[36] 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.

[37] J. R. Stevens, R. Venkatesan, S. Dai, B. Khailany, and A. Raghunathan, "Softermax: Hardware/Software Co-Design of an Efficient Softmax for Transformers," *arXiv:2103.09301*, 2021.

[38] 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*, 2019.

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

[40] 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 *Advances in Neural Information Processing Systems*, 2017.

[41] G. Wang, Y. Lin, and W. Yi, "Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU," in *2010 IEEE/ACM International Conference on Green Computing and Communications International Conference on Cyber, Physical and Social Computing*, 2010.

[42] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[43] 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. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-Art Natural Language Processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020.

[44] 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 *Advances in Neural Information Processing Systems*, 2020.